universität wien

DIPLOMARBEIT

Titel der Diplomarbeit

Max-Flow Min-Cut

angestrebter akademischer Grad

Magister der Naturwissenschaften (Mag. rer. nat.)

Verfasser:	TIMON THALWITZER
Matrikel-Nummer:	0103857
Studienkennzahl lt. Studienblatt:	A 405
Studienrichtung lt. Studienblatt:	Mathematik
Betreuer:	Univ.Prof. Dr. Christian Krattenthaler

Wien, am 11.11.2008

Abstract

This is a monograph on network flows. Network flow theory constitutes a widely applied mathematical field, which is used in practice in fields like economics, traffic routing, urban planning, telecommunication, and countless more. From a mathematical point of view, it is interesting that networks can be treated equally well by methods from graph theory and from linear programming. In fact, this text deals with a certain kind of linear programs with a fairly specific structure. Both of these seemingly different approaches have their advantages and disadvantages.

Networks and flows are thoroughly studied notions, and there has been done a lot of research on them since the 1950s. A vast body of literature on both network flows and linear programming exists. In this thesis, I focus on the Max-Flow Min-Cut Theorem, as well as on describing various algorithms for constructing maximal flows in a given network. I also try to present the combinatorial and the linear programming point of view as homogeneous as possible, using a notation that lends itself well to both.

One aspect that might appear a little unusual is that I formulated all definitions and theorems using throughout a fairly general network model, whereas in most works done on this topic, a more restricted definition for networks is used, with the aim of keeping the proofs and calculations shorter. Since the more general case can easily be reduced to the more specific one, my approach constitutes by no means the gain of new or unknown theorems. But all of the proofs extended very naturally to the generalizations without becoming more complicated, and hence it made sense to me to present the material in this way.

Zusammenfassung

Die vorliegende Arbeit ist eine Monographie über Netzwerke. Die Theorie der Netzwerkflüsse ist ein mathematisches Gebiet, das weitverbreitete Anwendung findet in Feldern wie Ökonomie, Verkehrs- und Städteplanung, Telekommunikation und vielen anderen mehr. Aus mathematischer Sicht ist es interessant, dass Netzwerke gleichermaßen mit Methoden der Graphentheorie, als auch mit solchen aus der linearen Programmierung behandelt werden können. Im Grunde behandelt der vorliegende Text eine bestimmte Art von linearen Programmen, die eine sehr spezifische Struktur aufweisen. Jeder dieser scheinbar unterschiedlichen Zugänge hat seine Vor- und Nachteile.

Netzwerke und Netzwerflüsse stellen ausführlich behandelte Begriffe dar, und sind seit den 1950ern Gegenstand einer intensiven Forschungstätigkeit. Ein umfangreicher Bestand an Literatur zu sowohl Netzwerkflüssen als auch zur linearen Programmierung ist heute verfügbar. In dieser Diplomarbeit behandle ich vor allem das Max-Flow Min-Cut Theorem, als auch verschiedene Algorithmen, die zum Auffinden von maximalen Netzwerkflüssen dienen. Ich versuche dabei, die kombinatorische Herangehensweise und jene von der linearen Programmierung herkommende so homogen wie möglich zu präsentieren und dabei eine Notation zu verwenden, die beiden Zugängen gerecht wird.

Ein Aspekt der etwas ungewöhnlich erscheinen könnte ist, dass ich alle Definitionen und Sätze unter Verwendung eines sehr allgemeinen Netzwerkbegriffs formuliere, während in den meisten Arbeiten zu diesem Thema ein engerer Netzwerkbegriff verwendet wird, welcher dazu dienen soll, die Beweise und Rechnungen kürzer zu halten. Da die allgemeinere Definition leicht auf die speziellere zurückgeführt werden kann, ergibt sich durch meine Vorgehensweise keinesfalls der Gewinn neuer oder unbekannter Sätze. Jedoch ließen sich alle Beweise auf sehr natürliche Weise verallgemeinern ohne dabei komplizierter zu werden, und deshalb erschien es mir sinnvoll, das Thema in dieser Art und Weise darzustellen.

Contents

Chapter 1

Introduction and Preliminaries

In Section 1.1, I give a brief overview of the history and development of network flow theory and its applications, as well as a rough description of the main results that are presented in this thesis. I also outline the material included in each of the upcoming four main chapters.

In the subsequent Sections 1.2–1.4, I shall introduce some rather general notions, mainly from graph theory and linear programming, that are needed in the remainder of the text. As most of the material of these sections is only preliminary and will probably be familiar to some readers, it might well be skipped, or alternatively be consulted on demand.

1.1 Introduction

Networks can be observed in all areas of modern life. They occur in rather diverse forms, ranging from small scale objects, like electronic circuits, to fairly large scale structures, like international trade relations. The common abstract background of what is casually referred to as a network is the following: some quantity (be it electricity, water, information, goods, money, people,...) is moved along an underlying supporting medium (circuit boards,

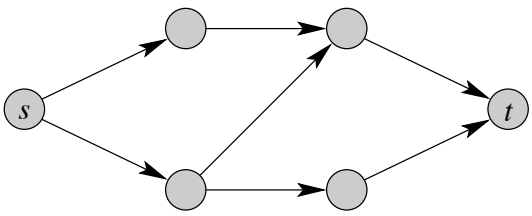

Figure 1.1: A network.

pipelines, roads, antennas and the air between them,...).

The circles and lines depicted in Figure 1.1 on page 3 are meant to grasp the underlying universals of all these instances. This 'generalized' network essentially consists of some points, called *vertices*, and lines (or arrows), called *arcs*, connecting some of them. In mathematical terminology, this kind of object is usually referred to as a *graph*. A *network* in the mathematical sense is a graph with some additional properties. For example, one might specify two 'special' vertices (like s and t in the above figure). Then a task that might be of interest would be to find all possible connections between s and t. In addition, one could assign numbers to all the arcs, representing the capacity of the respective arc; in other words, the amount of some quantity that can at most be sent through the arc. A question that naturally arises asks for the maximum amount of that quantity that can altogether be sent from s to t. Furthermore, if another set of numbers representing per-unit costs of using each of the arcs is assigned to the network, then a problem worth considering might be how a certain amount of the quantity can be sent from s to t such that the total costs are minimized. These are the kind of questions that are the subject of network flow theory.

The hour of birth of graph theory is usually considered to be 1736. In that year, Leonhard Euler (1707–1783) gave a solution to the well-known "Königsberger Brückenproblem"[1]. The beginnings of network flow theory can only be traced back to somewhat more recently: the works of Gustav Robert Kirchhoff (1824–1887) and other pioneers of electrical engineering laid the ground for modern network flow theory as it is pursued by researchers today. A nice exposition of how electrical circuits can be described in terms of mathematical network theory is found in [24], among many other books on the topic. Yet more recently started the formal and detailed analysis of the abstract mathematical notion of networks as it is still in use now. In the 1950s, the term *flow* first came up, namely in the RAND group[2], then being the working place of, among others, George Bernard Dantzig (1914–2005), Delbert Ray Fulkerson (1924–1976), and Lester Randolph Ford Jr. (*1927). Flow is the mathematical take on what I described earlier as the quantity that is to be transported along the arcs of the network. In Section 5.2.1, I present the practical 'application' that ultimately gave rise to the first comprehensive monograph on network flows: *Flows in Networks*, written by Ford and Fulkerson in 1962 (see [8]). In the past five decades, especially since the inception of complexity theory as a mathematical discipline in its own

[1] also known as "Seven Bridges of Königsberg"

[2] Research ANd Development; according to [25] "a nonprofit global policy think tank first formed to offer research and analysis to the United States armed forces". Formed in 1946. Also see [19].

right, efforts have mainly focussed on the development of efficient algorithms for the practical and computational solution of the problems hinted at above.

This thesis comprises four main chapters. Chapters 2 and 3 are both mostly concerned with one of the central theorems of network flow theory. Though not difficult to prove, it has far-reaching im- and applications, of both inner- and outer-mathematical nature.

In Chapter 2, the discussion is a purely graph theoretical one. Using a definition of networks that allows upper and lower capacity bounds for arcs, arbitrary sets of source and sink vertices, and balance constraints for all other vertices, the basic notions and relations of network theory are developed and ultimately led to a detailed proof of the Max-Flow Min-Cut Theorem. Several illustrating examples are provided along the way.

Chapter 3 uses a different approach. The same theorem is proved again, but with techniques from linear programming.

In Chapter 4, various successful algorithms for flow maximization in a network that have been designed over the years are presented. Here again the position of network flows in the intersection of graph theory and linear programming becomes apparent; there are algorithms derived from combinatorial considerations and algorithms derived from linear programming methods.

Finally, in Chapter 5, I first demonstrate how the Max-Flow Min-Cut Theorem can be utilized as a tool to prove other graph theoretical theorems. Then, to give an idea of how network flow techniques are typically applied in practical contexts, I shall state some characteristic examples.

1.2 General Terminology

In this section I gather together a few basic notions that could be regarded as 'general mathematical knowledge'. I still wanted to include them here to have a quick reference at hand, if needed.

Numbers, Sets and Functions

$\mathbb{N} = \{0, 1, 2, \dots\}$ is the set of natural numbers. $\mathbb{R}$ is the set of real numbers. To simplify some notations, I set

$$\overline{\mathbb{R}} \coloneqq \mathbb{R} \cup \{-\infty, \infty\}.$$

Here the infinity-symbol is used in the familiar way: $-\infty < r < \infty$ for all $r \in \mathbb{R}$.

Let X be any set and $f\colon X \to \mathbb{R}$ any function. For a subset $Y \subseteq X$, define $f(Y) \coloneqq \sum_{y\in Y} f(y)$.

If $X = \{x_i\}_{i\in I}$ for an arbitrary index set I, then set $X^{-1} \coloneqq \{x_i^{-1}\}_{i\in I}$. This is a purely formal definition; for the moment it does not matter whether or not the symbols x_i^{-1} have a specific meaning or not.

Ordered Sets

Let X be some set. A set of ordered pairs of elements from X is called a *relation* or a *partial relation* on X. For such a a relation R, often xRy is written instead of $(x, y) \in R$.

A relation R on X is called *total* if for every pair $(x, y) \in X \times X$, at least one of xRy or yRx holds. It is called *reflexive* if xRx for all $x \in X$. It is called *transitive* if for all $x, y, z \in X$, xRy and yRz imply xRz. It is called *anti-symmetric* if for all $x, y \in R$, xRy and yRx imply $x = y$.

A relation $\leq$ is called a *pre-order* on X if $\leq$ is reflexive and transitive. A pre-order $\leq$ is called an *order* on X if $\leq$ is anti-symmetric. In that case, X is referred to as a *partially ordered set*. If in addition $\leq$ is total, then X is called an *ordered set*.

Complexity and Algorithms

In the study of running times of algorithms (Section 4), I will need the following (a little informal) definitions, [3]

1.1 Definition. Let A be an algorithm with inputs from some set X. Let $f\colon X \to \mathbb{R}$. If there exists a constant $\alpha \geq 0$ such that A terminates its computations for input $x \in X$ after no more than $\alpha f(x)$ elementary steps (such as basic arithmetic operations), then A is said to *run in time* $O(f)$, or that its *running time* or (*time*) *complexity* is $O(f)$.

Let $g\colon X \to \mathbb{R}$ be another function. If there exists a constant $\alpha \in \mathbb{R}$ such that for every $x \in X$ the equation $|g(x)| \leq \alpha f(x) + \alpha$ holds, we write $g = O(f)$. Note that the so defined relation is transitive and reflexive, but not symmetric.

For further information on what 'algorithm' or 'elementary steps' means, or why this '$O(.)$-notation' is a widespread, and broadly accepted, way of measuring the efficiency of algorithms, I refer the interested reader to [14], [21], [1], [3], [6], [17], [18], [9], [13], or [2].

[3] They can be found in many textbooks. These versions are taken from [14, p. 6], respectively [21, p. 13].

Partitions

Let X be some set. A family of subsets $\{Y_i \mid i \in I\}$ (here I is an arbitrary index set), is called a *partition* of X if and only if the following two conditions hold:

$$\text{(i)} \qquad \bigcup_{i \in I} Y_i = X$$

$$\text{(ii)} \qquad i \neq j \Longrightarrow Y_i \cap Y_j = \emptyset.$$

1.3 Preliminaries on Graph Theory

As the terminology for this combinatorial field is (unfortunately) not standardized at all, I introduce briefly all the concepts and notions that will be utilized later on. Most of it is fairly basic material. I also provide numerous examples for nearly all of the definitions. Consequently, I think that even a reader who has never encountered a graph before should be able to comprehend this thesis (at least this is the aim of this section).

Detailed introductions to graph theory are readily available in numerous text books. Examples include [21], [2], [13], [1], [12], and [4].

1.3.1 Undirected Graphs

The two main objects of interest in graph theory are *undirected graphs* (or simply *graphs*) and *directed graphs* (or shorter *digraphs*). The difference between the two is whether or not the edges are oriented. In this thesis, we will mainly encounter certain directed graphs, but as undirected graphs do crop up in a few places, I provide the basic definitions for them as well.

Graphs

A *graph* or *undirected graph* is an ordered pair $G = (V, E)$, where V is some finite set, called the *set of vertices* or the *set of nodes* of G, and E is a family of unordered pairs $\{u, v\}$ of elements of V, called *the family of edges* of G. For a given graph G, I sometimes write $V(G)$ (respectively $E(G)$) for the set of vertices of G (respectively the family of edges of G). The elements of V (respectively E) are (as one could have guessed by now) called *vertices* or *nodes* (respectively *edges*). A pair occurring more than once in E is called a *multiple* edge. Two edges which are represented by the same pair of vertices are called *parallel*. An edge of the form $\{v, v\}$ (for some $v \in V$) is called

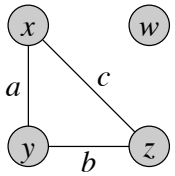

Figure 1.2: A simple graph.

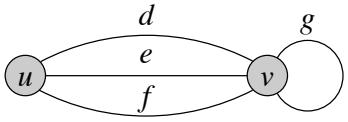

Figure 1.3: A non-simple graph.

a *loop*.[4] A graph with neither loops nor multiple edges is called *simple*. A graph which is not simple is simply called *non-simple*.

1.2 Example. In Figure 1.2, a graph is shown. Its vertices are represented by the little grey circles, its edges by the thin black lines connecting some pairs of vertices. I will always depict graphs in this fashion. If needed, the names of the vertices are written inside them for reference, as in the case of vertices x, y, z, or w. Similarly, the reference names of the edges are attached to them if convenient, as is done with a, b, and c. Note that the graph in Figure 1.2 is simple: it contains neither loops nor multiple edges.

The graph in Figure 1.3 is not simple: it contains the loop g. Furthermore, the edge $\{u, v\}$ is a multiple edge. Another way of stating this is to say that d, e, and f are parallel.

Incidence and Neighbours

The edge $e = \{u, v\}$ is said to *connect* u and v and to be *incident* with u and *incident* with v, whereas u and v are referred to as the *ends* of $\{u, v\}$. Each of them is said to be *incident* with $\{u, v\}$. If, on the other hand, $u \notin e$ holds, u and e are called *disjoint* from one another. Two edges e and f are said to be *incident* if $e \cap f \neq \emptyset$ and are called *disjoint* if $e \cap f = \emptyset$. Two vertices u and v are called *neighbours* if $\{u, v\} \in E$. For $U \subseteq V$, define

$$\delta_G(U) \coloneqq \{e \in E \mid |e \cap U| = 1\} \text{ and}$$
$$N_G(U) \coloneqq \{v \in V \backslash U \mid \exists u \in U \colon \{u, v\} \in E\},$$

the family of edges incident to U, respectively the set of U's neighbours. If $U = \{u\}$, I often leave out the braces. The subscript G is omitted when no confusion is to be expected.

Let $U, W \subseteq V(G)$. If for an edge e we have $e \cap U \neq \emptyset$, then we say e and U are *incident* to each other. If $e \cap W \neq \emptyset$ as well, then e is said to *connect* U and W. If $e \cap U = \emptyset$, then e and U are said to be *disjoint*.

[4] If edges are regarded as *sets*, then a loop would contain only one element, because then $\{v, v\} = \{v\}$. To avoid this, one can simply define edges as *multisets*, that can contain elements in multiplicities greater than one.

1.3 Example. I continue Example 1.2 by presenting instances of the just introduced notions, taken from Figures 1.2 and 1.3 on page 8: the edge a connects the vertices x and y. b is incident with y, as is a. Moreover, b is incident with c. z is an end of b. Both of the depicted graphs happen to contain not a single pair of disjoint edges. u and v are neighbours. We have $\delta(x) = \{a, c\}$ and $N(x) = \{y, z\}$.

Define $X \coloneqq \{y, z\}$. Then we can say that a is incident with X. x and X are connected by a. The same is true for $\{x\}$ and X. w and c are disjoint, and so are $\{w\}$ and c.

Subgraphs

A *subgraph* of a graph G is a graph H with $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Let this relation be denoted by $H \leq G$. A subgraph $H \leq G$ is called the *subgraph of G induced by W* if

$$\begin{aligned} &V(H) = W \quad \text{and} \\ &E(H) = \{a = \{u, v\} \in E(G) \mid u, v \in W\}. \end{aligned}$$

The set of subgraphs of a given graph G is a partially ordered set with respect to the just defined relation $\leq$.

1.4 Example. The graph from Figure 1.2 on page 8 (let us call it G) contains all of the following subgraphs:

$$\begin{aligned} H &\coloneqq (\{x\}, \emptyset) \\ I &\coloneqq (\{x, y\}, \emptyset) \\ J &\coloneqq (\{x, y\}, \{a\}) \\ K &\coloneqq (\{x, y, z\}, \{b, c\}) \end{aligned}$$

We have $H \leq I \leq J \leq G$ and also $H \leq I \leq K \leq G$. But J and K cannot be compared in this way; we have $J \not\leq K$ and $K \not\leq J$. The subgraph of G induced by $\{x, y\}$ is J (not I, for instance). On the other hand, the subgraph of K induced by $\{x, y\}$ is I. Note that for example the following ordered pair is not a subgraph of G:

$$L \coloneqq (\{x, y\}, \{a, b\}).$$

Although L satisfies $V(L) \subseteq V(G)$ as well as $E(L) \subseteq E(G)$, L fails to be a graph. This is because $b \not\subseteq E(L)$.

Paths

Let G be a graph. A *walk* in G is a sequence

$$P = (s = v_0, e_1, v_1, \ldots, e_n, v_n = t)$$

with $n \in \mathbb{N}$, $v_i \in V(G)$ for $i = 0, \ldots, n$, and $e_i \in E(G)$ an edge connecting v_{i-1} and v_i (i.e., $e_i = \{v_{i-1}, v_i\}$), for $i = 1, \ldots, n$. s is called the *starting* or *first vertex* of P, t its *end* or *last vertex*. Sometimes s and t are both called the *end vertices* of P and $v_1, \ldots, v_{n-1}$ P's *internal vertices*. P is also referred to as an *$s-t$-walk*. n, the number of edges in P, is called the *length* of P. A walk of length 0 is called *trivial*, walks of length ≥ 1 are referred to as *nontrivial*. If the end vertices coincide, P is called a *closed walk*. We denote

$$\begin{aligned} V(P) &\coloneqq \{v_0, \ldots, v_n\}, \text{ and} \\ E(P) &\coloneqq \{e_1, \ldots, e_n\}, \end{aligned}$$

the families of vertices respectively edges occurring in P, counted by their multiplicities. P is said to *traverse* all of the vertices in $V(P)$, as well as all the edges in $E(P)$.

P is said to *connect* s and t. If $S, T \subseteq V(G)$ and $s \in S$, $t \in T$, then P is said to *connect* S and T or to be an *$S-T$-walk*. The notions *$s-T$-walk* and *$S-t$-walk* are likewise defined.

If all the internal vertices are distinct (and consequently all the e_i's as well), then P is called a *path*, or, in analogy to the above, an *$s-t$-*, respectively an *$S-T$-path*. A closed path is called a *cycle*. (Explicitly, this means that the first vertex is the same as the last, and all other vertices as well as the edges traversed are distinct.)

1.5 Example. Figure 1.4 on page 11 shows the graph G:

$$\begin{aligned} V(G) =& \{s, s', s'', u, v, w, x, t, t'\}, \\ A(G) =& \{\{s,u\}, \{s,s'\}, \{s',u\}, \{s',v\}, \{s'',v\}, \{s'',w\}, \\ & \{u,v\}, \{u,t\}, \{v,t'\}, \{v,x\}, \{w,t\}, \{t',x\}\}. \end{aligned}$$

The grey dashed line is meant to indicate the subsets $S \coloneqq \{s, s', s''\}$ and $T \coloneqq \{t, t'\}$ of $V(G)$. The thick black line with arrows next to it represents the following walk in G (the arrows indicating the direction in which the path is traversed):

$$P \coloneqq (s, \{s,u\}, u, \{u,v\}, v, \{v,s'\}, s', \{s',u\}, u, \{u,t\}, t).$$

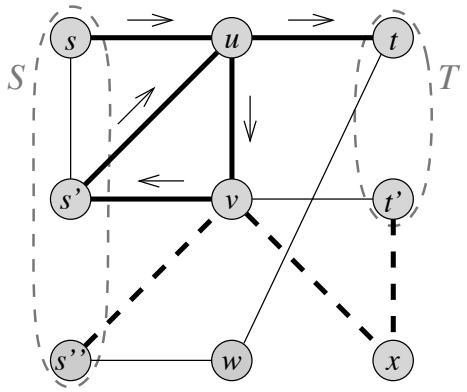

Figure 1.4: An $S-T$-walk that is no path (thick black line) and an $S-T$-path (dashed black line).

This is a walk in G, but not a path (u is traversed twice). P is an $s-t$-walk, as well as an $S-T$-walk, an $s-T$-walk, and an $S-t$-walk. We have

$$V(P) \coloneqq \{s, s', u, u, v, t\}, \text{ and}$$
$$E(P) \coloneqq \{\{s,u\},\{s',u\},\{s',v\},\{u,v\},\{u,t\}\}.$$

The length of P is 5. If we wanted, we could create an $S-T$-walk of greater length by going around the triangle in the middle twice:

$$P' \coloneqq (s,\{s,u\},u,\{u,v\},v,\{v,s'\},s',\{s',u\},\\ u,\{u,v\},v,\{v,s'\},s',\{s',u\},u,\{u,t\},t).$$

The edges highlighted by the thick dashed lines form a path of length 3:

$$R \coloneqq (s'',\{s'',v\},v,\{v,x\},x,\{x,t'\},t').$$

R is an $S-T$-path. One could as well define the following $T-S$-path:

$$Q \coloneqq (t',\{t',x\},x,\{x,v\},v,\{v,s''\},s'').$$

A closed walk is given by the concatenation of R and Q, i.e., first walk from s'' to t' along R, then from there back to s'' along Q. This does not result in a cycle. An example of a cycle is given by:

$$(s'',\{s'',w\},w,\{w,t\},t,\{t,u\},u,\{u,v\},v,\{v,s''\},s'').$$

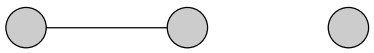

Figure 1.5: A simple example of a simple graph that is not connected.

Connectivity and Components

A graph G is called *connected* if there is a path connecting s and t, for all $s, t \in V(G)$. A maximal connected nonempty subgraph (maximal in the set of subgraphs, with respect to $\leq$) is called a *component* or a *connected component* of G. Every vertex and every edge of G belongs to exactly one component of G.

1.6 Example. Figure 1.5 shows a graph that is not connected: it is not hard to identify a pair of vertices that cannot be connected by a path. The graph has exactly two components. One contains two vertices and an edge, the other one just one vertex.

Matchings and Vertex Covers

Let G be a graph. A *matching in* G is a set of pairwise disjoint edges of G. I denote the set of all matchings of G by $\mathcal{M}(G)$. For a matching M, or in fact for any subset $M \subseteq E(G)$ of edges, it is handy to set

$$V(M) \coloneqq \bigcup_{\{u,v\} \in M} \{u, v\}.$$

This is the set of all vertices of G that are incident with some edge of M.

A *vertex cover of* G is a set of vertices of G that intersects each edge of G, i.e., a set $W \subseteq V(G)$ such that $e \cap W \neq \emptyset$ for all $e \in E(G)$, or in other words, all edges of G are incident with some $w \in W$. I denote the set of all vertex covers of G by $\mathcal{V}(G)$.

1.7 Example. In Figure 1.6 on page 13, the set of black-coloured vertices is a vertex cover of the graph depicted: all of the edges are incident with some black vertex. The set of all vertices is an example of a vertex cover containing more vertices. The two thick black lines form a matching: they are disjoint. The empty set is an example of a matching containing fewer edges.

Try to find a vertex cover containing fewer vertices and a matching containing more edges!

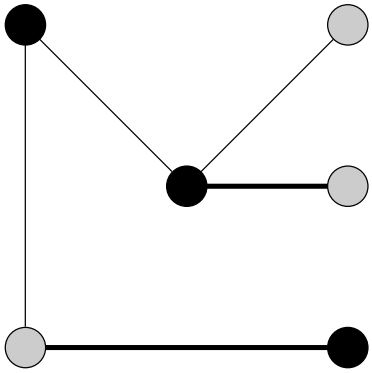

Figure 1.6: A vertex cover and a matching.

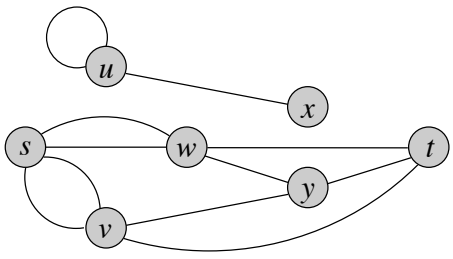

Figure 1.7: A typical graph.

1.8 Example. Let us now once again go through all the important notions introduced in this section, and give another example for each. In Figure 1.7 on page 13, some graph G is depicted. We have

$$\begin{aligned} V(G) &= \{s, t, u, v, w, x, y\}, \text{ and} \\ E(G) &= \{\{s, v\}, \{s, v\}, \{s, w\}, \{s, w\}, \{t, v\}, \\ &\qquad \{t, w\}, \{t, y\}, \{u, u\}, \{u, x\}, \{v, y\}, \{w, y\}\}. \end{aligned}$$

There are multiple edges: two pairs of two parallel edges each ($\{s, v\}$ and $\{s, w\}$). There is also one loop ($\{u, u\}$). Therefore, G is not a simple graph. The graph $H = (V(H), E(H))$ with

$$\begin{aligned} V(H) &\coloneqq \{t, w, y\} \\ E(H) &\coloneqq \{\{t, w\}, \{t, y\}, \{w, y\}\} \end{aligned}$$

is a subgraph of G. In fact, it is the subgraph of G induced by $\{t, w, y\}$. H is a simple graph, as is easily verified.

The sequence $(s, \{s, w\}, w, \{w, y\}, y, \{y, t\}, t, \{t, w\}, w)$ is a walk in G. More exactly, it is an $s - w$-walk, as well as an $s - V(H)$-walk. It is not a path in G, since w occurs in it more often than once.

G is not connected because there is no path connecting s and u. It has exactly two components, namely the subgraphs induced by $\{u, x\}$ and by $\{s, t, v, w, y\}$.

A matching in G is given by $\{\{u, x\}, \{w, y\}, \{t, v\}\}$. An example of a vertex cover of G is $\{s, t, u, y\}$.

1.3.2 Directed Graphs

The central object to be examined in this thesis are networks. Basically, these are directed graphs, only with some additional information and a few numbers assigned to them. This section includes the essential definitions, most of which are used many times in the remainder of the text. Nearly all of these definitions can be found in any standard introductory textbook on graph theory.

Digraphs

A *directed graph*, or simply *digraph*, is an ordered pair $D = (V, A)$, where V is some finite set, called *the set of vertices* or *the set of nodes* of D, and A is a family of ordered pairs (u, v) of elements of V, called *the family of arcs* of D. For a given digraph D, one sometimes writes $V(D)$ (respectively

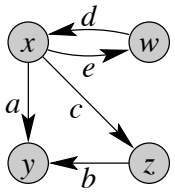

Figure 1.8: A simple digraph.

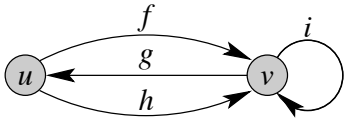

Figure 1.9: A non-simple digraph.

$A(D)$) for the set of vertices or nodes (respectively the family of arcs) of D. The elements of V (respectively A) are (as one could have guessed by now) called *vertices* or *nodes* (respectively *arcs*). An ordered pair occurring more than once in A is called a *multiple* arc. Two arcs which are represented by the same pair of vertices in the same order are called *parallel*. For an arc $a = (u, v)$, the ordered pair $a^{-1} := (v, u)$ is called the *inverse* of a. An arc of the form (v, v) (for some $v \in V$) is called a *loop*. A digraph with neither loops nor multiple arcs is called *simple*. A graph which is not simple is called *non-simple*.

1.9 Example. In Figures 1.8 and 1.9, two examples of digraphs are shown. The pictures should give an idea of how most people think of digraphs: as a number of little circles, plus some arrows, each pointing from one circle to another.

The digraph D, depicted in the first of the two graphics, consists of the following elements:

$$V(D) = \{w, x, y, z\}, \text{ and}$$
$$A(D) = \{a, b, c, d, e\}, \text{ with}$$
$$a = (x, y),\ b = (z, y),\ c = (x, z),\ d = (w, x), \text{ and } e = (x, w).$$

It is a simple digraph: there are no loops and no multiple arcs. In particular, d and e are not parallel. Rather, we have $(w, x) = (x, w)^{-1}$. In other words, d and e^{-1} are parallel, but as $e^{-1} \notin A(D)$, this does not affect the simpleness of D.

In the second of the two figures, the digraph E is shown. We have:

$$E = \big(\{u, v\}, \{f, g, h, i\}\big), \text{ with}$$
$$f = (u, v),\ g = (v, u),\ h = (u, v), \text{ and } i = (v, v).$$

E is not a simple digraph: i is a loop. Moreover, E contains the multiple arc (u, v): f and h are parallel.

Incidence and Neighbours

The arc $a = (u, v)$ is said to *connect* u *and* v, to *run from* u *to* v, to *leave* u, and to *enter* v. u and v are referred to as the *ends* of (u, v). More specifically, u is a's *head* or its *starting vertex* and v is a's *tail* or its *ending vertex*. If, on the other hand, u is not an end of a, then u and a are called *disjoint* from one another. Two arcs a and b are said to be *incident* with each other if they have a vertex in common and are called *disjoint* otherwise. If there is an arc connecting u and v, then they are called *adjacent* or *connected*. The vertex v is called an *outneighbour* of the vertex u if $(u, v) \in A$. It is called an *inneighbour* of u if $(u, v) \in A$. For $u \in V$, define

$$\begin{aligned}
\delta^{\mathrm{out}}_D(u) &:= \{(u, v) \in A \mid v \in V \backslash \{u\}\},\\
\delta^{\mathrm{in}}_D(u) &:= \{(v, u) \in A \mid v \in V \backslash \{u\}\},\\
N^{out}_D(u) &:= \{v \in V \backslash \{u\} \mid (u, v) \in A\}, \text{ and}\\
N^{in}_D(u) &:= \{v \in V \backslash \{u\} \mid (v, u) \in A\}.
\end{aligned}$$

These sets are (in this order): the family of arcs leaving u, or *outarcs of* u, the family of arcs entering u, or *inarcs of* u, the set of u's outneighbours, and the set of u's inneighbours. As usual, the subscript D is omitted when no confusion is to be expected. Similarily, set for $U \subseteq V$

$$\begin{aligned}
\delta^{\mathrm{out}}_D(U) &:= \{(u, v) \in A \mid u \in U, v \notin U\},\\
\delta^{\mathrm{in}}_D(U) &:= \{(v, u) \in A \mid v \notin U, u \in U\},\\
N^{out}_D(U) &:= \{v \in V \backslash U \mid \exists\, u \in U \colon (u, v) \in A\}, \text{ and}\\
N^{in}_D(U) &:= \{v \in V \backslash U \mid \exists\, u \in U \colon (v, u) \in A\}.
\end{aligned}$$

Let $S, T \subseteq V(D)$. An arc $a = (u, v)$ is said to *leave* S if $u \in S$ and $v \notin T$. It is said to *enter* T if $u \notin S$ and $v \in T$. If $u \in S$ and $v \in T$, then one says that a *connects* S and T.

1.10 Example. Let us look for examples of these ideas in Figures 1.8 and 1.9 on page 15. As before, I will refer to the first of these graphs as D and to the second as E.

In D, d's head is x. x is also the tail of a. c is incident with all other arcs. e and a are incident, whereas e and b are disjoint. x is adjacent to all other vertices. In fact, all other vertices are outneighbours of x, and w is an inneighbour as well. We have

$$\delta^{\mathrm{in}}_D(y) = \{a, b\} = \delta^{\mathrm{out}}_D(\{x, w, z\}).$$

If we set $U := \{x, y\}$ and $V := \{w, z\}$, then the arcs b, c, d, and e all are connecting U and V. More specifically, c and e are leaving U and entering V, whereas b and d are leaving V and entering U.

In E, due to the relatively small number of vertices, we find that every arc is incident with all other arcs as well as with all vertices. Moreover, all vertices are adjacent to all other vertices. We have $\delta_E^{\text{out}}(v) = \{g\}$. Note that loop i is not an out- nor an inarc for any vertex or any set of vertices. The same is true for any loop in any digraph.

Subgraphs

A *subgraph* E of a digraph D is a digraph with $V(E) \subseteq V(D)$ and $A(E) \subseteq A(D)$. This is sometimes denoted by $E \leq D$. A subgraph $E \leq D$ is called the *subgraph of D induced by W* if

$$\begin{aligned} &V(E) = W \text{ and} \\ &A(E) = \{(u, v) \in A(D) \mid u, v \in W\}. \end{aligned}$$

The set of subgraphs of a given digraph D is a partially ordered set with respect to $\leq$.

1.11 Example. The graph E depicted in Figure 1.9 on page 15 has exactly the following graphs as simple subgraphs:

$$\begin{aligned} &(\{u\}, \emptyset),\ (\{v\}, \emptyset),\ (\{u, v\}, \emptyset),\ (\{u, v\}, \{f\}),\ (\{u, v\}, \{g\}), \\ &(\{u, v\}, \{h\}),\ (\{u, v\}, \{f, g\}),\ \text{and}\ (\{u, v\}, \{g, h\}). \end{aligned}$$

The subgraph induced by $\{u\}$ is the first graph of this list, the one induced by $\{v\}$ is the second, and the subgraph of E induced by $\{u, v\}$ is E itself. The 'smallest' non-simple digraph (in the sense that is has not more vertices and not more arcs than any other non-simple digraph) is also a subgraph of E: $(\{v\}, i)$.

Directed and Undirected Paths

Let $D = (V, A)$ be a digraph. A *directed walk* or *diwalk*, sometimes simply *walk*, in D, is a sequence

$$P = (s = v_0, a_1, v_1, \ldots, a_n, v_n = t)$$

with $n \in \mathbb{N}$, $v_i \in V$ for $i = 0, \ldots, n$, and $a_i \in A$ an arc running from v_{i-1} to v_i (i.e., $a_i = (v_{i-1}, v_i)$), for $i = 1, \ldots, n$. s is called the *starting* or *first vertex* of P, t its *end* or *last vertex*. Sometimes s and t are both called the *end*

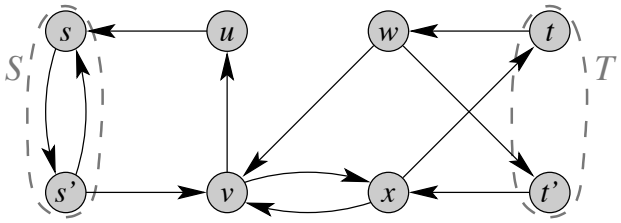

Figure 1.10: A digraph containing lots of paths.

vertices of P and $v_1, \ldots, v_{n-1}$ P's *internal vertices*. P is also referred to as a (*directed*) $s-t$-*walk*, or an $s-t$-*diwalk*. n is called the *length* of P. A walk of length 0 is called *trivial*, walks of length ≥ 1 are referred to as *nontrivial*. If the first and the last vertex coincide (i.e., $s = t$), the walk is called a *closed walk*. I denote by

$$V(P) \coloneqq \{v_0, \ldots, v_n\} \text{ respectively}$$
$$E(P) \coloneqq \{a_1, \ldots, a_k\}$$

the families of vertices respectively arcs occurring in P, counted by their multiplicities. P is said to *traverse* all of the vertices in $V(P)$, as well as all the arcs in $A(P)$.

Suppose S, $T \subseteq V$ and $s \in S$, $t \in T$. Then P is said to *connect* S and T, to *run from* s *to* t, or to be an $S-T$-*path*. If $S = \{s\}$ or $T = \{t\}$, the braces are usually left out.

If all the internal vertices are distinct, and consequently all the traversed arcs as well, then P is called a *directed path* or *dipath*, or sometimes simply a *path*. Similarly to the above, I will also speak of (*directed*) $S-T$-*paths* or $S-T$-*dipaths*. A closed path is called a *cycle*. (So in this case the first and last vertex coincide, whereas all other traversed vertices and all traversed arcs are distinct.) I denote the set of all paths of a given digraph D by $\mathcal{P}(D)$, and for sets S, $T \subseteq V$ the set of all $S-T$-paths by $\mathcal{P}_{S,T}(D)$. Here again, if $S = \{s\}$ or $T = \{t\}$, the braces are likely to be left out.

An *undirected walk* in $D = (V, A)$ is a directed walk in the directed graph $D' \coloneqq (V, A \cup A^{-1})$. Analogously defined are undirected paths, *closed undirected walks*, *undirected cycles*, *undirected* $S-T$-*walks*, and *undirected* $S-T$-*paths*.

1.12 Example. In the graph D, shown in Figure 1.10, there is a path from every vertex to every other vertex. In some cases, there are not many choices though, how to reach a vertex from another one, using paths only. Starting at s, there is for example only one path that ends in t'. This is expressed by

the following statement:

$$\mathcal{P}_{s,t'} = \{(s, (s, s'), s', (s', v), v, (v, x), x, (x, t), t, (t, w), w, (w, t'), t')\}.$$

Starting in u instead, we find that the situation is similar: there is only one $u - t'$-path P in D. It is obtained from the unique $s - t'$-path by adding $u, (u, s)$ to its left. P traverses all vertices of D, formally: $V(P) = V(D)$. It can be prolonged to a closed walk traversing all of D's vertices by going from t' back to x, then to v and finally to u again. After trying hard to find one for some time, we conclude that there is no closed *path* (i.e., no cycle) in D that contains all vertices.

The light grey dashes lines encircling s and s', respectively t and t', with the uppercase letters written next to them, shall indicate the two sets

$$S \coloneqq \{s, s'\} \text{ respectively } T \coloneqq \{t, t'\}.$$

It is not hard to convince oneself that there is also just one of each of the following: an $s - t$-path, an $s' - t$-path, and an $s' - t'$-path. Therefore, there are altogether 4 different $S - T$-paths. Of course, there are a lot more $S - T$-walks. In fact, there are infinitely many, because one could go round the graph in 'circles' over and over again.

There are two $x - v$-paths; one has length 1, the other one has length 3. The cycle $(u, (u, s), s, (s, s'), s', (s', v), v, (v, u), u)$ has length 4. Is there a cycle of greater length in D? The following is an undirected cycle:

$$(v, (v, x), x, (x, t'), t', (t, w), w, (w, v), v).$$

$(v, (v, x), x, (x, v), v)$ is another one—but not a very interesting one, because it is a directed cycle as well. Contrary to that, the undirected cycle $(x, (x, t'), t', (t', x), x)$ is not a directed cycle. Trying to find at least an *undirected* cycle that traverses all vertices of D, we have to conclude that not even that is possible.

Connectivity and Components

For digraphs, the notion of connectivity is a little more subtle than in the undirected case. There are two alternative concepts, corresponding to different 'degrees' of connectivity. A digraph D is called *connected* if for every pair $u, v \in V(D)$, there is an undirected path connecting u and v. A digraph D is called *strongly connected* if for every pair $u, v \in V(D)$, there is a dipath connecting u and v. Hence, every strongly connected digraph is connected, whereas the converse is not true.

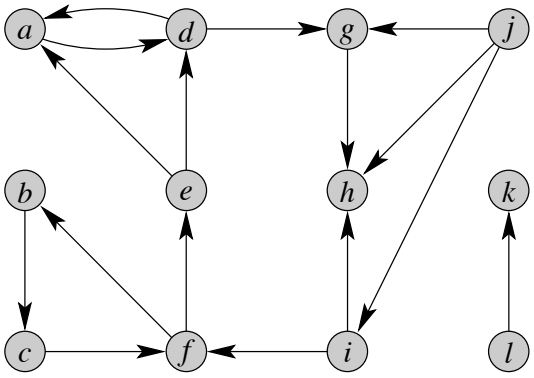

Figure 1.11: A digraph that is not weakly connected.

A maximal connected nonempty subgraph (maximal in the set of subgraphs, with respect to taking subgraphs) is called a *connected component* or a *weakly connected*, or *weak*, *component* of D. Every vertex and every arc of D belongs to exactly one weak component of D.

A maximal strongly connected nonempty subgraph (maximal in the set of subgraphs, with respect to $\leq$) is called a *strongly connected component* or a *strong component* of D. Every vertex of D belongs to exactly one strong component of D, but there may be arcs that belong to no strong component. In fact, it is not very hard to prove that $(u, v) \in A(D)$ belongs to some strong component of D if and only if there exists a $v - u$-dipath in D.

1.13 Example. Figure 1.11 illustrates the concepts of strong and weak connectivity in digraphs. Let us call the depicted graph D. It has 12 vertices and 16 arcs. D is a simple graph. It is not connected, since there is no $i - l$-path in D. Consequently, it is not strongly connected either.

There are exactly two weak components in D. One of them is the subgraph of D induced by $\{k, l\}$, the other component is the subgraph induced by $V(D)\backslash\{k, l\}$. Let them be called E, F respectively. Note that every arc in $A(D)$ belongs to exactly one of the sets $A(E)$ or $A(F)$, just as every vertex in $V(D)$ belongs to exactly one of the sets $V(E)$ or $V(F)$.

Looking for strong components, we notice that the situation is quite different. D contains exactly the following strong components (remember that strong components are subgraphs):

$$(\{a, d\}, \{(a, d), (d, a)\}),\ (\{b, c, f\}, \{(b, c), (c, f), (f, b)\}),$$
$$(\{e\}, \emptyset),\ (\{g\}, \emptyset),\ (\{h\}, \emptyset),\ (\{i\}, \emptyset),\ (\{j\}, \emptyset),\ (\{k\}, \emptyset),\ (\{l\}, \emptyset).$$

Again, every vertex belongs to exactly one strong component. But there are a lot of arcs that belong to no strong component (11 out of 16). Take

for example the arc (i, h). If a subgraph of D contains that arc, it has to contain the vertices h and i as well. But there is no dipath from h to i in D. Consequently, no subgraph of D can contain such a path. This means that no strongly connected subgraph can contain h and i at the same time, implying that no strongly connected subgraph (and hence no component of D) can contain the arc (i, h).

Let us now construct a new graph D' from D by reversing the arc (j, h). That is, delete arc (j, h) from D and add the new arc (h, j) instead. As far as weak connectivity is concerned, this does not change anything. But the new situation is quite different in terms of strong components: there are now only 3 of them (instead of 9 for D). They are: $(\{k\}, \emptyset)$, $(\{l\}, \emptyset)$, and the subgraph of D' induced by its remaining vertices. Equivalently, this last component is the subgraph of D' obtained from D' by deleting E (defined above). For example, the (unique) $a - c$-path in this component is:

$$(a, (a, d), d, (d, g), g, (g, h), h, (h, j), j, (j, i), i, (i, f), f, (f, b), b, (b, c), c).$$

The two existing $c - a$-paths are also paths in D, they do not use the new arc. They are:

$$(c, (c, f), f, (f, e), e, (e, a), a) \text{ and } (c, (c, f), f, (f, e), e, (e, d), d, (d, a), a).$$

Trees

Let $D = (V, A)$ be a digraph. A subgraph containing no nontrivial undirected cycles is called a *forest*. A connected forest is called a *tree*. A subgraph of a tree T that is a tree is called a *subtree* of T. A tree T containing all vertices of D, i.e., $V(T) = V$, is called a *spanning tree* of D. It is not difficult to show that the following proposition is correct:

1.14 Proposition. *Let $D = (V, A)$ be a digraph and $T \leq D$ a subgraph of D. Then the following statements are equivalent:*

1. *T is a spanning tree.*
2. *T is a forest and $|A(T)| = |V| - 1$.*

If D is connected, then it contains a spanning tree.

From the fact that there are no nontrivial undirected cycles in a forest, it can be deduced that there is at most one undirected path in a forest between every pair of vertices. In the case of trees, as they are connected, there exists exactly one undirected path between every pair of its vertices.

Let T be a spanning tree of the digraph $D = (V, A)$. We call T a *shortest path tree* if for every pair u, $v \in V$ of vertices of the digraph, the unique path from u to v in T is at the same time a shortest path from u to v in the graph.

A vertex of a tree that is adjacent to at most one other vertex is called a *leaf*. (The case that a leaf vertex has zero neighbours can only occur if it is the only vertex of the tree.) For some purposes a special vertex r, called *root*, is designated. In that case, one also speaks of a *tree rooted at* r.

Underlying Undirected Graphs

For every directed graph $D = (V, A)$, define the *underlying undirected graph* to be the undirected graph $G_D \coloneqq (V, E(A))$, where $E(A)$ is the family of unordered pairs of vertices of D obtained from A by deleting the orientaion of its elements:

$$E(A) \coloneqq \{\{u, v\} \mid (u, v) \in A\}.$$

For example, the graph in Figure 1.7 on page 13 is the underlying undirected graph of the graph in Figure 1.12 on page 24. Each walk (respectively path) $P = (v_0, a_1, v_1, a_2, \ldots, v_{k-1}, a_k, v_k)$ in D, directed or undirected (i.e., $a_i \in A \cup A^{-1}$, $i = 1, \ldots, k$ is allowed), gives rise in a natural manner to a walk (respectively path) in G_D, namely $R \coloneqq (v_0, e_1, v_1, e_2, \ldots, v_{k-1}, e_k, v_k)$ with $e_i \coloneqq \{v_{i-1}, v_i\}$.

One could have also defined the weak components and weak connectivity using underlying undirected graphs. The weak components of a digraph correspond exactly to the components of its underlying graph. In particular, a digraph D is weakly connected if and only if its underlying undirected graph is connected. A subgraph is a weak component of D if and only if its underlying undirected graph is a component of the underlying undirected graph of D. To verify this for some examples, have another look at Figure 1.11 on page 20, or at Figure 1.12 on page 24, discussed in Example 1.17.

Observe that walks in the underlying undirected graph are not quite the same as undirected walks in the digraph itself: suppose for example that the edge $\{v_{i-1}, v_i\}$ is traversed in a walk of the underlying undirected graph. Then, in a hypothetical equivalent undirected walk in the digraph itself, this could correspond to the traverse of (v_{i-1}, v_i) or to the traverse of the inverse of the conversely oriented arc $(v_i, v_{i-1})^{-1}$. Of course this ambivalence can only occur if we have $(v_{i-1}, v_i) \in A$ and $(v_i, v_{i-1}) \in A$ at the same time. Hence, the two notions coincide if and only if the implication

$$a \in A \Longrightarrow a^{-1} \notin A \tag{1.1}$$

holds true. This is the also the reason why the following definition makes

sense for paths in the underlying undirected graph of a digraph (in place of undirected paths in a digraph) only if Condition (1.1) is satisfied:

Characteristic Function of a Path

If P is a (possibly undirected) path in a digraph $D = (V, A)$, define the *characteristic funtion* $\chi^P \in \mathbb{R}^A$ of P through

$$\chi^P(a) \coloneqq \begin{cases} 1 & a \in A(P), \\ -1 & a^{-1} \in A(P), \\ 0 & \text{otherwise.} \end{cases}$$

Note that no ambiguity can arise in this definition: at most one of a, a^{-1} can be an element of $A(P)$, since P is a path (in which each vertex occurs at most once).

1.15 Example. Let us go back to the digraph D of Figure 1.11 on page 20 for a moment. I shall calculate the values of χ^P for some randomly chosen P. To illustrate the minor technical point mentioned above, that could cause confusion and needs a little care, call the arc $(a, d) =: x$ and the arc $(d, a) =: y$ for the moment. I define the following undirected path in the graph depicted:

$$P = (f, (f, e), e, (e, a), a, y^{-1}, d, (d, g), g, (g, j), j, (j, h), h, (h, i), i, (i, f), f).$$

Here are the values of the characteristic function of P:

$$\chi^P(\alpha) \coloneqq \begin{cases} 1 & \alpha \in \{(f, e), (e, a), (d, g), (j, h), (i, f)\}, \\ -1 & \alpha \in \{y, (j, g), (i, h)\}, \\ 0 & \alpha \in \{(f, b), (b, c), (c, f), (e, d), x, (g, h), (j, i), (l, k)\}. \end{cases}$$

Incidence Matrices

Let $D = (V, A)$ be a digraph. For subsets $U \subseteq V$ and $B \subseteq A$, define the *node-arc incidence matrix* (*of* U *and* B), alternatively called $U \times B$*-incidence matrix*, as the following $|U| \times |B|$ matrix Δ:

$$\Delta = (\delta_{u,b})_{(u,b) \in U \times B}, \quad \text{wherein}$$

$$\delta_{u,b} \coloneqq \begin{cases} 1 & \text{if } b \in \delta^{\text{out}}(u) \\ -1 & \text{if } b \in \delta^{\text{in}}(u) \\ 0 & \text{otherwise.} \end{cases}$$

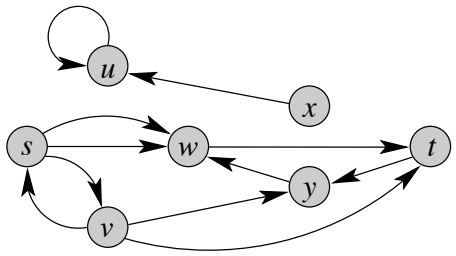

Figure 1.12: A typical digraph.

1.16 Example. Let us write down the node-arc incidence matrix for a particular digraph. This should give a feeling for how they typically look like: quite big, but with a lot of zero's as entries. The graph $D = (V, A)$, shown in Figure 1.12, has 7 vertices and 11 arcs. Therefore, its node-arc incidence matrix (so in the above definition, take $U = V$ and $B = A$) is a 7×11-matrix. In order to fit the whole matrix nicely on the page, I write down its transpose:

	s	t	u	v	w	x	y
(s, v)	1	0	0	−1	0	0	0
(s, w)	1	0	0	0	−1	0	0
(s, w)	1	0	0	0	−1	0	0
(t, y)	0	1	0	0	0	0	−1
(u, u)	0	0	0	0	0	0	0
(v, s)	−1	0	0	1	0	0	0
(v, t)	0	−1	0	1	0	0	0
(v, y)	0	0	0	1	0	0	−1
(w, t)	0	−1	0	0	1	0	0
(x, u)	0	0	−1	0	0	1	0
(y, w)	0	0	0	0	−1	0	1

Note that columns (or rows, in our case) corresponding to multiple arcs, like (s, w), are identical. Columns (rows) corresponding to loops, like (u, u) in our example, contain only zeros.

1.17 Example. Figure 1.12 illustrates once again various of the concepts introduced in this section. It shows the digraph D. D is a non-simple digraph: it has a multiple arc, namely (s, w), which appears twice in $A(D)$. Moreover, D contains the loop (u, u). D is not weakly connected, hence even less so strongly. There are two weak components. These are the subgraphs induced by $\{x, u\}$ and by $\{s, t, v, w, y\}$. The strong components are the subgraphs of D induced by the following sets of vertices: $\{x\}$, $\{u\}$, $\{s, v\}$, and $\{t, w, y\}$. The strong components all happen to be simple graphs.

What are the directed cycles in D? A cycle is always contained entirely in a single strong component. (This is because, wandering along the cycle, we can find a path from every vertex of the cycle to each of the other vertices of the same cycle.) For every vertex, there is exactly one cycle of length 0 (namely (z), for vertex z). Cycles of length 1 arise from loops; there is exactly one of those: $(u, (u, u), u)$. The cycles of length 2 occur, if there is a pair of 'anti-parallel' arcs; the only two examples in D are $(s, (s, v), v, (v, s), s)$ and $(v, (v, s), s, (s, v), v)$. Finally, there are three cycles of length 3: they can be found in the 'largest' strong component of D, the one containing the vertices t, w, and y. Starting from each of these vertices, there is exactly one cycle of length 3. As there is no strong component containing more than three vertices, we do not have to look for cycles of greater length; our list is complete already.

Set $b \coloneqq (s, v)$, $c \coloneqq (v, s)$. Furthermore, assign the labels d respectively e to the two parallel arcs (s, w). The following is an undirected cycle C (that is not directed) containing all vertices of the larger of the two weak components:

$$C \coloneqq (s, b, v, (v, t), t, (t, y), y, (y, w), w, d^{-1}, s).$$

It uses the ordered pair $d^{-1} = (w, s)$, which is not an arc of D. The characteristic function of this cycle is the following:

$$\chi^C(a) = \begin{cases} 1 & \text{if } a \in \{b, (v, t), (t, y), (y, w)\} \\ -1 & \text{if } a = d \\ 0 & \text{if } a \in \{c, (v, y), (w, t), e, (x, u), (u, u)\} \end{cases}.$$

Here is an $x - u$-walk that is not a path: $(x, (x, u), u, (u, u), u)$. There are four $s - t$-paths: $(s, d, w, (w, t), t)$, $(s, e, w, (w, t), t)$, $(s, (s, v), v, (v, t), t)$, and $(s, (s, v), v, (v, y), y, (y, w), w, (w, t), t)$.

Vertex s has two outneighbours (w and v) and one inneighbour (v). The vertex having the most outneighbours is v. It has three: s, t, and y. The vertex with the fewest outneighbours is u: it does not have any.

1.4 Preliminaries on Linear Programming

This section briefly introduces some of the basic concepts of linear programming. There are no proofs provided, and very little detail or broader contextual concepts. I just included the definitions and facts that are necessary to describe network flows as linear programs, and to understand how the general results from the theory of linear programming apply to flow optimization.

More comprehensive introductions to linear programming can for example be found in [5], [10], [3], [17], [14], [9], [16], [23], [6], [18], or [21].

1.4.1 Vectors and Matrices

Let $m, n \in \mathbb{N}$, and X be any set. Elements $x \in X^n$ are column vectors:

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}.$$

The set of $m \times n$ matrices with entries from some set X will be denoted by $M_{m,n}(X)$. If $m = n$, the notation $M_n(X)$ is also in common use. A matrix $M \in M_n(X)$ is called *square*.

The transpose of a vector $x \in X^n$ (respectively a matrix $M \in M_{m,n}(X)$) is written as x^{T} (respectively M^{T}). The *dimension of* $x \in X^n$ is defined to be the natural number

$$\dim(x) \coloneqq n.$$

Analogously, for a matrix $M \in M_{m,n}(X)$,

$$\dim(M) \coloneqq mn.$$

For $x, y \in \mathbb{R}^n$, $x = (x_i)_{i\in\{1,\dots,n\}}$, $y = (y_i)_{i\in\{1,\dots,n\}}$, the *scalar product of* x *and* y is the following sum:

$$x^{\mathrm{T}}y \coloneqq \sum_{i=1}^{n} x_i y_i.$$

A partial order on $\mathbb{R}$ can be defined through

$$x \leq y :\Longleftrightarrow \forall i \in \{1, \dots, n\} : x_i \leq y_i \,.$$

Although this use of the symbol $\leq$ is a little ambiguous, it will be utilized such that no confusion should arise. A norm on $\mathbb{R}^n$ that will be needed in one place is given by

$$\|x\|_1 \coloneqq \sum_{i=1}^{n} |x_i|.$$

This norm induces the following example of a metric on $\mathbb{R}^n$:

$$d_1(x, y) \coloneqq \|x - y\|_1.$$

Hence, $\mathbb{R}^n$ is a metric space. I will also need the following quite elementary existence result from analysis. It is cited from [20, p. 102], but can be found in any calculus textbook.

1.18 Theorem. *Let X be a nonempty, compact subset of a metric space. Let $f\colon X \to \mathbb{R}$ be a continuous function. Set*

$$M \coloneqq \sup_{v\in X} f(v), \qquad m \coloneqq \inf_{v\in X} f(v). \tag{1.2}$$

Then there exist v_0, u_0 with $f(v_0) = M$ and $f(u_0) = m$. In other words, the supremum in (1.2) is a maximum and the infimum is a minimum.

Let A, B be some finite sets. As the set X^A of all functions from A to X is isomorphic to $X^{|A|}$, I use both notions without clear distinction. Similarly, for matrices I use notations like $M_{A,B}(X)$ instead of $M_{|A|,|B|}(X)$. For a set X and $x \in X^n$, $y \in X^m$, the vector $z = \begin{pmatrix} x \\ y \end{pmatrix}$ is the element of X^{n+m} defined as

$$z = (z_i)_{i\in\{1,\dots,n+m\}}, \quad z_i \coloneqq \begin{cases} x_i & \text{if } 1 \le i \le n \\ y_{i-n} & \text{if } n+1 \le i \le n+m \end{cases}.$$

I will also use similar notations for matrices consisting of smaller matrices. If a vector x is indexed by some set J, i.e., $x = (x_j)_{j\in J} \in X^J$, and some subset $I \subseteq J$ is given, then $x_I \coloneqq (x_j)_{j\in I}$. A square matrix $M = (m_{i,j})$ is called *lower triangular* (respectively *upper triangular*) if it contains only nonzero entries from the main diagonal downwards (respectively upwards). In other words, if it satisfies $m_{i,j} \neq 0 \Rightarrow i \ge j$ (respectively $i \le j$). Finally, define I_A to be the identity matrix on the set A, i.e.,

$$I_A \coloneqq (i_{a,b})_{(a,b)\in A\times A} \quad \text{with} \quad i_{a,b} \coloneqq \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}.$$

1.4.2 Linear Programs

Linear programs constitute a most powerful tool in applied mathematics. In the form we know them today, they were developed in the first half of the last century in military context. Since then, linear programs have been applied successfully in a great variety of different settings. They can be found whenever a large number of interacting entities is to be modelled mathematically. The linearity assumption in many cases provides a sufficiently accurate approximation, while at the same time it guarantees solvability and efficient computability.

An excellent introduction (not entirely modern though) including a detailed overview of the prehistory and history of linear programming up to the early 1960s is [5]. This book can be regarded as the staring point of linear

programming in today's form. Its author, George B. Dantzig, is commonly perceived as the 'father' of linear programming.

Basically, a linear program consists of a set of variables, together with a set of linear constraints imposed on them, and a linear objective function that is to be optimized (while respecting all constraints). The following is a standard formulation, as found in many textbooks:

1.19 Definition. Let $m, n \in \mathbb{N}$. A set $P \subseteq \mathbb{R}^n$ is called a *polyhedron* if there exists an $m \times n$ matrix A with real entries and a vector $b \in \overline{\mathbb{R}}^m$ such that

$$P = \{x \in \mathbb{R}^n \mid Ax \leq b\}. \tag{1.3}$$

1.20 Definition. A *linear program* is the task of maximizing (or minimizing) a given linear function over a polyhedron. In other words: Let $m, n \in \mathbb{N}$, $A \in M_{m,n}(\mathbb{R})$, $b \in \overline{\mathbb{R}}^m$, $c \in \overline{\mathbb{R}}^n$, and $P \coloneqq \{x \in \mathbb{R}^n \mid Ax \leq b\}$. The following problem is a *linear program*:

$$\begin{aligned} &\text{Find } x \in \mathbb{R}^n \text{ such that} \\ &\text{(i)} \quad x \in P \\ &\text{(ii)} \quad c^{\mathrm{T}}x \text{ is maximal.} \end{aligned}$$

Or, shorter:

$$\max\{c^{\mathrm{T}}x \mid Ax \leq b\} = \max\{c^{\mathrm{T}}x \mid x \in P\}. \tag{1.4}$$

The function $x \mapsto c^{\mathrm{T}}x$ is called *objective function* or *cost function*[5]. c is sometimes called *cost vector*. If $P \neq \emptyset$, the program is called *feasible*. If $x \in P$ (not necessarily optimal), x is called *feasible*.

As this is not exactly the formulation suited best for the network model I am going to introduce, I give another, fairly general, definition of linear programs. It is taken from [21].

1.21 Definition. Let A, B, C, D, E, F, G, H, K be (real) matrices, and let α, β, γ, δ, ε, ζ be (real) vectors. A *linear program* is the following problem (assuming that the dimensions of all vectors and matrices are compatible):

$$\begin{aligned} \max\{\delta^{\mathrm{T}}x + \varepsilon^{\mathrm{T}}y + \zeta^{\mathrm{T}}z \mid\ & x \geq \mathbf{0},\ z \leq \mathbf{0}, \\ & Ax + By + Cz \leq \alpha, \\ & Dx + Ey + Fz = \beta, \\ & Gx + Hy + Kz \geq \gamma\}. \end{aligned} \tag{1.5}$$

[5]The word 'cost' stems from the analog minimization problem.

Any linear program can be written in many different forms. There are several standard forms found frequently in linear programming literature. Quite often, they are stated in terms of minimizing the objective function, instead of maximizing it. The linear program given in (1.5) can be converted to such a formulation by exchanging δ, ε, and ζ by $-\delta$, $-\varepsilon$, and $-\zeta$. Here are some examples of common definitions of linear programs, and how they can be derived from (1.5):

general form : $\zeta = \mathbf{0}$, $A = B = C = F = K = \mathbf{0}$.

canonical form : $\varepsilon = \zeta = \mathbf{0}$, $A = B = C = D = E = F = H = K = \mathbf{0}$.

standard form : $\varepsilon = \zeta = \mathbf{0}$, $A = B = C = E = F = G = H = K = \mathbf{0}$.

It is not hard to show that all of these versions of linear programs are equivalent. Every formulation can be converted into each of the alternative formulations by some notational changes. In Chapter 3, I am going to use formulation (1.5) with $\delta = \zeta = \mathbf{0}$, $A = C = D = F = G = K = \mathbf{0}$, and $B = H = I$ the identity matrix. The reason for using just that formulation simply is that the constraints needed come up naturally in that way. I would see no advantage in transferring them artificially to a different, more standardized, formulation. In fact, that would make the linear programs considered more complicated and would blur the underlying ideas.

1.4.3 The Simplex Method

The Simplex Method was the first algorithm to solve the general linear program. It was conceived by Dantzig. As mentioned earlier, he describes it extensively in his classical monograph [5]. Other detailed expositions include [16], [10], [18], and [17]. It works very efficiently in practice, and it is the most widespread used algorithm for solving linear programs.

I shall describe the Simplex Algorithm for linear programs in the exact form that will naturally arise from the graph theoretical discussion in Chapter 2. This section does not provide detailed information on the various steps performed during the Simplex Algorithm, in a way that it could for example be implemented efficiently without further reference. Mention of the numerous variants of the algorithm, most of which being suited better for some situations and not as good for other ones, is rather rudimentary. This short introduction is meant to give more of an overview of the basic ideas utilized by the algorithm. It is mainly based on [1] and [16].

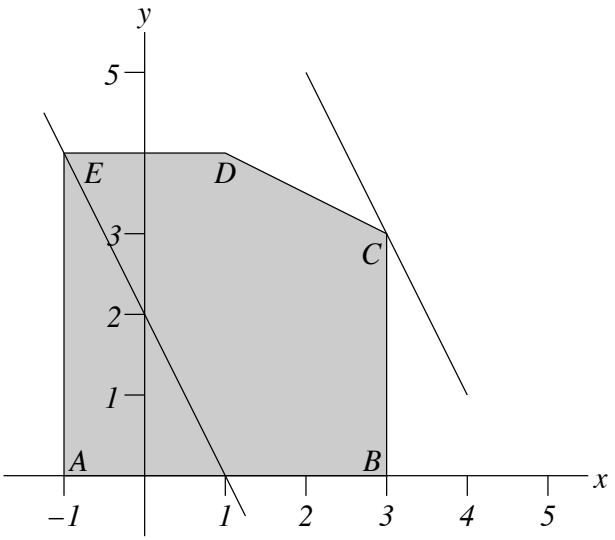

Figure 1.13: Feasible region of a linear program.

A Graphical Solution - Extreme Points

If a linear program involves only one, two or three variables, then there exists an illuminating graphical representation and solution method that will help to understand the main ideas behind the Simplex Algorithm. I want to start this brief résumé by an example illustrating this graphical approach:

1.22 Example. Consider the following linear program:

$$\begin{aligned} \text{Maximize} \quad & u(x,y) = 2x + \ y \\ \text{subject to} \quad & x + 2y \leq 9 \\ & -1 \leq x \leq 3, \quad 0 \leq y \leq 4 \end{aligned}$$

The grey shaded region of Figure 1.13 is the set of feasible solutions for this program. Besides, it is an example of a polyhedron.

The vertices $A = (-1, 0)$, $B = (3, 0)$, $C = (3, 3)$, $D = (1, 4)$, and $E = (-1, 4)$ are the *extreme points* of this polyhedron. These are the points formed by the intersection of the lines corresponding to the various constraints of the linear program. Furthermore, they are not a strict convex combination of any two distinct points of the polyhedron. More generally, it can be shown that an element of a polyhedron is an extreme point of it if and only if it is not the strict convex combination of two distinct points of the polyhedron.

The linear program asks for a point (x, y) of the polyhedron $ABCDEA$ that maximizes $u(x, y) = 2x + y$. Equivalently, one can ask for the largest

value of U, such that the line $2x + y = U$ has a point in common with the polyhedron. This last problem can be solved by successively drawing parallel lines, moving as far to the right as possible. In this specific case, this can be done until the line $2x + y = U$ intersects the polyhedron only in the extreme point $C = (3, 3)$, where the maximum value $U = 9$ of the objective function is obtained. Moving the line even further to the right, it would end up disjoint from the polyhedron, such that no feasible solutions were lying on the line any more.

The most important property of linear programs that should be illustrated by this example is, that if an optimal solution exists, then they always have optimal solutions that are extreme points of the underlying polyhedron.[6] Hence, when looking for an optimal solution, one needs only consider a finite candidate set of points (the set of extreme points is always finite, if the number of constraints is finite).

The Simplex Algorithm starts at some feasible extreme point and moves to an adjacent extreme point in each iteration, while improving the values of the objective function in every step (or sometimes not altering them). This continues until an optimal extreme point solution is reached. Which extreme point is chosen as the next one to be visited is dependent on certain rules known as *pivot rules*. They exist in many variants, each leading to a different version of the Simplex Algorithm. For example, if in the above polyhedron the Simplex Algorithm would be initiated with vertex A as a starting point, it might first visit the points E, then D, before finally finding the optimum solution C. Alternatively, it might first consider the point B, then C.

Basic (Feasible) Solutions and Canonical Form

Let the following linear program be given:

$$\max\{u^{\mathrm{T}} f \mid d \leq f \leq c,\ \Delta f = b\}. \tag{1.6}$$

Herein, u, f, c, and d are n-dimensional vectors, b is an m-dimensional vector, and Δ is an $m \times n$-matrix. The equation $\Delta f = b$ can be viewed as m constraints imposed on the n variables $(f_a)_{a \in A}$. Let A be a set with n elements

[6] Not all optimal solutions need to be extreme points. For example, changing the objective function to $u'(x, y) = x + 2y$, all the points of the line segment CD end up being optimal solutions.

and let V be one with m elements. Then I henceforth write

$$\begin{aligned} b &= (b_v)_{v\in V}, \\ c &= (c_a)_{a\in A}, \\ d &= (d_a)_{a\in A}, \\ f &= (f_a)_{a\in A}, \\ u &= (u_a)_{a\in A}, \text{ and} \\ \Delta &= (\delta_{v,a})_{v\in V,\, a\in A}. \end{aligned}$$

As mentioned earlier, a vector f with $d \le f \le c$ satisfying $\Delta f = b$ is called *feasible*, or a *feasible solution* of (1.6). If the number of linearly independent constraints $\sum_{a\in A} \delta_{v,a} f_a = b_v$ is greater than the number n of variables, then (1.6) is infeasible, i.e., no feasible solutions exist. As linearly dependent constraints can be omitted without altering the set of feasible solutions, let us henceforth assume that $n \ge m$. For $a \in A$, I will also refer to $[d_a, c_a]$ as the *feasible* or the *feasibility interval* of f_a, or as the *capacity interval* corresponding to index a.

I now want to explain the so-called *basic solutions* of (1.6). Basic solutions actually need not really be solutions, as they need not be feasible. If they are feasible, they are called *basic feasible solutions*, and these latter ones are of central importance for the Simplex Algorithm. To facilitate understanding, I first give an example.

1.23 Example. Have a look at the following linear program in the form of (1.6):

$$\begin{aligned} \text{Maximize} \quad & u(f) = 3f_1 + \ f_2 + 4f_3 - 6f_4 && (1.7)\\ \text{subject to} \quad & f_1 + \ f_2 - 3f_3 + 4f_4 = 4 && (1.8)\\ & f_1 + 2f_2 - 4f_3 + 2f_4 = -2 && (1.9)\\ & 0 \le f_1, f_3 \le 10, \quad -2 \le f_2, f_4 \le 5 && (1.10) \end{aligned}$$

A well-known way of finding a solution is to use *elementary row operations*. The general Simplex Algorithm also employs them. They consist of the following actions in arbitrary sequence:

1. Multiplying a row (i.e., a constraint) by a constant, or
2. Adding one row to another or to the objective function.

We could for example multiply (1.8) by -3 and add the result to the objective function. If then (1.9) is multiplied by $-\frac{1}{2}$ and added to (1.8), the result is

as follows:

$$\begin{aligned}
\text{Maximize} \quad & u(f) = 0f_1 - 2f_2 + 13f_3 - 18f_4 + 12 && (1.11)\\
\text{subject to} \quad & \frac{1}{2}f_1 \quad - \quad f_3 + \ 3f_4 = 5 && (1.12)\\
& f_1 + 2f_2 - \ 4f_3 + \ 2f_4 = -2 && (1.13)\\
& 0 \leq f_1, f_3 \leq 10, \quad -2 \leq f_2, f_4 \leq 5 && (1.14)
\end{aligned}$$

Now multiplying (1.12) by -2 and adding the result to (1.13) transforms our linear program into the following form:

$$\begin{aligned}
\text{Maximize} \quad & u(f) = 0f_1 - 2f_2 + 13f_3 - 18f_4 + 12 && (1.15)\\
\text{subject to} \quad & \frac{1}{2}f_1 \quad - \quad f_3 + \ 3f_4 = 5 && (1.16)\\
& 2f_2 - \ 2f_3 - \ 4f_4 = -12 && (1.17)\\
& 0 \leq f_1, f_3 \leq 10, \quad -2 \leq f_2, f_4 \leq 5 && (1.18)
\end{aligned}$$

Finally, we add (1.17) to (1.15), then multiply (1.16) by 2 and (1.17) by $\frac{1}{2}$. This yields the following linear program:

$$\begin{aligned}
\text{Maximize} \quad & u(f) = 0f_1 - 0f_2 + 11f_3 - 22f_4 + 24 && (1.19)\\
\text{subject to} \quad & f_1 \quad - \ 2f_3 + \ 6f_4 = 10 && (1.20)\\
& f_2 - \ f_3 - \ 2f_4 = -6 && (1.21)\\
& 0 \leq f_1, f_3 \leq 10, \quad -2 \leq f_2, f_4 \leq 5 && (1.22)
\end{aligned}$$

Note that (1.7)–(1.10) and (1.19)–(1.22) are equivalent linear programs, i.e., they have the same set of feasible solutions, and the optimum values of their respective objective functions coincide.

In (1.19)–(1.22), the variables f_1 and f_2 are said to be *isolated*. That is, they appear in exactly one constraint equation, with coefficient 1, and do neither appear in any other equation nor in the objective function. If exactly one variable is isolated in each constraint and if all of these variables have coefficient zero in the objective function, then the linear program is said to have *canonical form*. Under the initially made assumption that $n \geq m$, it is always possible to transform a linear program into canonical form using elementary row operations.

The reason why it is useful to transform a linear program into canonical form is that one can calculate a certain type of solutions from it quite easily: in (1.19)–(1.22), arbitrarily set the non-isolated variables f_3 and f_4 to its respective upper or lower bounds, e.g., $f_3 = 0$, f_4=5. Then to satisfy (1.20), we must have $f_1 = -20$. Similarly, to satisfy (1.21), we necessarily have

$f_2 = 4$. These values for (f_1, f_2, f_3, f_4) are an example of a *basic solution*. In this case, f_2 satisfies its required bounds (1.22), whereas f_1 does not. Therefore this basic solution (f_1, f_2, f_3, f_4) is not a feasible one. If it was, it would be called a *basic feasible solution*.

Interestingly, it can be shown that the basic feasible solutions of a linear program are exactly the extreme points (the vertices) of the polyhedron defined by the constraints of the linear program. The Simplex Algorithm moves from one basic feasible solution to another, until an optimal solution is reached.

In general, a basic solution is obtained as follows. The set of variables is partitioned into three subsets: the set of variables that are to be isolated (of which there is one per linearly independent constraint), henceforth referred to as *basic variables*, the set of non-basic variables that are set to their upper bounds, and the set of non-basic variables that are set to their lower bounds. I denote the respective index sets of the variables by B, C and D. Hence, for the index set A of all variables, $A = B \cup C \cup D$. Such a triplet (B, C, D) is called a *basis structure*. For every basis structure, some further notational conventions will prove useful: let the column of the constraint matrix Δ corresponding to index $a \in A$ be denoted by Δ_a, i.e., $\Delta = (\Delta_a)_{a \in A}$. Then the following matrices are needed in the sequel:

$$\Delta_B \coloneqq (\Delta_a)_{a \in B}, \quad \Delta_C \coloneqq (\Delta_a)_{a \in C}, \quad \Delta_D \coloneqq (\Delta_a)_{a \in D}.$$

Here the matrix Δ_B, corresponding to the basic variables, is an $m \times m$ square matrix. Likewise, the variables $f = (f_a)_{a \in A}$ are partitioned into three sets as follows:

$$f_B \coloneqq (f_a)_{a \in B}, \quad f_C \coloneqq (f_a)_{a \in C}, \quad f_D \coloneqq (f_a)_{a \in D}.$$

With these definitions, the constraint equation $\Delta f = b$ can be reformulated in the following way:

$$\Delta_B f_B + \Delta_C f_C + \Delta_D f_D = b. \tag{1.23}$$

If the columns of Δ corresponding to the m basic variables f_B are linearly independent, then these variables are called a *basis*, and the matrix Δ_B is called a *basis matrix*. Henceforth, it will always be assumed implicitly that the basic variables meet this requirement. Then Δ_B can be inverted and the following solution can be inferred from (1.23):

$$f_B = \Delta_B^{-1} b - \Delta_B^{-1} \Delta_C f_C - \Delta_B^{-1} \Delta_D f_D, \quad f_C = c_C, \quad f_D = d_D. \tag{1.24}$$

For some choices of (B, C, D) this basic solution will be feasible, which is the case if and only if $d_B \leq f_B \leq c_B$, and for other choices it will not. Accordingly, one speaks of *feasible* and *infeasible basis structures*.

What is left in order to transform the linear program into canonical form is to make sure that all basic variables have a zero coefficient in the objective function. This is the topic of the next section.

Reduced Costs

For a given basis structure (B, C, D), transforming the objective function of a linear program into canonical form can be achieved by performing a sequence of elementary row operations. This is equivalent to multiplying each constraint $\sum_{a\in A} \delta_{v,a} f_a - b_v = 0$ by a constant π_v, and subtracting the result from the cost function $\sum_{a\in A} u_a f_a$. This results in the following updated cost function:

$$\sum_{a\in A} u_a f_a - \sum_{v\in V} \pi_v \big(\sum_{a\in A} \delta_{v,a} f_a - b_v \big).$$

Collecting terms in this expression, and setting $u_0 \coloneqq \sum_{v\in V} \pi_v b_v$, it can be rewritten as

$$U(f) \coloneqq \sum_{a\in B} \big(u_a - \sum_{v\in V} \pi_v \delta_{v,a}\big) f_a + \sum_{a\in C\cup D} \big(u_a - \sum_{v\in V} \pi_v \delta_{v,a}\big) f_a + u_0. \quad (1.25)$$

To obtain an objective function in canonical form, one has to choose the vector $\pi = (\pi_v)_{v\in V}$ such that

$$u_a = \sum_{v\in V} \pi_v \delta_{v,a} \text{ for all } a \in B.$$

Using matrices, this can be written as

$$\pi^{\mathrm{T}} \Delta_B = u_B^{\mathrm{T}} \quad \text{or, equivalently,} \quad \pi^{\mathrm{T}} = u_B^{\mathrm{T}} \Delta_B^{-1}.$$

The numbers $(\pi_v)_{v\in V} = \pi \coloneqq (\Delta_B^{\mathrm{T}})^{-1} u_B$ are called the *simplex multipliers* associated with the basis B. For every $a \in A$, $u_a^\pi \coloneqq u_a - \sum_{v\in V} \pi_v \delta_{v,a}$ is referred to as the *reduced cost* of the variable f_a with respect to the basis B. Consequently, all the basic variables have reduced cost 0. Note that the value of the objective function in canonical form corresponding to the basis structure (B, C, D) can now be calculated, using (1.24), as follows (with

non-basic variables set to their upper or lower bounds):

$$\begin{aligned}
U(f) &= \sum_{a\in C}\big(u_a - \sum_{v\in V}\pi_v\delta_{v,a}\big)c_a + \sum_{a\in D}\big(u_a - \sum_{v\in V}\pi_v\delta_{v,a}\big)d_a + u_0 \\
&= \sum_{a\in C}\big(u_a - \pi\Delta_a\big)c_a + \sum_{a\in D}\big(u_a - \pi\Delta_a\big)d_a + \pi b \\
&= u_C^{\mathrm{T}}c_C - \pi^{\mathrm{T}}\Delta_C c_C + u_D^{\mathrm{T}}d_D - \pi^{\mathrm{T}}\Delta_D d_D + \pi^{\mathrm{T}}b \\
&= u_C^{\mathrm{T}}c_C - u_B^{\mathrm{T}}\Delta_B^{-1}\Delta_C c_C + u_D^{\mathrm{T}}d_D - u_B^{\mathrm{T}}\Delta_B^{-1}\Delta_D d_D + u_B^{\mathrm{T}}\Delta_B^{-1}b \\
&= u_B^{\mathrm{T}}f_B + u_C^{\mathrm{T}}c_C + u_D^{\mathrm{T}}d_D = u^{\mathrm{T}}f.
\end{aligned}$$

This verifies that for the basic solution f, the modified cost function (1.25) assumes the same value as the original cost function.

Description of the Simplex Algorithm

Simplex Algorithm is actually the name of a family of numerous closely related algorithms. They all work as follows: given an initial basic feasible solution, it maintains such a solution at every step. First, a certain optimality criterion is applied to the solution. If the solution is optimal, the algorithm (naturally) stops. Otherwise, an operation called *pivot operation* is performed to obtain another basis structure with corresponding equal or greater value of the objective function. This is done by first selecting one non-basic variable, called *entering variable*, which is to become a basic variable in the next step. Then one of the basic variables, called *leaving variable*, is selected to become a non-basic variable. Afterwards, the linear program is updated in order to obtain canonical form with respect to the new basis structure. This whole procedure is repeated until the optimality criterion is satisfied. As mentioned before, which basis structure (which entering and leaving variable) is chosen in each step depends on the so-called *pivot rules*. Different sets of pivot rules lead to differences in the performance of the Simplex Algorithm. For almost all versions of the Simplex Algorithm, there are counterexamples, showing that the running time is not necessarily polynomial. However, despite this exponential worst-case time complexity, the Simplex Method works very well and fast empirically, and for problems of seemingly arbitrary size.

I summarize all of these basic operations of the Simplex Algorithm in the following table:

Simplex Algorithm

Input
- Vectors b, c, d, u and a matrix Δ of the linear program $\max\{u^{\mathrm{T}} f \mid d \leq f \leq c,\ \Delta f = b\}$
- An initial feasible basis structure (B, C, D)

Output A feasible solution $f_{\max}$ of maximal value

① `Compute the basic solution` f `and the reduced costs` $(u_a^\pi)_{a \in A}$ `corresponding to the current basis structure.`

② `Look for non-basic variables violating their optimality criterion.`

- `If none exists, stop.` $f_{\max} \coloneqq f$ `is maximal.`
- `Otherwise, select one of them as the entering variable and go to ③.`

③ `Select a leaving variable.`

④ `Update` (B, C, D) `and go to ①.`

One problem that could possibly arise is that if the value of the objective function does not increase during a couple of consecutive steps, the algorithm might get caught in an infinite cycle, in which the same sequence of solutions is considered repeatedly. In order to prevent this, a couple of *anti-cycling rules* have been conceived, that constitute adaptions to the various pivot rules.

Now, I would like to have a closer look at the various steps that are performed during an execution of the Simplex Algorithm.

Finding an Initial Feasible Solution

As a basic solution need not be feasible, the first important topic is how to find an initial feasible solution. Unfortunately, there is no easy way for determining a feasible basis structure. In fact, finding one is almost as difficult as finding an optimal solution, given some feasible solution is already known.

Nevertheless, there exists a simple technique of finding an optimal solution to an augmented linear program for which an initial basic feasible solution is known and whose optimal solutions coincide with the ones of the original program (in case this latter one has any feasible solutions at all). The

method consists of introducing a set of artificial new variables $g = (g_v)_{v \in V}$ with sufficiently large negative costs $M = (M_v)_{v \in V}$, one for every constraint equation of the original linear program. This augmented program is the following linear program:

$$\max\{u^{\mathrm{T}} f + M^{\mathrm{T}} g \mid d \leq f \leq c,\ 0 \leq g < \infty,\ \Delta f + \varepsilon g = b\}. \tag{1.26}$$

Here $\varepsilon = (\varepsilon_{u,v})_{u,v \in V}$ is a diagonal matrix, with the only nonzero elements being diagonal entries equalling 1 or -1. The v-th entry $\varepsilon_{v,v}$ is chosen according to whether the v-th row of $b - \Delta d$ is positive or negative. The purpose of this is that g can then be restricted to nonnegative values, as is done in (1.26), without sacrificing the easy solvability of the program. The system of constraint equations in (1.26) is still in the form used in (1.6), as by setting $\Delta' \coloneqq (\Delta, \varepsilon)$ and $f' \coloneqq (f, g)$, it can be written as $\Delta' f' = b$. Analogously, the objective function can be dealt with.

It is not difficult to show that the original program has a feasible solution if and only if the auxiliary program has an optimal solution for which $g = \mathbf{0}_V$. This is because the large negative cost coefficients of the variables g_v prevent them from being nonzero in an optimal solution if this is not necessary for obtaining any solution at all. The solutions of the auxiliary program in which the artificial variables are all zero are also solutions to the original program. Therefore, solving the auxiliary program also solves the original program.

The point herein is that an initial basis structure and a basic feasible solution for the augmented program is obtained very easily: the basis is the set of artificial variables, all non-artificial variables are set to their lower bounds. The values of the artificial variables can be calculated according to the equation $\varepsilon g = b - \Delta d$. Due to the choice of $\varepsilon_{v,v}$, g_v is guaranteed to be ≥ 0, i.e., to be feasible.

An Optimality Criterion

The value of the objective function in the canonical form of a linear program, with respect to a feasible basis structure (B, C, D), is the following:

$$U(f) = \sum_{a \in C} u_a^{\pi} c_a + \sum_{a \in D} u_a^{\pi} d_a + u_0 \tag{1.27}$$

Here the coefficients $u_a^{\pi} = u_a - \sum_{v \in V} \pi_v \delta_{v,a}$ are the reduced costs of the non-basic variables f_a and $u_0 = \sum_{v \in V} \pi_v b_v$. The numbers π_v are the simplex multipliers associated with B. Since for all $a \in A$ we have $f_a \leq c_a$ and $f_a \geq d_a$, the value of (1.27) is an upper bound on the value of the objective function if

1. $u_a^\pi \geq 0$ for all $a \in C$, and

2. $u_a^\pi \leq 0$ for all $a \in D$.

If 1. and 2. are satisfied, then a decrease of the value of some f_a for $a \in C$ would result in a decrease of the value of the objective function. The same is true for a potential increase of the value of f_a for some $a \in D$. The condition $u_a^\pi \geq 0$, respectively $u_a^\pi \leq 0$, is therefore called *optimality condition* for the variable $a \in C$, respectively $a \in D$.

Consequently, as for the current basic feasible solution f the objective function U assumes the value $U(f)$ given in (1.27), if Conditions 1. and 2. hold, this solution must be optimal.

Entering Variable Criterion

Any variable that violates its optimality condition is eligible as an entering variable. That is, the entering variable f_e has to satisfy one of

1. $e \in C$ and $u_e^\pi < 0$, or

2. $e \in D$ and $u_e^\pi > 0$.

Obviously, there is more than one way to choose an entering variable among possible candidates. Equation (1.27) implies that, in the case that 1. respectively 2. holds for some index a, $|u_a^\pi|$ is the rate of increase of the objective function $U(f)$ per unit decrease of the variable f_a (for $a \in C$), or per unit increase of the variable f_a (for $a \in D$). If f_a is chosen as the entering variable, in the next step the algorithm will try to decrease (for $a \in C$) or increase (for $a \in D$) the value of f_a as much as possible. Hence, it is a good idea to choose a variable f_a whose reduced cost currently has maximum absolute value $|u_a^\pi|$ as the next entering variable f_e. This criterion is often referred to as *Dantzig's pivot rule*, as it was first suggested by George B. Dantzig. The drawback of this method are high computational costs, since a lot of variables have to be tested in every step. Another common rule for choosing an entering variable is the following: a list of all variables violating their optimality condition is maintained and, quite simple, the first variable in that list is selected. Compared to Dantzig's rule, this is the opposite extreme: there is very little computation necessary to find a suitable variable, but it is chosen with little carefulness, such that the number of iterations the algorithm needs for finding an optimal solution might increase. There are also implementations that are sort of a mixture and use elements of both of these approaches.

Leaving Variable Criterion

Suppose the current basis structure for the linear program

$$\max\{u^{\mathrm{T}} f \mid d \leq f \leq c,\ \Delta f = b\}$$

is (B, C, D), and the non-basic variable f_e has been chosen as the next entering variable. Recall that

$$f_B = \Delta_B^{-1} b - \Delta_B^{-1} \Delta_C f_C - \Delta_B^{-1} \Delta_D f_D. \tag{1.28}$$

Here $\Delta = (\Delta_a)_{a \in A}$, with $\Delta_a = (\delta_{v,a})_{v \in V}$. Also, for $J \subseteq A$, $\Delta_J = (\Delta_a)_{a \in J}$. Choose any Bijection $B \to V$. Then one can write $b = (b_a)_{a \in B}$ instead of $b = (b_v)_{v \in V}$. Similarly, $\Delta_a = (\delta_{v,a})_{v \in B}$. For enhanced readability, a few further notations should be introduced:

$$\begin{aligned} \overline{b} =& (\overline{b_a})_{a \in B} \coloneqq \Delta_B^{-1} b \\ \overline{\Delta_j} =& (\overline{\delta_{a,j}})_{a \in B} \coloneqq \Delta_B^{-1} \Delta_j \quad \text{for all } j \in A \\ \overline{\Delta_J} \coloneqq& (\overline{\Delta_j})_{j \in J} = \Delta_B^{-1} \Delta_J \quad \text{for all } J \subseteq A. \end{aligned}$$

Since for all non-basic variables we have $f_a = c_a$ (for $a \in C$), or $f_a = d_a$ (if $a \in D$), (1.28) can now compactly be rewritten as

$$f_B = \overline{b} - \overline{\Delta_C} c_C - \overline{\Delta_D} d_D,$$

or, equivalently,

$$f_a = \overline{b_a} - \sum_{j \in C} \overline{\delta_{a,j}} c_j - \sum_{j \in D} \overline{\delta_{a,j}} d_j \quad \text{for all } a \in B. \tag{1.29}$$

Assume for the moment that $e \in C$. In order to increase the value of $U(f)$, let us try to decrease the value of $f_e = c_e$ as much as possible, subject to the requirement that the constraints $\Delta f = b$ and $d \leq f \leq c$ remain valid. Suppose that the value of f_e is decreased by the amount θ, i.e., $f_e = c_e - \theta$. Then for $a \in B$, by (1.29), the value of f_a changes to:

$$f_a = \overline{b_a} - \sum_{j \in C} \overline{\delta_{a,j}} c_j - \sum_{j \in D} \overline{\delta_{a,j}} d_j + \overline{\delta_{a,e}} \theta. \tag{1.30}$$

If $\overline{\delta_{a,e}} > 0$ for some $a \in B$, then the value of the basic variable f_a will gradually increase as the value of θ is increased, until it finally reaches c_a. Similarly, if $\overline{\delta_{a,e}} < 0$ for $a \in B$, then the value of f_a will decrease as the value of θ is increased, until it reaches d_a. If $\overline{\delta_{a,e}} = 0$, then θ can be increased

arbitrarily, without f_a ever leaving its feasible interval. As we would like to maintain the validity of the inequalities $d_a \leq f_a \leq c_a$ for all $a \in B$ at the same time (and hence the feasibility of the current solution), we define for $a \in B$:

$$\theta_a \coloneqq \begin{cases} (\overline{\delta_{a,e}})^{-1}\big(c_a - \overline{b_a} + \sum_{j\in C} \overline{\delta_{a,j}} c_j + \sum_{j\in D} \overline{\delta_{a,j}} d_j\big) & \text{for } \overline{\delta_{a,e}} > 0 \\ \infty & \text{for } \overline{\delta_{a,e}} = 0 \\ (\overline{\delta_{a,e}})^{-1}\big(d_a - \overline{b_a} + \sum_{j\in C} \overline{\delta_{a,j}} c_j + \sum_{j\in D} \overline{\delta_{a,j}} d_j\big) & \text{for } \overline{\delta_{a,e}} < 0 \end{cases}.$$

It is also necessary to take care that the variable f_e does not leave its own feasible interval. Therefore, the maximum possible value for θ that maintains feasibility of the solution, and which is hence chosen, in order to increase the value of the objective function as much as possible, is:

$$\theta \coloneqq \min\big\{c_e - d_e, \min\{\theta_a \mid a \in B\}\big\} \geq 0.$$

If $\theta = c_e - d_e$, then the basis structure remains unaltered, the index e of the variable f_e just moves from C to D. In the remaining case, if $\theta = \theta_a$ for some $a \in B$, then the values of all the variables f_a with $\theta_a = \theta$ assume one of c_a or d_a. One of these variables can be chosen as the leaving variable f_l. Again, there are various possibilities as to which one of them is selected. This can in practice be quite an important topic, the reasons for which will be discussed in the next section.

The other case, in which a variable f_e at its lower bound d_e is selected as the entering variable, i.e., $e \in D$, is completely analogous to the above considerations.

Finally, once a leaving variable f_l has been selected, the triplet (B, C, D) is updated and the system is brought into canonical form with respect to the new basis structure. This can be done using elementary row operations. There exist implementations of the Simplex Algorithm with largely differing computational costs. In particular, in most of the linear programs arising in practice, the constraint matrix Δ contains many entries that are zero (90 percent or more are not a rarity). This can be exploited by modifications to the method discussed above and by specifically designed implementations. For example, the *Revised Simplex Algorithm* is a clever variant in which not all of the columns $\overline{\Delta_j}$ have to be computed in every step and thus the number and complexity of elementary row operations that have to be performed are in most cases drastically reduced.

Finite Termination of the Simplex Algorithm

During the execution of the Simplex Algorithm, in each iteration it moves from one basic feasible solution to another. While doing this, according to

(1.27), the value of the objective function is improved by the amount $u_e^\pi \theta$ in every step. Here e is the index of the entering variable. If $\theta = 0$, the pivot is referred to as *degenerate*. If $\theta > 0$, it is called *nondegenerate*. The situation that $\theta = \infty$ might also possibly arise. Since $u_e^\pi \neq 0$ (if $u_e^\pi = 0$, then f_e does not violate its optimality condition), this means that the value of the objective function can be increased arbitrarily and is hence unbounded. Therefore, no finite optimal solution exists. However, this can only possibly happen if there is some $a \in A$ with $c_a = \infty$ or $d_a = -\infty$.

If every pivot is nondegenerate, the value of the objective function strictly increases as the algorithm proceeds, which means that no solution can occur twice during an execution of the Algorithm. Since there are only $\binom{n}{m}2^{n-m}$ different basis structures, i.e., only so many basic solutions, this would imply the finiteness of the method.

Unfortunately, in general there is no simple way to guarantee the nondegeneracy of every pivot. In fact, in practice typically by far the most pivots are degenerate. As in this case solutions might be repeated any number of times, finiteness of the algorithm cannot be guaranteed without some further precautions. There are several possibilities to ensure finiteness of the Simplex Algorithm. Most of them use refined rules with regard to which variables should be chosen as entering or leaving variables. For example, using a simple lexicographic rule to break ties in the case of multiple entering or leaving candidates, solves the problem successfully.

Concluding, here are again the three possible outcomes of an execution of the Simplex Algorithm:

- The algorithm terminates with a pivot in which $\theta = \infty$. In this case, the objective function is unbounded and hence no finite optimal solution exists.
- The algorithm terminates with an optimal solution in which the value of some artificial variable is nonzero (cf. page 37). In this case the original program has no feasible solution.
- The algorithm terminates with a finite solution in which all artificial variables have zero value. Then this solution is a finite optimal solution for the original program.

1.4.4 The Ellipsoid Method

Although in practice almost useless due to its long average running time, the Ellipsoid Method for solving linear programs has a better worst-case running time than the Simplex Algorithm and has therefore proved useful

for the derivation of some theoretical results. In fact, it was the algorithm with which the polynomiality of the general linear programming problem originally was established, namely by Khachiyan in 1979.

For an introduction see for example [18, Section 8.7], or [21, Section 5.11]. A more in-depth survey including some implications for combinatorial optimization can be found in [9].

In [14, p. 82], the following running time bound for solving the linear program (1.4) using Kachiyan's Ellipsoid Method is given:

$$O\Big(\big(\dim(b)+\dim(c)\big)^9\big(\operatorname{size}(A)+\operatorname{size}(b)+\operatorname{size}(c)\big)^2\Big).$$

Here $\operatorname{size}(x)$ denotes the binary encoding length of x. When compared with the time complexities of the algorithms examined in Chapter 4, this illustrates the practical infeasibility of the Ellipsoid Method for computation of maximal network flows or related problems.

1.4.5 Duality

Duality is arguably the most useful concept in all of linear programming. According to Dantzig, the first one to point out the power of this principle was John von Neumann (cf. [5, p. 29]). This piece of work deals mainly with a specialization of the general duality principles, namely with the concepts that result when applied to networks.

1.24 Definition. Associated with every linear program is its *dual program*. For the program (1.4), which in this context is called the *primal program*, it looks as follows:

$$\begin{aligned} &\text{Find } y \in \mathbb{R}^m \text{ such that}\\ \text{(i)}\quad & y \geq \mathbf{0}\\ \text{(ii)}\quad & y^{\mathrm{T}}A = c^{\mathrm{T}}\\ \text{(iii)}\quad & y^{\mathrm{T}}b \text{ is minimal.} \end{aligned}$$

Or shorter,

$$\min\{y^{\mathrm{T}}b \mid y \geq \mathbf{0},\, y^{\mathrm{T}}A = c^{\mathrm{T}}\}. \tag{1.31}$$

Analogously to Definition 1.20, the mapping $y \mapsto y^{\mathrm{T}}b$ is called *objective function* or *cost function*.

The following is the most important theorem of this chapter. It is one of the most fundamental theorems in linear programming. Proofs can be found in nearly all introductory books on linear programs. It was first proved by

von Neumann in 1947. This version is taken from [21, p. 62]. The Max-Flow Min-Cut Theorem, that I want to examine in detail in Section 2.4, is a special case of the general Duality Theorem of Linear Programming. This fact will be proved in Section 3.2.

1.25 Theorem (Duality Theorem of Linear Programming)**.** *Let $m, n \in \mathbb{N}$ and $A \in M_{m,n}(\mathbb{R}), b \in \mathbb{R}^m, c \in \mathbb{R}^n$. Then*

$$\max_{x \in \mathbb{R}^n} \{c^T x \mid Ax \leq b\} = \min_{y \in \mathbb{R}^m} \{y^T b \mid y \geq \mathbf{0},\, y^T A = c^T\}, \tag{1.32}$$

if at least one of these optima is finite.

1.26 Remark. It is not hard to show that the minimum is always greater or equal the maximum in (1.32), not only in the case of finiteness of one of the optima. Provided finiteness is known, the converse inequality is harder to prove. For the special case of networks, I shall deduce this statement in Section 2.4. From Theorem 1.25, it follows that there are only the following possibilities for a pair of dual programs:[7]

- primal program has a finite optimum $\Longrightarrow$ dual program has a finite optimum
- primal program has an unbounded objective function $\Longrightarrow$ dual program is infeasible
- primal program is infeasible $\Longrightarrow$ dual program is either infeasible, or feasible and has an unbounded objective function

Simple examples show that all of these cases can actually occur (see for example [18, p. 70f.]). Hence, there are *exactly* the above possibilities for a pair of dual programs.

1.27 Remark. As mentioned before, every linear program can be written in many equivalent ways. Moreover, the dual program (1.31) can be written as a primal program of the form (1.4). It is not hard to verify that then the dual of the dual is again the original primal program. Hence, which program deserves to be called the primal one, and to which the adjective dual should be assigned, is quite an arbitrary choice.

I now want to state the Duality Theorem of Linear Programming in a more general version, as the linear programs that will come up in our study of networks are not exactly in the form for which Theorem 1.25 is formulated. This formulation is also taken from [21, p. 62]:

[7] this is taken from [18, p. 70]

1.28 Theorem (General Duality Theorem of Linear Programming)**.** *Let A, B, C, D, E, F, G, H, K be matrices and α, β, γ, δ, ε, ζ be vectors. Assume that all dimensions in the following expression are compatible. Then*

$$\begin{aligned}\max\{\delta^T x + \varepsilon^T y + \zeta^T z \mid\ & x \geq \mathbf{0},\ z \leq \mathbf{0},\\ & Ax + By + Cz \leq \alpha,\\ & Dx + Ey + Fz = \beta,\\ & Gx + Hy + Kz \geq \gamma\}\\ = \min\{u^T\alpha + v^T\beta + w^T\gamma \mid\ & u \geq \mathbf{0},\ w \leq \mathbf{0},\\ & u^TA + v^TD + w^TG \geq \delta^T,\\ & u^TB + v^TE + w^TH = \varepsilon^T,\\ & u^TC + v^TF + w^TK \leq \zeta^T\},\end{aligned}$$

in case that at least one of these optima is finite.

Chapter 2

A Combinatorial Approach to Maximal Flow

In this chapter, I am going to introduce networks, the objects that are examined in the remainder of this text. Networks are basically digraphs, just with some additional numbers assigned to the arcs and vertices. Flows can be thought of as some quantity that is exchanged—or 'flows'—along the arcs of these networks. The problem of finding flows satisfying certain properties can be regarded as a linear optimization problem. Although this might not be completely apparent at first sight, if formulated appropriately, the translation from graph theory to linear programming, or vice versa, is a very natural one. First, I want to discuss networks and flows utterly in graph theoretical terms. Then, in Chapter 3, I will show what the very same theory looks like if formulated in the language of linear programming. In order to achieve maximal consistency, and to emphasize the equivalence between the two approaches, I tried to choose the notation such that the change is effortless and intuitive.

In Section 2.1, a certain function is described that will reappear frequently later on. Section 2.2 contains the definition of most of the central objects, as well as some of their basic properties and a couple of illustrating examples. The subsequent Section 2.3 is dedicated to cuts, a notion that is dual to the one of flows. Most of the theory of network flows could equally well be formulated focussing on cuts rather than on flows. Sections 2.4, respectively 2.5, consist mainly of the detailed proof of the famous Max-Flow Min-Cut Theorem, respectively the completely analogous Min-Flow Max-Cocapacity Theorem. Section 2.6 addresses a topic that is suspended in the preceding sections: examining the exact assumptions needed to guarantee the existence of flows of certain types for a given network. Finally, in Section 2.7, various alternative definitions of networks that are found in the literature are

presented.

In the line of proof of the central theorem of this chapter, the Max-Flow Min-Cut Theorem, in both its classical and its slightly more general version, I mostly follow [21, Chapter 10]. There you can find a very clear, concise and elegant exposition of the topic. The structure and the ideas used are more or less the same as in the classical monograph [8] by Ford and Fulkerson. The proof is almost constructive, i.e., an instruction how to find a maximal flow can be inferred from it. Except for one point, that is, which is left open: there is no recipe given how to find a so-called *augmenting path*. Augmenting paths are used to construct a flow of greater value from a given flow. In Section 4.2, we will see how the proof of the Max-Flow Min-Cut Theorem gives rise to different algorithms that find a maximal flow, the various algorithms basically differing in the way an augmenting path is chosen.

In graph theory in general, there are no fully established notational conventions. Of course few of the alternatives that are in use represent major qualitative differences between the various approaches, but the deviations are still numerous and big enough to be annoying, and sometimes they cause irritation when reading texts from an unknown author. The same is true for the subject area of this thesis; there is no real standard terminology for network flow theory. Over the years many variations have surfaced and disappeared again. I tried to use a terminology that I found most intuitive and flexible. To this end, I used ideas from [21] and [2], as well as some new elements that I found indespensable.

2.1 The Function excess

First, I want to introduce a function that will prove hugely useful for the formulation and the proof of the Max-Flow Min-Cut Theorem.

Remember that for any function $f\colon A \to \mathbb{R}$ and any finite subset $U \subseteq A$, we have set $f(U) \coloneqq \sum_{u\in U} f(a)$. Hence, $f(\emptyset) = 0$.

2.1 Definition (Excess Function)**.** Let $D = (V, A)$ be a digraph and let $f\colon A \to \mathbb{R}$ be some function. Define the *excess function* of f as

$$\operatorname{excess}_f\colon V \to \mathbb{R},\ v \mapsto f(\delta^{\mathrm{in}}(v)) - f(\delta^{\mathrm{out}}(v)).$$

2.2 Lemma. *Let $D = (V, A)$ be a digraph, $f\colon A \to \mathbb{R}$ a function and $U \subseteq V$ arbitrary. Then*

$$\operatorname{excess}_f(U) = f(\delta^{in}(U)) - f(\delta^{out}(U)).$$

Proof. We have

$$\begin{aligned} \operatorname{excess}_f(U) &= \sum_{u \in U} \operatorname{excess}_f(u) \\ &= \sum_{u \in U} \Big(\sum_{a \in \delta^{\text{in}}(u)} f(a) - \sum_{a \in \delta^{\text{out}}(u)} f(a) \Big), \text{ and} \qquad (2.1) \\ f(\delta^{\text{in}}(U)) - f(\delta^{\text{out}}(U)) &= \sum_{a \in \delta^{\text{in}}(U)} f(a) - \sum_{a \in \delta^{\text{out}}(U)} f(a). \qquad (2.2) \end{aligned}$$

Now count the multiplicities with which the terms on the right-hand sides of (2.1) and (2.2) occur. All the terms are of one of the forms $f(a)$ or $-f(a)$, for some $a \in A$. Choose $a = (u, v) \in A$. There are four cases:

- First, suppose $u \in U$, $v \notin U$, i.e., $a \in \delta^{\text{out}}(U)$. Then $-f(a)$ occurs exactly once in (2.1) and (2.2).
- Second, $u \notin U$, $v \in U$, i.e., $a \in \delta^{\text{in}}(U)$. Then $f(a)$ occurs exactly once in (2.1) and (2.2).
- Third, $u, v \notin U$. In neither (2.1) nor (2.2) any term involving a occurs.
- Fourth, $u, v \in U$. This is the only interesting case. Neither $f(a)$ nor $-f(a)$ occurs in (2.2), but in (2.1), there are two vertices w with $a \in \delta^{\text{in}}(w) \cup \delta^{\text{out}}(w)$ (namely $w = u$ and $w = v$). So we have the terms $f(a)$ and $-f(a)$ once each, adding to 0.

We have thus seen term-wise equality between the right-hand sides of (2.1) and (2.2). This completes the proof. □

2.2 Networks and Flows

Networks can be found everywhere around us. Personal networks, telecomunication systems, urban traffic, pipelines, ... Although appearing in many different forms, they all share some basic common features. These are condensed in the mathematical description given below. This description is simple enough that one could hope that even persons with no mathematical knowledge are able to comprehend it, while also general enough to capture most characteristics one would expect intuitively from a network.

Often associated with a network, in its heuristic sense, is something that moves along the network, something that is exchanged through it. The network usually is not a static object, constructed for its own sake, but rather

functions as a carrier for something else, be it water, people, information, money, ... This 'something' that travels around a network will abstractly be introduced as *flow*.

The following is the central definition of this exposé, in which the objects of our main interest are introduced. The remaining text is mainly a survey on these notions.

2.3 Definition (Networks, Flows, Balance Vectors, Capacity and Demand Functions)**.** Let $D = (V, A)$ be a digraph, $S, T \subseteq V$, and let $c, d\colon A \to \overline{\mathbb{R}}$, $f\colon A \to \mathbb{R}$, and $b\colon V \to \overline{\mathbb{R}}$ be functions. In the context of networks, such a function f is called a *flow in* D. More specifically, a flow f in D is called a b, c, d*-flow from* S *to* T *in* D, or an (S, T, b, c, d)*-flow in* D, or simply a *feasible flow in* D, if and only if

$$\text{(i)} \quad \operatorname{excess}_f(v) = b(v) \qquad \forall v \in V\backslash(S \cup T), \tag{2.3}$$

$$\text{(ii)} \quad f(a) \leq c(a) \qquad \forall a \in A, \tag{2.4}$$

$$\text{(iii)} \quad f(a) \geq d(a) \qquad \forall a \in A. \tag{2.5}$$

b is called a *balance vector*, c a *capacity function*, and d a *demand function* for D. The interval $[d(a), c(a)] \subseteq \overline{\mathbb{R}}$ is called *capacity interval of* a. It could be empty or unbounded. I will refer to the quintuple (S, T, b, c, d) as a *flow-quintuple* for D.

(D, S, T) is usually referred to as a *network*. The vertices $s \in S$ are called the *sources* of the network, the vertices $t \in T$ its *sinks*. For every such triplet (D, S, T), I will, for convenience and readability, abbreviate $R \coloneqq V\backslash(S \cup T)$. Sometimes, I will refer to a graph D together with a whole flow-quintuple (S, T, b, c, d) as a network. If for such a network (D, S, T, b, c, d) some feasible flow exists, the network is called *feasible*. Otherwise, it is called *infeasible*.

For $b = \mathbf{0}_R$, the condition (2.3) is usually called *flow-condition* or *flow-conservation law*. To avoid confusion, I will use the term *balance-condition* if $b \neq \mathbf{0}_R$.

I denote the set of all flows in D by $\mathcal{F}(\mathrm{D})$. For the set of all (S, T, b, c, d)-flows in D, I write $\mathcal{F}_{S,T}^{b,c,d}(D)$. In all of the above notations, if $S = \{s\}$ or $T = \{t\}$, I will usually leave out the braces. If it is understood from the context which digraph I am talking about, I will omit specific reference to D.

An (S, T, b, c, d)-flow f is called *integral* if it is an integer-valued function, i.e., if $f(A) \subseteq \mathbb{Z}$.

2.4 Remark. Clearly, a feasible flow f as in the above definition can only exist if $c \geq d$. A few additional examples of networks with $\mathcal{F}_{S,T}^{b,c,d} = \emptyset$ are depicted in Figures 2.1–2.3 on page 51.

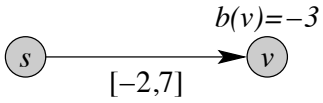

Figure 2.1: An infeasible network.

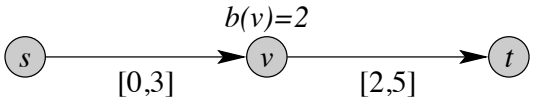

Figure 2.2: Another infeasible network.

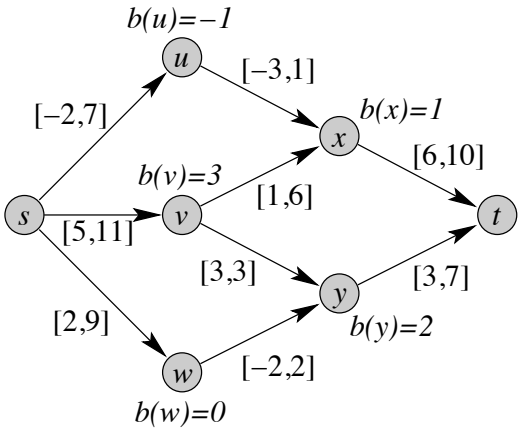

Figure 2.3: Yet another infeasible network.

In Figure 2.1, a simple graph with only two vertices and one arc (weakly) connecting them is depicted. If we set $T := \emptyset$, then there is no feasible $s-T$-flow: such a flow f would have to satisfy $\text{excess}_f(v) = f(\delta^{\text{in}}(v)) - f(\delta^{\text{out}}(v)) = -3$. But for any choice of f we have $f(\delta^{\text{out}}(v)) = f(\emptyset) = 0$. As $f(\delta^{\text{in}}(v)) = f((s,v)) \geq d((s,v)) = -2$, the balance-condition of vertex v cannot be satisfied by any flow f with $d \leq f \leq c$. Hence, no flow in the graph is feasible.

Figure 2.2 shows the network (D, s, t, b, c, d). Here

$$D = \Big(\big\{s, v, t\big\}, \big\{(s,v), (v,t)\big\}\Big).$$

The balance-condition for v, $b(v) = 2$, tells us that 2 flow-units more must enter v than leave v. But at most $c((s,v)) = 3$ units can enter v. At least $d((v,t)) = 2$ have to leave v. These are contradictory requirements, whence the network is infeasible.

Finally, the network depicted in Figure 2.3 is a little bit more complex. The task of verifying that no feasible $s-t$-flow for this network exists is left to the reader. This example is meant to illustrate the fact that in bigger networks it might be a rather nontrivial question whether or not there exists a feasible flow.

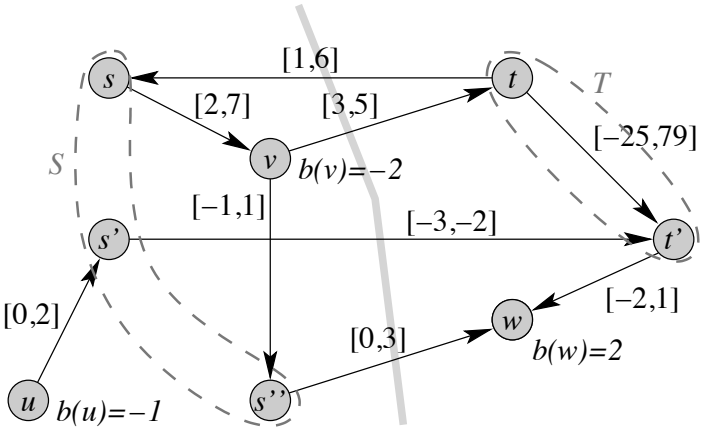

Figure 2.4: A typical network.

Necessary and sufficient conditions for the existence of feasible flows in a given network are discussed in more detail in Section 2.6. Moreover, an algorithm is presented that finds a feasible flow or decides that there is none.

2.5 Example. The network of Figure 2.4 is a bit more complicated than the previous examples. We have:

$$
\begin{aligned}
&V = \{s, s', s'', t, t', u, v, w\},\\
&A = \{(s,v), (s',t'), (s'',w), (t,s), (t,t'), (t',w), (u,s'), (v,s''), (v,t)\},\\
&S = \{s, s', s''\}, \text{ the set of sources,}\\
&T = \{t, t'\}, \text{ the set of sinks, and}\\
&R = \{u, v, w\}.
\end{aligned}
$$

The closed interval next to each arc a shall indicate the lower and upper capacity bounds $[d(a), c(a)]$. Furthermore, the values of b are written next to the respective vertices. Let us try to find out whether there is a feasible (S, T, b, c, d)-flow for D.

For this purpose, we first try to match the balance-vectors of u, v, and w. No flow can enter u, therefore $-1 = \text{excess}_f(u) = -f(\delta^{\text{out}}(u)) = -f((u, s'))$. This implies that a feasible flow must satisfy $f((u, s')) = 1$, which luckily lies in the capacity interval $[0, 2]$ of (u, s').

For vertex v, from the demand and capacity constraints on adjacent arcs, we infer $2 \leq f(\delta^{\text{in}}(v)) \leq 7$ and $2 \leq f(\delta^{\text{out}}(v)) \leq 6$. As 2 units more should leave v than enter it, this can clearly be achieved (e.g., set $f((s, v)) = 4$, $f((v, s'')) = 1$, $f((v, t)) = 5$).

Skilled as we are by now, it is easy to see that $f((s'', w)) = 3$, $f((t', w)) = -1$ is a solution for w that respects $b(w)$ as well as all involved demands and capacities. Setting the values of f arbitrarily within the respective capacity intervals of the remaining arcs completes our construction of a feasible $f \in \mathcal{F}_{S,T}^{b,c,d}$:

$$f((t, s)) = 1,\ f((t, t')) = -17,\ f((s', t')) = -2.$$

Note that these choices make f an integer flow.

2.6 Definition (Value of a Flow)**.** Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. Let f be a (not necessarily feasible) flow. The real number

$$\text{value}(f) \coloneqq f(\delta^{\text{out}}(S)) - f(\delta^{\text{in}}(S))$$

is called the *value of* f. A feasible flow is called *maximal* if and only if it is of maximum value (in the set of feasible flows). Analogously, it is called *minimal* if and only if it is of minimum value (in the set of feasible flows).

2.7 Example. For the feasible flow constructed in Example 2.5, we have:

$$\begin{aligned}\text{value}(f) &= f(\delta^{\text{out}}(S)) - f(\delta^{\text{in}}(S))\\ &= f((s, v)) + f((s', t')) + f((s'', w)) - f((t, s)) - f((u, s')) - f((v, s''))\\ &= 4 + (-2) + 3 - 1 - 1 - 1 = 2\end{aligned}$$

Is this a maximal flow or can we find one of greater value? Or maybe it happens to be minimal? We take a heuristic approach and see if we can answer these questions straightforwardly.

We have seen that *any* feasible flow must satisfy $f((u, s')) = 1$, no room for improvement here. The arcs (t, s) and (s', t') are easy to handle, too: the only constrains that have to be met are given by capacity and demand. Hence, to achieve maximality, flow will be at the lower bound on arc (t, s) and at its capacity on arc (s', t'), whereas for a minimal flow, it will be the other way round. $f((t, t'))$ has no influence on value(f), so we can choose that as we like within capacity bounds. As f, constructed in Example 2.5, is at the upper bound on arc (s'', w) already, no other flow could do better there. To achieve little value, we would rather set $f((s'', w)) = 1 = f((t', w))$, which also satisfies the balance prescribed for w.

Slightly more complicated is the situation at v, since there all of the three remaining arcs are involved. Two units more must flow out of v than into it, and for maximal value, we have to maximize $f((s, v)) - f((v, s''))$. At most five units can leave v via (v, t). So all we send into v exceeding 3 units along (s, v) must re-enter S via (v, s''), such that nothing would be gained or lost. Here lies a certain flexibility, there is more than one choice for a maximal

flow. The choice for f from above can be seen to be maximal already. For minimal flows, the situation is different: of course we want $f((s,v))$ to be no more than 2 if possible, and ideally $f((v,s'')) = 1$ as well. But this becomes feasible by setting $f((v,t)) = 3$. Consequently, here the choice for a minimal flow is unique.

We summarize our findings in the following table, giving an example of a maximal, and the unique minimal flow for the network in Figure 2.4 on page 52:

	f_{max}	f_{min}
(s,v)	4	2
(s',t')	-2	-3
(s'',w)	3	1
(t,s)	1	6
(t,t')	73	37
(t',w)	-1	1
(u,s')	1	1
(v,s'')	1	1
(v,t)	5	3
value	2	-8

One thing these calculations were supposed to show is that the utilized, fairly heuristic and intuitive, approach is a little clumsy and likely to not be efficient when the problems become more complex and the networks bigger. In fact, describing the used 'algorithm' formally, such that a computer could execute the procedure, one would probably get a highly inefficient tool for finding maximal flows. It is also not clear at all what such a description would look like. It might not always be possible to decide directly—as we managed in the last example—which decisions are the right ones, if the 'interdependencies' between constraints and requirements on different arcs and vertices become more involved

Consequently, one might hope to find better algorithms that are easier to describe, that are guaranteed to work for every network, and that are more efficient than our heuristic attempt. Luckily, there are many such algorithms readily available. A discussion of a few of the most popular ones is given in Chapter 4.

A straightforward calculation verifies the following fact, that will be needed later on:

2.8 Lemma (Continuity of the Function value)**.** *The function* value: $\mathcal{F} \to \mathbb{R}$ *is linear. Consequently, it is continuous.*

2.9 Remark. Let D, (S, T, b, c, d), and f be as in Definition 2.6. The value of f can be regarded as the net amount of 'flow' leaving S. Taking $U = V$ in Lemma 2.2, it can be seen that

$$\begin{aligned} 0 = \operatorname{excess}_f(V) &= \sum_{s\in S} \operatorname{excess}_f(s) + \sum_{v\in R} \operatorname{excess}_f(v) + \sum_{t\in T} \operatorname{excess}_f(t) \\ &= \operatorname{excess}_f(S) + \sum_{v\in R} b(v) + \operatorname{excess}_f(T). \end{aligned}$$

Therefore,

$$\operatorname{value}(f) = -\operatorname{excess}_f(S) = b(R) + f(\delta^{\text{in}}(T)) - f(\delta^{\text{out}}(T)).$$

For $b(R) = 0$ (e.g., if $b = \mathbf{0}_R$), this amounts to the net amount of flow leaving S being equal to the net amount of flow entering T.

For $S = \emptyset$, all $f \in \mathcal{F}(D)$ satisfy $\operatorname{value}(f) = 0$. So in this case, to search for a maximum flow (which is the topic of Chapter 4) is equivalent with just searching for *any* feasible flow.

2.10 Remark. As mentioned before, there are many variations in the terminology used to describe networks. A lot of authors (certainly not all though) prefer to define networks in forms that appear to be less general than how I introduced them above. For example, usually the set of sources and the set of sinks are required to contain exactly one element. But this is not really a confinement, since the 'general' case can easily be reduced to the one-source one-sink-case: just adjoin two additional vertices s_0 and t_0 (a 'super-source' and a 'super-sink') to the network, along with arcs of unlimited capacity and zero demand, of the form (s_0, s) for every $s \in S$ and (t, t_0) for every $t \in T$. Also, set $b(v) = 0$ for all $v \in S \cup T$. Then there is a one-to-one correspondence between the ST-flows of value W in the augmented network and the $s_0 - t_0$-flows of value W in the original network.

Similar auxiliary networks can be constructed for reductions of the 'general' version to the cases $b = \mathbf{0}_R$ respectively $d = \mathbf{0}_A$ (see [2, p. 101] or [21, p. 175f.], respectively [2, p. 99f.] or [4, p. 140–143]). Another detailed elaboration on transformations between different network models can be found in [1, p. 38–46].

The case of $S = T = \emptyset$ is most often treated separately and called a *circulation* (for $b = \mathbf{0}_R$) or a (b-)*transshipment* (if $b \neq \mathbf{0}_R$ is allowed). The classical case considered by Ford and Fulkerson in [8], as well as by most

other authors on the subject, is

$$\begin{aligned} S =& \{s\}, \\ T =& \{t\}, \\ b =& \mathbf{0}_R, \\ c \geq& \mathbf{0}_A, \\ d =& \mathbf{0}_A. \end{aligned}$$

Summarizing, it can be said that the most often encountered approach is to define network flows in a seemingly specialized form and then reduce all generalizations to the basic model.

When I was studying Ford and Fulkerson's proof of the Max-Flow Min-Cut Theorem, I found that there was no real alteration necessary in order to formulate and prove the different lemmas and theorems with the terminology introduced above (hardly more than using capital letters instead of some lowercase ones had to be changed; and writing $d(a)$ or $\sum_{v \in R} b(v)$ instead of 0 in a few places). Moreover, the basic ideas were not disguised by the relatively few additional technical details. Therefore, I found it very natural to use this terminology. I also tried to add a few further 'generalizations' that are in common use (all of which can be reduced to the more basic network definitions with little effort). But in most cases I did not quite see how to include them in the proofs without having to use rather different approaches, or getting technically cluttered. That is why I decided to introduce the notions of networks and flows in just the generality that can be seen above.

2.11 Remark. The set $\mathcal{F}(D)$ is isomorphic to the real vectorspace $\mathbb{R}^A$. It is partially ordered by the relation

$$f_1 \leq f_2 :\Longleftrightarrow f_1(a) \leq f_2(a) \text{ for all } a \in A.$$

The following is a total pre-order on $\mathcal{F}(D)$:

$$f_1 \leq f_2 :\Longleftrightarrow \text{value}(f_1) \leq \text{value}(f_2).$$

Furthermore, the set $\mathcal{F}_{S,T}^{0,\infty,0}(D)$ is a submonoid of $\mathcal{F}(D)$ (with respect to the arc-wise addition of flows), whereas $\mathcal{F}_{S,T}^{0,\infty,-\infty}(D)$ is even a subspace.

All of these claims can be verified in an elementary way. As I will not need them and have stated them just for illustration, the proofs are omitted.

2.12 Lemma (Properties of $\mathcal{F}_{S,T}^{b,c,d}(D)$)**.** *Let $D = (V, A)$ be a digraph and let (S, T, b, c, d) be a flow-quintuple for D. Then the set $\mathcal{F}_{S,T}^{b,c,d}(D)$ is a closed, convex subset of $\mathcal{F}(D)$. It is bounded and hence compact if $-\infty < d$ and $c < \infty$.*

Proof. If $\mathcal{F}_{S,T}^{b,c,d}(D) = \emptyset$, then it is a closed, convex and bounded set trivially. Now suppose $\mathcal{F}_{S,T}^{b,c,d}(D) \neq \emptyset$. To see convexity, let $f, g \in \mathcal{F}_{S,T}^{b,c,d}(D)$ and $0 \leq \lambda \leq 1$. We have $d \leq f$, $g \leq c$, implying

$$d = \lambda d + (1-\lambda)d \leq \lambda f + (1-\lambda)g \leq \lambda c + (1-\lambda)c = c.$$

This is the capacity constraints (2.5) and (2.4) for the flow $\lambda f + (1-\lambda)g$. To verify the balance-condition (2.3), let $v \in R$. Then

$$\begin{aligned}\operatorname{excess}_{\lambda f+(1-\lambda)g}(v) &= (\lambda f + (1-\lambda)g)(\delta^{\text{in}}(v)) - (\lambda f + (1-\lambda)g)(\delta^{\text{out}}(v)) \\ &= \lambda\big(f(\delta^{\text{in}}(v)) - f(\delta^{\text{out}}(v))\big) + (1-\lambda)\big(g(\delta^{\text{in}}(v)) - g(\delta^{\text{out}}(v))\big) \\ &= \lambda b(v) + (1-\lambda)b(v) = b(v).\end{aligned}$$

To see that $\mathcal{F}_{S,T}^{b,c,d}(D)$ is a closed set, observe that all of the following sets are closed:

$$\begin{aligned}&\{f \in \mathcal{F}(D) \mid f \leq c\},\\ &\{f \in \mathcal{F}(D) \mid f \geq d\},\\ &\{f \in \mathcal{F}(D) \mid \forall v \in V\colon \operatorname{excess}_f(v) = b(v)\}.\end{aligned}$$

Since the proof of this last statement is analytic and can be found in most introductory textbooks on analysis, let us just assume it is true.[1] Now, $\mathcal{F}_{S,T}^{b,c,d}(D)$ is the intersection of these three sets, therefore it is closed as well.

That $\mathcal{F}_{S,T}^{b,c,d}(D)$ is bounded, if $-\infty < d$ and $c < \infty$, is quite apparent. It is for example contained in the closed ball with centre $\mathbf{0}_A$ and radius $\max\{|d|, |c|\}$, which is a bounded set because $\mathcal{F}(D)$ is a finite-dimensional vector space. □

2.13 Lemma (Existence of Minimal and Maximal Flows)**.** *Let $D = (V, A)$ be a digraph. Let (S, T, b, c, d) be a flow-quintuple for D. Suppose that*

$$\mathcal{F}_{S,T}^{b,c,d} \neq \emptyset, \tag{2.6}$$

$$-\infty < d, \text{ and} \tag{2.7}$$

$$c < \infty. \tag{2.8}$$

Then there exists an (S, T, b, c, d)-flow of maximum value, as well as an (S, T, b, c, d)-flow of minimum value.

Proof. This follows from the previous theorem and from Theorem 1.18 on page 27: by (2.6) – (2.8), $\mathcal{F}_{S,T}^{b,c,d}(D)$ is a nonempty, compact subset of the metric space $\mathcal{F}(D)$. As value is a continuous function by Lemma 2.8, it has a maximum as well as a minimum in $\mathcal{F}_{S,T}^{b,c,d}(D)$. □

[1] Good introductions to analysis include [11] and [20].

2.3 Cuts

In this section, a concept is introduced that is in a certain sense dual to the one of flows in networks. What is nowadays usually called a *cut*, is what originally initiated the study of network flows.[2] In Chapter 1.4, we will see how the relationship between flows and cuts corresponds to the relationship between primal and dual linear programs. First, I give the definition of a related notion that will be needed in the sequel, although only on rare occasions.

2.14 Definition (Disconnecting Arc Sets)**.** Let $D = (V, A)$ be a digraph and $S, T \subseteq V$. A subset $C \subseteq A$ is called an *$S - T$-disconnecting arc set* if and only if it intersects every $S - T$-path in D. If $S = \{s\}$, or $T = \{t\}$, the braces are left out.

Let us now turn our attention to a certain, quite important, class of disconnecting arc sets. Although I formally introduce cuts not as sets of arcs but rather as sets of vertices, the connection will become apparent soon. The following is one of the central definitions in this thesis:

2.15 Definition (Cuts)**.** Let $D = (V, A)$ be a digraph. A subset $U \subseteq V$ is called a *cut* of D. Let $S, T \subseteq V$ be two sets of vertices. Then U is called an $S - T$-cut if and only if

$$U \supseteq S, \text{ and} \tag{2.9}$$
$$U \cap T = \emptyset. \tag{2.10}$$

I denote the set of all cuts of D by $\mathcal{C}(D)$ and the set of all $S - T$-cuts of D by $\mathcal{C}_{S,T}(D)$. If $S = \{s\}$ or $T = \{t\}$, the braces are left out. In this case, conditions (2.9) and (2.10) amount to $s \in U$, $t \notin U$.

If (S, T, b, c, d) is a flow-quintuple for D, then for any cut $U \subseteq V$, the *capacity of U* is defined as

$$\text{capacity}(U) \coloneqq c(\delta^{\text{out}}(U)) - d(\delta^{\text{in}}(U)) + b(U \backslash S).$$

I also define the cocapacity *of U* as

$$\text{cocapacity}(U) \coloneqq d(\delta^{\text{out}}(U)) - c(\delta^{\text{in}}(U)) + b(U \backslash S).$$

A *minimum ($S - T$-) cut* is a cut $U \in \mathcal{C}_{S,T}(D)$ of minimum capacity.

[2] cf. Section 5.2.1

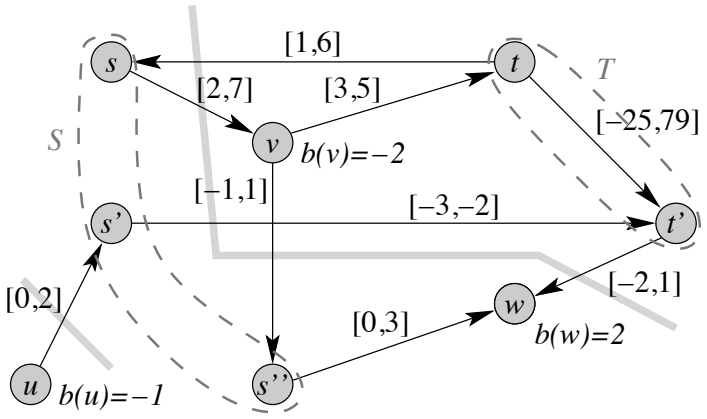

Figure 2.5: A typical cut.

2.16 Example. Let us revisit the network $(D = (V, A), S, T)$ examined in Examples 2.5 and 2.7. Figure 2.5 shows a cut for this network. To be more exact, $U := \{s, s', s'', w\}$ is an $S - T$-cut, and the thick light grey line marks the arcs in $\delta^{\text{out}}(U) \cup \delta^{\text{in}}(U)$. U is also an $s - T$-cut, an $s' - t'$-cut, etc. The partition $(U, V\backslash U)$ of the vertices of the network corresponds to another cut as well: the $T - S$-cut $W := V\backslash U = \{t, t', u, v\}$. We have $\delta^{\text{out}}(W) \cup \delta^{\text{in}}(W) = \delta^{\text{out}}(U) \cup \delta^{\text{in}}(U)$. Let us calculate the capacity and the cocapacity of U:

$$\begin{aligned}
\text{capacity}(U) &= c(\delta^{\text{out}}(U)) - d(\delta^{\text{in}}(U)) + b(U\backslash S) \\
&= c(\{(s, v), (s', t')\}) - d(\{(t, s), (t', w), (u, s'), (v, s'')\}) + b(w) \\
&= (7 + (-2)) - (1 + (-2) + 0 + (-1)) + (2) = 9; \\
\text{cocapacity}(U) &= d(\delta^{\text{out}}(U)) - c(\delta^{\text{in}}(U)) + b(U\backslash S) \\
&= d(\{(s, v), (s', t')\}) - c(\{(t, s), (t', w), (u, s'), (v, s'')\}) + b(w) \\
&= (2 + (-3)) - (6 + 1 + 2 + 1) + (2) = -9.
\end{aligned}$$

2.17 Remark. Here again, I have chosen to use terminology which slightly differs from all variants I have so far encountered in textbooks on network flows. If one is not familiar with the notion of cuts in graphs, then it is at first not at all apparent why an arbitrary set of vertices should now suddenly be called a *cut*. Indeed, the above definition gives at first little inside into the idea behind cuts.

The essential fact about cuts is that they separate the vertices of a (di)graph into two classes: the vertices which belong to a set $U \subseteq V$, and the ones that belong to $V\backslash U$. It 'cuts' the (di)graph into two parts. Therefore, a cut is often defined as a pair $(U, \overline{U})$, where $\overline{U}$ is the complement of U in V

(although one of the two sets in the pair is clearly redundant).

On the other hand, the object one is actually *really* interested in, in network flow theory, is the set of arcs $\delta^{\text{out}}(U)$. Hence, this object should actually deserve to be called a cut (a terminology for example used by Schrijver in [21]). Nice about this notation is, that in the (standard) case with $d = \mathbf{0}_A$, $b = \mathbf{0}_R$, the capacity of a cut can simply be defined as the thing one would expect it to be, namely as $c(\delta^{\text{out}}(U))$.

Unfortunately, the set U is not uniquely determined by $\delta^{\text{out}}(U)$ unless one has some further information, like $\delta^{\text{in}}(U)$. Since it is sometimes convenient to have a well-defined set U of vertices at hand when speaking of a cut, one again would have to determine a *pair*, such as $(\delta^{\text{out}}(U), \delta^{\text{in}}(U))$ or $(\delta^{\text{out}}(U), U)$ (this last version again being redundant).

Summarizing, I felt the most natural and easiest compromise would be to give the definition you can see above, while bearing in mind that one is actually examining $\delta^{\text{out}}(U)$ and $\delta^{\text{in}}(U)$. To remind of this fact, a peculiar name is given to a simple set of vertices.

2.18 Remark. Let $D = (V, A)$ be a digraph. In correspondence to Remark 2.11, the sets $\mathcal{C}(D)$ and $\mathcal{C}_{S,T}(D)$ are partially ordered by the relation

$$C_1 \leq C_2 :\Longleftrightarrow C_1 \subseteq C_2.$$

A total pre-order for both sets is given by

$$C_1 \leq C_2 :\Longleftrightarrow \text{capacity}(C_1) \leq \text{capacity}(C_2).$$

Note that if S is empty, then $\emptyset$ is an $S - T$-cut (of capacity 0) for every $T \subseteq V$, since then $\emptyset \supseteq S$, and $\emptyset \cap T = \emptyset$. If $T = \emptyset$, then V is an $S - T$-cut (of capacity 0) for every $S \subseteq V$, since $\emptyset = \delta^{\text{out}}(V)$, $V \supseteq S$, and $V \cap T = \emptyset$.

2.19 Remark. Let $D = (V, A)$ be a digraph. We always have $\mathcal{C}(D) \neq \emptyset$, since $\emptyset \in \mathcal{C}(D)$. In fact, with the terminology chosen here, $\mathcal{C}(D)$ is nothing but 2^V.

As for the existence of $S - T$-cuts, (2.9) and (2.10) imply the necessity of $S \cap T = \emptyset$. In fact, we have

$$\mathcal{C}_{S,T}(D) \neq \emptyset \Longleftrightarrow S \cap T = \emptyset,$$

since then $U \coloneqq S$ satisfies (2.9) and (2.10). In this case, $\mathcal{C}_{S,T}(D) \cong 2^R$.

2.4 The Max-Flow Min-Cut Theorem

As mentioned before, the proof elaborated below is more or less constructive. It is basically the same proof as originally given by Ford and Fulkerson. There

have been proposed other proofs as well. For example, the Max-Flow Min-Cut Theorem could be derived from the famous theorem by Menger about disjoint paths. The reverse implication holds as well, which is proved in Section 5.1.1. There are also some other combinatorial results that imply the Max-Flow Min-Cut Theorem.

An alternative approach is considered in Chapter 3, where flow theory is regarded as a branch of linear programming, and consequently treated with linear programming techniques. In that way, another (nonconstructive) proof of the Max-Flow Min-Cut Theorem is obtained.

2.20 Theorem. *Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. Then, for every (S, T, b, c, d)-flow f and for every $S - T$-cut U, the inequalities*

$$\text{value}(f) \leq \text{capacity}(U) \tag{2.11}$$

$$\text{cocapacity}(U) \leq \text{value}(f) \tag{2.12}$$

are valid. Equality in (2.11) holds if and only if both

$$f(a) = c(a) \text{ for every } a \in \delta^{out}(U) \text{ and} \tag{2.13}$$

$$f(a) = d(a) \text{ for every } a \in \delta^{in}(U) \tag{2.14}$$

hold. Equality in (2.12) holds analogously if and only if both of the following conditions hold:

$$f(a) = d(a) \text{ for every } a \in \delta^{out}(U) \text{ and} \tag{2.15}$$

$$f(a) = c(a) \text{ for every } a \in \delta^{in}(U). \tag{2.16}$$

Proof. Let $f \in \mathcal{F}_{S,T}^{b,c,d}$ and $U \in \mathcal{C}_{S,T}$. We have

$$\begin{aligned} -\operatorname{excess}_f(U) &= -\sum_{u \in U} \operatorname{excess}_f(u) = -\sum_{u \in S} \operatorname{excess}_f(u) - \sum_{u \in U \setminus S} \operatorname{excess}_f(u) \\ &= -\operatorname{excess}_f(S) - \sum_{u \in U \setminus S} b(u) = \text{value}(f) - \sum_{u \in U \setminus S} b(u). \end{aligned}$$

Here the first and the last equalities are Lemma 2.2 (for U respectively for S), and the third one is Definition 2.3. Now this can be written as

$$\text{value}(f) = -\operatorname{excess}_f(U) + b(U \setminus S) = f(\delta^{\text{out}}(U)) - f(\delta^{\text{in}}(U)) + b(U \setminus S). \tag{2.17}$$

This expression is bounded from above by

$$\leq c(\delta^{\text{out}}(U)) - d(\delta^{\text{in}}(U)) + b(U \setminus S),$$

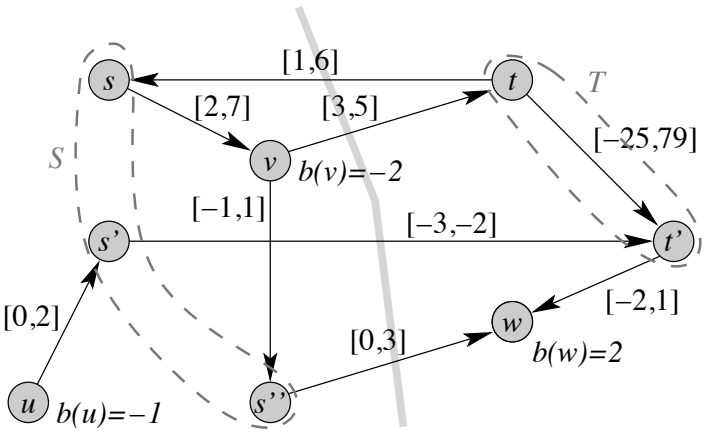

Figure 2.6: A minimal cut.

with equality holding if and only if

$$f(\delta^{\text{out}}(U)) = c(\delta^{\text{out}}(U)) \text{ and} \tag{2.18}$$

$$f(\delta^{\text{in}}(U)) = d(\delta^{\text{in}}(U)). \tag{2.19}$$

This equivalence follows from $f(\delta^{\text{out}}(U)) \leq c(\delta^{\text{out}}(U))$ and from $f(\delta^{\text{in}}(U)) \geq d(\delta^{\text{in}}(U))$. Similarly, the last expression in (2.17) is bounded from below by

$$\geq d(\delta^{\text{out}}(U)) - c(\delta^{\text{in}}(U)) + b(U \backslash S),$$

where in this case equality holds exactly if

$$f(\delta^{\text{out}}(U)) = d(\delta^{\text{out}}(U)) \text{ and} \tag{2.20}$$

$$f(\delta^{\text{in}}(U)) = c(\delta^{\text{in}}(U)). \tag{2.21}$$

To see that (2.18)–(2.21) are equivalent with (2.13)–(2.16), just remember that for all $a \in A$, we have $d(a) \leq f(a) \leq c(a)$. □

2.21 Example. Let us see if we can verify the statement of this theorem for the network $(D = (V, A), S, T)$ that we already examined in Examples 2.5, 2.7, and 2.16. In Example 2.7, we have constructed (S, T, b, c, d)-flows f_{max} and f_{min}, satisfying

$$\text{value}(f_{max}) = 2 \text{ and } \text{value}(f_{min}) = -8. \tag{2.22}$$

In Example 2.16, we had a look at the $S - T$-cut U, with

$$\text{capacity}(U) = 9 \text{ and } \text{cocapacity}(U) = -9.$$

Reassuringly, these values are in agreement with the predictions of Theorem 2.20. Let us see if we can improve the values of capacity(U) and cocapacity(U). Note that there are not too many choices: any $S-T$-cut X must satisfy s, s', $s'' \in X$ and t, $t' \notin X$. Hence, we can only decide whether u, v, and w should be in the cut or in its compliment. This gives a total of 8 different $S-T$-cuts in D: $|\mathcal{C}_{S,T}(D)| = 8$.

First, we try to find a cut of minimum capacity. Then u should be in the cut: $b(u) = -1$ lessens the capacity while $-d((u,s')) = 0$ would add nothing to it (if we left u outside the cut). Next, including w would add the term $b(w) = 2$ to the capacity, as well as $-d((t',w)) = 2$, adding to 4. Hence, it is better to leave w outside the cut, which amounts to only having the term $c((s'',w)) = 3$ in the calculation of the cut's capacity. For v, if we leave it outside, the following terms arise: $c((s,v)) = 7$, and $-d((v,s'')) = 1$, which makes 8 altogether. Clearly, including v and hence having $b(v) = -2$ and $c((v,t)) = 5$ in the sum, yields a cut of much smaller capacity. We can thus define $U_{min} \coloneqq \{s, s', s'', u, v\}$. This cut is depicted in Figure 2.6 on page 62.

Second, we try to find a cut of maximum cocapacity. For this purpose, it is again better to include u in the cut. Although this gives rise to the term $b(u) = -1$, this is still better than $-c((u,s')) = -2$, which we would have otherwise. How about w? If it is in, we have $b(w) = 2$ and $-c((t',w)) = -1$, adding to 1. If it is out, $d((s'',w)) = 0$ is all there is, so w is in. Finally v: leaving it outside the cut, v leads to the terms $d((s,v)) = 2$ and $-c((v,s'')) = -1$. Having it in the cut, the terms $b(v) = -2$ and $d((v,t)) = 3$ arise. This is a draw: we can choose v to be in the cut or in its complement, both choices will yield a cut of maximum cocapacity. We hence set $U^{co}_{max} \coloneqq \{s, s', s'', u, v, w\}$.

Let us calculate the capacity and the cocapacity of these two cuts:

$$\begin{aligned}
\text{capacity}(U_{min}) &= c(\delta^{\text{out}}(U_{min})) - d(\delta^{\text{in}}(U_{min})) + b(U_{min} \backslash S) \\
&= c(\{(s',t'),(s'',w),(v,t)\}) - d(\{(t,s)\}) + b(\{u,v\}) \\
&= ((-2)+3+5) - (1) + ((-1)+(-2)) = 2; \\
\text{cocapacity}(U^{co}_{max}) &= d(\delta^{\text{out}}(U^{co}_{max})) - c(\delta^{\text{in}}(U^{co}_{max})) + b(U^{co}_{max} \backslash S) \\
&= d(\{(s',t'),(v,t)\}) - c(\{(t,s),(t',w)\}) + b(\{u,v,w\}) \\
&= ((-3)+3) - (6+1) + ((-1)+(-2)+2) = -8.
\end{aligned}$$

Comparing this with (2.22), it can be inferred from Theorem 2.20 that our considerations were correct; it cannot be hoped to find cuts of smaller capacity or greater cocapacity.

2.22 Definition (Residual Graph)**.** Let $D = (V, A)$ be a digraph and let (S, T, b, c, d) be a flow-quintuple for D. Let $f\colon A \to \mathbb{R}$ be any function.

Then two useful subsets of A, and two of $A \cup A^{-1}$, are defined as follows:

$$A^{f<c} := \{a \in A \mid f(a) < c\} \tag{2.23}$$
$$A^{f>d} := \{a \in A \mid f(a) > d\} \tag{2.24}$$
$$A_f := A^{f<c} \cup (A^{f>d})^{-1} \tag{2.25}$$
$$A_f^{co} := (A^{f<c})^{-1} \cup A^{f>d} \tag{2.26}$$

The *residual graph* of f is $D_f := (V, A_f)$. The *co-residual graph* of f is $D_f^{co} := (V, A_f^{co})$. So D_f and D_f^{co} are digraphs, in fact subgraphs of the digraph $(V, A \cup A^{-1})$. Obviously, D_f and D_f^{co} do not only depend on D and f, but on c and d as well. However, I will only use them in contexts where D, c, and d stay fixed while f is varying, such that no ambiguity should arise. A directed $S - (T \backslash S)$-path in D_f is called a *flow-augmenting path*, for reasons that will become apparent soon. Note that in the case $S \cap T = \emptyset$, we have $\mathcal{P}_{S,(T\backslash S)} = \mathcal{P}_{S,T}$. A directed $S-(T\backslash S)$-path in D_f^{co} is called a *flow-diminishing path*, for reasons that will become apparent in Section 2.5.

2.23 Proposition. *Let $D = (V, A)$ be a digraph, and let (S, T, b, c, d) be a flow-quintuple for D. Assume*

$$\mathcal{F}_{S,T}^{b,c,d}(D) \neq \emptyset,$$
$$\mathcal{C}_{S,T}(D) \neq \emptyset.$$

Let f be an (S, T, b, c, d)-flow. Suppose there is no $S-T$-dipath in the residual graph D_f. Define $U \subseteq V$ to be the set of all vertices reachable from S in D_f (i.e., the set of all vertices $u \in V$ such that there is an $S - u$-dipath in D_f). Then

$$\text{value}(f) = \text{capacity}(U).$$

In particular, it follows from Theorem 2.20 that f is maximal.

Proof. First note that $U \supseteq S$ and $U \cap T = \emptyset$, hence U is an $S - T$-cut. Let $a \in \delta_D^{\text{out}}(U)$, $a = (u, v)$. So $u \in U$, $v \notin U$. We have $a \notin A_f$, because otherwise, since $u \in U$ (i.e., reachable from S in D_f), v would be reachable from S in D_f, too (i.e., $v \in U$), a contradiction. It follows from (2.23) and (2.25) that $f(a) = c(a)$ for every $a \in \delta_D^{\text{out}}(U)$.

Now let $a \in \delta_D^{\text{in}}(U)$, $a = (u, v)$. This means $u \notin U$ and $v \in U$. Again, $a^{-1} \in A_f$ would imply the reachability of u from S in D_f, so it can be concluded that $a^{-1} \notin A_f$. From (2.24) and (2.25) it follows that $f(a) = d(a)$ for all $a \in \delta_D^{\text{in}}(U)$.

We thus have seen that conditions (2.13) and (2.14), guaranteeing equality in Theorem 2.20, hold in this situation and the proof is finished. □

The following proposition is a little technical. Although being not very difficult, its proof still is a little lengthy, due to the necessity to tediously distinguish between several cases. Fortunately, this is about the only point in the deduction of the Max-Flow Min-Cut Theorem where it is necessary to resort to this kind of technicality.

2.24 Proposition (Augmentation of Flows)**.** *Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. Let $f \in \mathcal{F}_{S,T}^{b,c,d}(D)$. Suppose there is an $S-(T\backslash S)$-path P in D_f. Then there exists an $f' \in \mathcal{F}_{S,T}^{b,c,d}(D)$ of greater value than f. If all of f, c, and d are integral, then f' can be chosen integral as well.*

Proof. Let $P = (v_0, a_1, v_1, \ldots, a_k, v_k)$ be a path with $v_0 \in S$, $v_k \in (T\backslash S)$. Write $s := v_0$ and $t := v_k$. Also, set

$$\begin{aligned}\varepsilon &:= \min\big\{c - f(a) \mid a \in A^{f<c} \cap A(P)\big\} \\ &\qquad \cup \big\{f(a) - d(a) \mid a^{-1} \in (A^{f>d})^{-1} \cap A(P)\big\} > 0, \\ f' &:= f + \varepsilon\chi^P.^3\end{aligned}$$

In case the above value for ε was $= \infty$, just define ε as any real number > 0, e.g., $\varepsilon := 1$. f' still satisfies demand- and capacity-conditions (2.5) and (2.4) by construction. To see that f' also satisfies the balance-condition (2.3) for all $v \in R$, i.e., f' is an (S, T, b, c, d)-flow, some cases have to be examined.

For any vertex $v \notin V(P)$, and all arcs a adjacent to v, we have $f'(a) = f(a)$. This means that the left-hand side of (2.3) for v remains term-wise the same, if f is replaced by f'. In particular, f' satisfies (2.3) for v.

Now let $v \in V(P) \cap R$. The validity of (2.3) for v needs to be verified. Let for example $v = v_i$. For all arcs $a \in A$ adjacent to v in D with $a, a^{-1} \notin A(P)$, we have again $f'(a) = f(a)$. Since P is a path and $v \neq s, t$, there are exactly two arcs $a \in A(D)$ adjacent to v that satisfy one of $a \in A(P)$ or $a^{-1} \in A(P)$. One of them, let us call it x, is either a_i or a_i^{-1}, the other one, y, is either a_{i+1} or a_{i+1}^{-1}. Thus far, it can be concluded that

$$\begin{aligned}\operatorname{excess}_{f'}(v) &= \sum_{a \in \delta_D^{\text{in}}(v)} f'(a) - \sum_{a \in \delta_D^{\text{out}}(v)} f'(a) \\ &= \Big(\sum_{a \in \delta_D^{\text{in}}(v)\backslash\{x,y\}} f(a) - \sum_{a \in \delta_D^{\text{out}}(v)\backslash\{x,y\}} f(a)\Big) \pm f'(x) \pm f'(y), \qquad (2.27)\end{aligned}$$

where the signs of the last two terms depend on whether $x \in \delta_D^{\text{in}}(v)$ or $x \in \delta_D^{\text{out}}(v)$, and on whether $y \in \delta_D^{\text{in}}(v)$ or $y \in \delta_D^{\text{out}}(v)$.

[3] The definition of χ^P was given on page 23 in Section 1.3.2.

- $x = a_i$, $y = a_{i+1}$. Then $x \in \delta_D^{\text{in}}(v)$, $y \in \delta_D^{\text{out}}(v)$, which means the signs in (2.27) should be $+$, $-$ respectively. Also, $f'(x) = (f + \varepsilon\chi^P)(x) = f(x) + \varepsilon$. Similarly $f'(y) = f(y) + \varepsilon$. Therefore, with (2.27), it follows that

$$\begin{aligned}\text{excess}_{f'}(v) &= \sum_{a \in \delta_D^{\text{in}}(v) \setminus \{x\}} f(a) - \sum_{a \in \delta_D^{\text{out}}(v) \setminus \{y\}} f(a) + (f(x) + \varepsilon) - (f(y) + \varepsilon) \\ &= f(\delta_D^{\text{in}}(v)) - f(\delta_D^{\text{out}}(v)) = \text{excess}_f(v) = b(v),\end{aligned}$$

  so (2.3) for v is valid for f'.

- $x = a_i$, $y = a_{i+1}^{-1}$. Then x, $y \in \delta_D^{\text{in}}(v)$, and the signs in (2.27) are both $+$. We have $f'(x) = (f + \varepsilon\chi^P)(x) = f(x) + \varepsilon$, whereas $f'(y) = (f + \varepsilon\chi^P)(y) = f(y) - \varepsilon$. So in this case, (2.27) yields

$$\begin{aligned}\text{excess}_{f'}(v) &= \sum_{a \in \delta_D^{\text{in}}(v) \setminus \{x,y\}} f(a) - \sum_{a \in \delta_D^{\text{out}}(v)} f(a) + (f(x) + \varepsilon) + (f(y) - \varepsilon) \\ &= f(\delta_D^{\text{in}}(v)) - f(\delta_D^{\text{out}}(v)) = \text{excess}_f(v) = b(v),\end{aligned}$$

  This means again, (2.3) for v is satisfied by f'.

- $x = a_i^{-1}$, $y = a_{i+1}$. Now x, $y \in \delta_D^{\text{out}}(v)$, the signs in (2.27) both being negative. $f'(x) = f(x) - \varepsilon$ and $f'(y) = f(y) + \varepsilon$. The calculation hence reads

$$\begin{aligned}\text{excess}_{f'}(v) &= \sum_{a \in \delta_D^{\text{in}}(v)} f(a) - \sum_{a \in \delta_D^{\text{out}}(v) \setminus \{x,y\}} f(a) - (f(x) - \varepsilon) - (f(y) + \varepsilon) \\ &= f(\delta_D^{\text{in}}(v)) - f(\delta_D^{\text{out}}(v)) = \text{excess}_f(v) = b(v),\end{aligned}$$

  Again, (2.3) for v remains valid for f'.

- $x = a_i^{-1}$, $y = a_{i+1}^{-1}$. Then $x \in \delta_D^{\text{out}}(v)$, $y \in \delta_D^{\text{in}}(v)$, and the signs in (2.27) this time are $-$, $+$ respectively. Also, $f'(x) = f(x) - \varepsilon$, $f'(y) = f(y) - \varepsilon$.

$$\begin{aligned}\text{excess}_{f'}(v) &= \sum_{a \in \delta_D^{\text{in}}(v) \setminus \{y\}} f(a) - \sum_{a \in \delta_D^{\text{out}}(v) \setminus \{x\}} f(a) - (f(x) - \varepsilon) + (f(y) - \varepsilon) \\ &= f(\delta_D^{\text{in}}(v)) - f(\delta_D^{\text{out}}(v)) = \text{excess}_f(v) = b(v).\end{aligned}$$

  Hence (2.3) for v holds for f'.

Thus it is established that f' satisfies (2.3) for every choice of $v \in R$, i.e., f' is again an (S, T, b, c, d)-flow for D. Let us now calculate value(f') =

$f'(\delta_D^{\text{out}}(S)) - f'(\delta_D^{\text{in}}(S))$. For all arcs $a \in \delta_D^{\text{out}}(S) \cup \delta_D^{\text{in}}(S)$ which are not traversed by P, we have $f'(a) = f(a) + \chi^P(a) = f(a)$. This means that in the following sum, there is no term involving a:

$$\begin{aligned} &\text{value}(f') - \text{value}(f) = \\ &\text{value}(f' - f) = (f' - f)(\delta_D^{\text{out}}(S)) - (f' - f)(\delta_D^{\text{in}}(S)). \end{aligned} \tag{2.28}$$

So let the edge $a \in \delta_D^{\text{out}}(S) \cup \delta_D^{\text{in}}(S)$ be traversed by P. Analogously to the above, there are four cases.

- Let P be leaving S in D_f via a or a^{-1}.
  - If $\chi^P(a)$=1, then $a \in \delta_D^{\text{out}}(S)$. Therefore, $f'(a) = f(a) + \varepsilon\chi^P(a) = f(a) + \varepsilon$.
  - Otherwise, $\chi^P(a) = -1$. Then $a \in \delta_D^{\text{in}}(S)$ and $f'(a) = f(a) + \varepsilon\chi^P(a) = f(a) - \varepsilon$.

  In either case, it is hence clear that for each time P leaves S, there is a corresponding term $+\varepsilon$ in the sum (2.28).

- Now suppose P is entering S in D_f via a or a^{-1}.
  - If $\chi^P(a) = 1$, then $a \in \delta_D^{\text{in}}(S)$ and $f'(a) = f(a) + \varepsilon$.
  - If $\chi^P(a) = -1$, then $a \in \delta_D^{\text{out}}(S)$, as well as $f'(a) = f(a) - \varepsilon$.

  From these two cases it can be inferred that for each time P leaves S, there is a corresponding term $-\varepsilon$ in the sum (2.28).

Now, P starts in S and terminates outside S. So P has to leave S at some stage. In fact, P has to leave S exactly once more than P enters S. Therefore, (2.28) can be calculated exactly: $\text{value}(f') - \text{value}(f) = \varepsilon$, in particular $\text{value}(f') > \text{value}(f)$, which concludes the proof of the first part of this theorem.

To see that its last sentence is true, too, observe that if f, c, and d are all integer-valued functions, then ε as defined above is also an integer. Consequently, f' is integral as well. □

2.25 Remark. As mentioned before, the path P of Proposition 2.24 is called a *flow-augmenting path*. The technique used in the proof is referred to as *augmenting f along P*. It was invented by Ford and Fulkerson. Once any feasible flow is known, this idea of flow-augmentation can be used algorithmically to construct a flow of maximum value, as described in the next section.

2.26 Theorem. *Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. Assume*

$$\mathcal{F}_{S,T}^{b,c,d}(D) \neq \emptyset,$$
$$\mathcal{C}_{S,T}(D) \neq \emptyset.$$

Let $f \in \mathcal{F}_{S,T}^{b,c,d}(D)$. Then f is maximal if and only if there is no flow-augmenting path, i.e., if and only if D_f contains no directed $S-T$-path.

Proof. If there is a flow-augmenting path, apply Proposition 2.24 to see that f is not of maximal value.[4] If there is no flow-augmenting path, apply Proposition 2.23 to see that f is maximal. □

Now everything is prepared to summarize all of the pieces gathered so far in one neatly formulated theorem about the relation between flows and cuts. The second part of the theorem, about integer flows, is sometimes called "Integrality Theorem". Both the Max-Flow Min-Cut Theorem and the Integrality Theorem were first proved by Ford and Fulkerson.

2.27 Theorem (Maximum-Flow Minimum-Cut)**.** *Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. Suppose that*

$$\mathcal{F}_{S,T}^{b,c,d}(D) \neq \emptyset, \tag{2.29}$$
$$\mathcal{C}_{S,T}(D) \neq \emptyset, \tag{2.30}$$
$$c < \infty, \tag{2.31}$$
$$d > -\infty. \tag{2.32}$$

Then

$$\max_{f \in \mathcal{F}_{S,T}^{b,c,d}(D)} \text{value}(f) = \min_{U \in \mathcal{C}_{S,T}(D)} \text{capacity}(U). \tag{2.33}$$

If, in addition, c and d are integral, and if there is an integer (S, T, b, c, d)-flow, then there exists an integer maximum flow, i.e., a flow g such that $g(A) \subseteq \mathbb{Z}$ and $\text{value}(g) = \max_{f \in \mathcal{F}_{S,T}^{b,c,d}(D)} \text{value}(f)$.

Proof. By (2.29) and Lemma 2.13, there exists an (S, T, b, c, d)-flow f of maximum value. For this f, by Theorem 2.26, there is no flow-augmenting path. By Proposition 2.23, $\geq$ in (2.33) is correct. By Theorem 2.20, we also have $\leq$.

If there is an integer (S, T, b, c, d)-flow f, it can be augmented until no $S-T$-path in D exists. This is, because it can be inferred from the construction

[4] Note that in the case of $\mathcal{C}_{S,T}(D) \neq \emptyset$, every $S-T$-path is in fact an $S-(T \backslash S)$-path, because then $S \cup T = \emptyset$.

in the proof of Proposition 2.24 that in each iteration the value of f can be augmented by at least 1, and because (2.31) and (2.32) are assumed to be true. Moreover, this can be done in such a way that the finally resulting flow g is integral as well (cf. Proposition 2.24). Then the rest of the statement again can be deduced from Proposition 2.23 and Theorem 2.20. □

In Theorem 2.33 in Section 2.6, it will be shown that in order to guarantee existence of an integer maximal flow, it suffices to require b, c, and d to be integral, as then an integer flow exists (if any flow exists).

2.28 Theorem (Maximum-Flow Minimum-Cut, Classical Version)**.** *Let $D = (V, A)$ be a digraph, $s \neq t \in V$. Let furthermore be $c\colon A \to [0, \infty)$ a capacity function. Then the maximum value of an $s-t$-flow $f\colon A \to [0, \infty)$ subject to c equals the minimum capacity of an $s-t$-cut U. If c is integral, there exists an integer flow of maximum value.*

Proof. In Theorem 2.27, take $S := \{s\}$, $T := \{t\}$, $b := \mathbf{0}_R$, and $d := \mathbf{0}_A$. Then the non-emptiness conditions are satisfied since $f := \mathbf{0}_A \in \mathcal{F}_{S,T}^{b,c,d}(D)$ and $\{s\} \in \mathcal{C}_{S,T}(D)$.

The statement about integrality follows from the fact that $d = \mathbf{0}_A$ and $f := \mathbf{0}_A$ are both integral. □

2.5 The Min-Flow Max-Cocapacity Theorem

Analogously to the theory developed in the last section, one can examine minimal (S, T, b, c, d)-flows. I will give an outline of the things that change when reversing the game. As hinted at implicitly in Theorem 2.20, it turns out that in this case, the quantity that gets optimized dually instead of cut-capacities is nicely described as cut-cocapacity, as introduced in Definition 2.15.

In Chapter 4, we will see that the Max-Flow Problem (respectively the Min-Flow Problem), that is, finding a maximum flow (respectively a minimum flow), can be solved by algorithms with a polynomial running time, i.e., these problems are in the class P. Interestingly, the Max-Cut Problem, i.e., finding a cut (or an $S-T$-cut) of maximal capacity in a digraph, is NP-complete (see [9, p. 249f.] or [13, p. 479]).

2.29 Proposition. *Let $D = (V, A)$ be a digraph, and let (S, T, b, c, d) be a flow-quintuple for D. Let $f \in \mathcal{F}_{S,T}^{b,c,d}(D)$. Suppose there is no $S-T$-dipath in D_f^{co}. Define $U \subseteq V$ to be the set of all vertices reachable from S in D_f^{co} (i.e., the set of all vertices $u \in V$ such that there is an $S-u$-dipath in D_f^{co}). Then*

$$\text{value}(f) = \text{cocapacity}(U).$$

In particular, it follows from Theorem 2.20 that f is minimal.

Proof. Completely analogous to the proof of Proposition 2.23. this time (towards the end of the proof) exploiting the other half of Theorem 2.20, i.e., (2.12). □

2.30 Proposition (Diminution of Flows)**.** *Let $D = (V, A)$ be a digraph. Let (S, T, b, c, d) be a flow-quintuple for D and $f \in \mathcal{F}^{b,c,d}_{S,T}(D)$. Suppose there is an $S - (T\backslash S)$-dipath P in D^{co}_f. Then there exists $f' \in \mathcal{F}^{b,c,d}_{S,T}(D)$ of smaller value than f. If, in addition, f, c, and d are integral, then f' can be chosen integral.*

Proof. Let $P = (s = v_0, a_1, v_1, \ldots, a_k, v_k = t)$ be a path with $s \in S$, $t \in (T\backslash S)$. Set

$$\varepsilon \coloneqq \min \big\{ c - f(a) \mid a^{-1} \in (A^{f<c})^{-1} \cap A(P) \big\} \\ \cup \big\{ f(a) - d(a) \mid a \in A^{f>d} \cap A(P) \big\} > 0.$$

The feasible flow of smaller value than f that we are looking for is

$$f' \coloneqq f - \varepsilon \chi^P.$$

Verification of feasibility and the of the fact that $\text{value}(f') < \text{value}(f)$ is completely analogous to the proof of Proposition 2.24. The same holds true for the statement about integrality. □

2.31 Theorem. *Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. Assume*

$$\mathcal{F}^{b,c,d}_{S,T}(D) \neq \emptyset,$$
$$\mathcal{C}_{S,T}(D) \neq \emptyset.$$

Let $f \in \mathcal{F}^{b,c,d}_{S,T}$. Then value$(f)$ *is minimal if and only if there is no flow-diminishing path, i.e., if and only if D^{co}_f contains no directed $S - T$-path.*

Proof. If there is a flow-diminishing path, apply Proposition 2.30 to see that f is not of minimal value. If there is no flow-diminishing path, apply Proposition 2.29 to see that f is minimal. □

2.32 Theorem (Minumum-Flow Maximum-Cocapacity)**.** *Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. Suppose that*

$$\mathcal{F}^{b,c,d}_{S,T}(D) \neq \emptyset, \tag{2.34}$$
$$\mathcal{C}_{S,T}(D) \neq \emptyset, \tag{2.35}$$
$$c < \infty, \tag{2.36}$$
$$d > -\infty. \tag{2.37}$$

Then

$$\min_{f\in\mathcal{F}_{S,T}^{b,c,d}(D)} \text{value}(f) = \max_{U\in\mathcal{C}_{S,T}(D)} \text{cocapacity}(U). \tag{2.38}$$

If, in addition c and d are integral, and if there exists any integer flow, then there is an integer flow of minimal value.

Proof. By (2.34) and Lemma 2.13, there exists an (S,T,b,c,d)-flow f of minimum value. For this f, by Theorem 2.31, there is no flow-diminishing path. By Proposition 2.29, $\leq$ in (2.38) is valid. By Theorem 2.20, the same is true for $\geq$.

The statement about integrality is proved completely analogous to the corresponding statement about maximal flows in the proof of Theorem 2.27. □

2.6 Existence of Feasible Flows

As I see it, the main difference between the above presented approach and the one using a narrower definition of networks and flows, is when the question of existence of flows has to be discussed (and this also certainly is a quantitative rather than a qualitative difference). In the standard case with

$$\begin{aligned} S =& \{s\},\\ T =& \{t\},\\ b =& \mathbf{0}_R,\\ c \geq& \mathbf{0}_A,\\ d =& \mathbf{0}_A, \end{aligned}$$

the question of existence does not arise at all, since always $\mathbf{0}_A \in \mathcal{F}_{S,T}^{b,c,d}$. Hence, the existence of feasible flows usually is discussed in the context of circulations (i.e., $S = T = \emptyset$) satisfying certain upper and lower bounds. With the definition given in Section 2.2, it is in general never clear whether or not $\mathcal{F}_{S,T}^{b,c,d} \neq \emptyset$, and if yes, how to find a feasible flow. This will be of particular importance for the algorithms constructing maximal flows, discussed in Chapter 4.

The main existence result stems from Hoffmann, proved in 1956, and is well-known under the name "Hoffmann's Circulation Theorem". I formulate it such that it fits Definitions 2.3. The proof is adapted from [21, p. 171f.].

2.33 Theorem. *Let $D = (V,A)$ be a digraph and (S,T,b,c,d) a flow-quintuple for D. Suppose $A \neq \emptyset$. Then we have $\mathcal{F}_{S,T}^{b,c,d}(D) \neq \emptyset$ if and only if*

the following conditions hold:

$$d \leq c \tag{2.39}$$
$$c > -\infty \tag{2.40}$$
$$d < \infty \tag{2.41}$$
$$\text{capacity}(U) \geq 0 \quad \textit{for all } U \subseteq R \tag{2.42}$$
$$\text{cocapacity}(U) \leq 0 \quad \textit{for all } U \subseteq R \tag{2.43}$$

In the trivial case $A = \emptyset$, we have

$$\mathcal{F}_{S,T}^{b,c,d} \neq \emptyset \Longleftrightarrow b = \mathbf{0}_R.$$

If, in addition, b, c, and d are integral, then there exists an integer flow.

Proof. Let $A \neq \emptyset$. The necessity of (2.39) was already mentioned and is quite obvious. Also obviously necessary for the existence of a function $f\colon A \to \mathbb{R}$ satisfying $d \leq f \leq c$ are (2.40) and (2.41). Now let $U \subseteq R$. Then

$$\text{capacity}(U) = c(\delta^{\text{out}}(U)) - d(\delta^{\text{in}}(U)) + \sum_{u \in U} b(u), \quad \text{and}$$
$$\text{cocapacity}(U) = d(\delta^{\text{out}}(U)) - c(\delta^{\text{in}}(U)) + \sum_{u \in U} b(u).$$

To see the necessity of (2.42) and (2.43), let $f \in \mathcal{F}_{S,T}^{b,c,d}$. Then

$$\begin{aligned}\sum_{u \in U} b(u) &= \sum_{u \in U} \text{excess}_f(u) = \text{excess}_f(U) = f(\delta^{\text{in}}(U)) - f(\delta^{\text{out}}(U)) \\ &\geq d(\delta^{\text{in}}(U)) - c(\delta^{\text{out}}(U)).\end{aligned} \tag{2.44}$$

Here the first equality is the balance-condition (2.3), the third is Lemma 2.2, and the final inequality is Conditions (2.4) and (2.5). Since $U \subseteq R$ was chosen without restriction, it is plain to see that (2.44) is equivalent with (2.42). Likewise obtained is the following equation, which is equivalent to (2.43):

$$\sum_{u \in U} b(u) \leq c(\delta^{\text{in}}(U)) - d(\delta^{\text{out}}(U)).$$

Conversely, assume (2.39)–(2.43). Choose a function $f\colon A \to \mathbb{R}$ with $d \leq f \leq c$ (this can be achieved due to (2.39)–(2.41)). This function can be chosen such that it minimizes the expression

$$\| \text{excess}_f - b \|_1{}^5 = \sum_{v \in R} | \text{excess}_f(v) - b(v) |. \tag{2.45}$$

To see this, observe that the set of functions $f\colon A \to \mathbb{R}$ with $d \leq f \leq c$ is a nonempty, compact subset of the metric space $\mathbb{R}^A$. As the function defined in (2.45) has that set as its domain, and as it is the composite of continuous function (hence it is itself continuous), Theorem 1.18 guarantees the existence of a minimum. Now set

$$R_S \coloneqq \{v \in R \mid \operatorname{excess}_f(v) - b(v) > 0\},$$
$$R_T \coloneqq \{v \in R \mid \operatorname{excess}_f(v) - b(v) < 0\}.$$

Let $D_f = (V, A_f)$ be the residual graph. If D_f contained an $R_S - R_T$-, an $R_S - S$-, an $R_S - T$-, an $S - R_T$-, or an $T - R_T$-path, then this path could be used to reduce $\|\operatorname{excess}_f - b\|_1$ without violating $d \leq f \leq c$, and while maintaining $\operatorname{excess}_f(v) = b(v)$ for $v \in R\backslash(R_S \cup R_T)$. This could be done completely similarly to the flow-augmenting technique used in the proof of Proposition 2.24. (Recall that for vertices in $S \cup T$ Condition (2.3) does not have to be satisfied. Therefore arbitrary amounts of excess flow could be transported from R_S to that set, or from that set to R_T.) Hence, there are no such paths in D_f.

Let $U \subseteq V$, respectively $W \subseteq V$, be the set of vertices reachable from R_S in D_f, respectively the set of vertices from which R_T is reachable in D_f. For every $a \in \delta_D^{\text{out}}(U)$ we have $a \notin A_f$ (otherwise the endpoint of a would also be reachable from R_S in D_f), and hence $f(a) = c(a)$. Similarly, the remaining three of the following statements are derived:

$$\begin{aligned} a \in \delta_D^{\text{out}}(U) &\implies f(a) = c(a), \\ a \in \delta_D^{\text{in}}(U) &\implies f(a) = d(a), \\ a \in \delta_D^{\text{in}}(W) &\implies f(a) = c(a), \text{ and} \\ a \in \delta_D^{\text{out}}(W) &\implies f(a) = d(a). \end{aligned}$$

Since $U \cap R_T = \emptyset$, and since $\operatorname{excess}_f(v) - b(v) = 0$ for every $v \in R\backslash(R_S \cup R_T)$, the capacity of U can now be calculated:

$$\begin{aligned} \operatorname{capacity}(U) &= c(\delta^{\text{out}}(U)) - d(\delta^{\text{in}}(U)) + b(U) \\ &= f(\delta^{\text{out}}(U)) - f(\delta^{\text{in}}(U)) + b(U) = -(\operatorname{excess}_f(U) - b(U)) \\ &= -(\operatorname{excess}_f(R_S) - b(R_S)) = -\Big(\sum_{v \in R_S} \operatorname{excess}_f(v) - b(v)\Big). \end{aligned}$$

Due to the definition of R_S, the last expression would be < 0 for $R_S \neq \emptyset$ (contradicting (2.42)). Hence, necessarily $R_S = \emptyset$ holds. A similar argument

[5]The norm $\|.\|_1$ was defined on page 26 in Section 1.4.1.

leads to $R_T = \emptyset$: we have $W \cap R_S = \emptyset$. Therefore,

$$\begin{aligned}\operatorname{cocapacity}(W) &= d(\delta^{\mathrm{out}}(W)) - c(\delta^{\mathrm{in}}(W)) + b(W)\\ &= f(\delta^{\mathrm{out}}(W)) - f(\delta^{\mathrm{in}}(W)) + b(W) = -(\operatorname{excess}_f(W) - b(W))\\ &= -(\operatorname{excess}_f(R_T) - b(R_T)) = -\Big(\sum_{v\in R_T} \operatorname{excess}_f(v) - b(v)\Big).\end{aligned}$$

The last expression would be > 0 for $R_T \neq \emptyset$, contradicting (2.43). Consequently $R_T = \emptyset$. This shows $\operatorname{excess}_f(v) = b(v)$ for all $v \in R$, which means that $f \in \mathcal{F}_{S,T}^{b,c,d}$, as desired.

Finally for $A = \emptyset$, $d \leq f$ and $f \leq c$ are vacuously true. Consequently, the only flow $f = \emptyset$ is feasible exactly if $b = \mathbf{0}_R$.

In the case of integer-valued functions b, c, and d, the flow f with $d \leq f \leq c$ with which the construction in this proof was started, can be chosen integral. This property could be maintained in every iteration while applying the flow-augmentation technique of Proposition 2.24. In this way, the statement about existence of integer flows can also be seen to be true. □

2.34 Remark. If Theorem 2.33 is applied to circulations ($S = T = \emptyset$), then taking $U \coloneqq V$ $(= R)$, the set of all vertices, in (2.42) and (2.43) yields

$$\begin{aligned}0 \geq \operatorname{cocapacity}(V) &= d(\delta^{\mathrm{out}}(V)) - c(\delta^{\mathrm{out}}(V)) + \sum_{v\in V} b(v)\\ &= d(\emptyset) - c(\emptyset) + \sum_{v\in V} b(v) = \sum_{v\in V} b(v).\end{aligned}$$

Analogously,

$$0 \leq \sum_{v\in V} b(v).$$

This means that it is necessary, for $\mathcal{F}_{S,T}^{b,c,d}$ to be nonempty, that

$$\sum_{v\in V} b(v) = 0.$$

This can be interpreted as no flow being 'produced' or 'absorbed' anywhere in D. This is probably what you would expect anyway, when there are neither sources nor sinks.

2.35 Remark. Similarly to the algorithms described in Chapter 4, the last proof can be used to find a feasible flow, if there is any. For this purpose, one could start with any flow f satisfying $d \leq f \leq c$ (for example $f = d$), and

then repeatedly use paths in the residual graph to modify f along them, in order to reduce $\|\operatorname{excess}_f - b\|_1$. When there are no suitable paths anymore, then either the resulting flow is feasible, or $\mathcal{F}_{S,T}^{b,c,d} = \emptyset$ can be concluded.

Moreover, all of the more efficient algorithms for finding maximal flows described in Chapter 4 could be used instead (with the necessary notational alterations). Summarizing, trying to find a maximal (S, T, b, c, d)-flow basically amounts to a two-phase task, where in each phase a flow maximization problem is to be solved.

2.36 Remark. In Chapter 3, we will see that maximizing value(f) over the set of feasible flows and minimizing capacity(U) over the set of $S-T$-cuts is in fact a primal-dual pair of linear programs. Therefore, there is a direct correspondence between the cases enumerated in Remark 1.26, where the possibilities for a primal-dual pair regarding feasibility/finiteness of the optima are listed, and the facts about flows and cuts gathered so far:

- The case examined in Theorem 2.27, where $\mathcal{F}_{S,T}^{b,c,d} \neq \emptyset$ and $\mathcal{C}_{S,T} \neq \emptyset$, is the first case listed in Remark 1.26: the primal and the dual program both have finite optima.

- If $\mathcal{F}_{S,T}^{b,c,d} \neq \emptyset$ and the function value is unbounded, then at least one of the Conditions (2.7) or (2.8) from Lemma 2.13 must be violated; in either case no $S-T$-cut of finite capacity can exist. This corresponds to the second case of Remark 1.26.

- The case $\mathcal{F}_{S,T}^{b,c,d} = \emptyset$ is the third case of Remark 1.26: the dual program could be infeasible (i.e., $S \cap T \neq \emptyset$, as discussed in Remark 2.19), or the function capacity could be unbounded.

2.7 Extensions of the Network Model

In Remark 2.10, I discussed the pros and cons of the terminology for networks and flows that I introduced in Definition 2.3. I also stated that from a theoretical point of view, there are not many significant differences between the various definitions found in the literature. Nevertheless, for practical purposes, when trying to put together a network model of some real life situation, it might be useful to have a fairly flexible terminology at hand. In that way, it is possible to first construct a model in which important features of the original structure can be represented directly in a one-to-one fashion. Then, in a separate second step, one can reduce the network developed in that way to a more basic network model (of probably greater size though)

that might for example satisfy a certain input format for a specific algorithm. This appears likely to be an easier task than having to accomplish these two steps at once.

Consequently, I would briefly like to outline a few more general and hence more flexible network notions (in the above sense) than the one from Definition 2.3.

2.7.1 Node Capacities

A question that comes up naturally in several modelling applications is that of vertex capacities. For example, the vertices could represent antennas or control centres in a telephone network. Then, even if there are no capacity problems otherwise, clearly it will not be possible to process an unlimited number of calls through each of the antennas or control centres. When modelling the waste water system of a town, the vertices might represent storage tanks (having a limited capacity). Likewise, the vertices could be used to model airports in a traffic network, capable of providing service only for a certain number of takeoffs and landings.

The common technique to reduce a network with capacitated nodes to the case without (unlimited) capacities is to split every vertex v into two vertices v_{in} and v_{out} plus two new arcs (v_{in}, v_{out}) and (v_{out}, v_{in}). The capacity of v is assigned to these two arcs. Moreover, every arc (u, v) respectively (v, w) is replaced by an arc (u_{out}, v_{in}) respectively (v_{out}, w_{in}). This approach is for example described in [8, Section 1.11], [1, p. 41–43], or [21, p. 176].

In the case $b(v) \neq 0$, restrictions could of course either be put upon flow into v, flow out of v, or both. In the mentioned case with vertices representing airports and the flow representing incoming and outgoing flights, it might even be suitable to put a capacity bound upon the sum $\sum_{a \in \delta^{\mathrm{in}}(v) \cup \delta^{\mathrm{out}}(v)} |f(a)|$.

2.7.2 Upper and Lower Balance Bounds

This variant is taken from [21, Section 11.5]: instead of prescribing exact values for excess_f at the vertices, one could impose upper and lower bounds b_c, b_d. To be a little bit more exact, let $D = (V, A)$ be a digraph, $S, T \subseteq V$, and c respectively d a capacity respectively a demand function. Let furthermore be b_c, $b_d \colon R \to \overline{\mathbb{R}}$. Then a function $f \colon A \to \mathbb{R}$ is called an

(S, T, b_c, b_d, c, d)-*flow* if and only if

$$\begin{array}{lcl} \text{(i)} & f(a) \geq d(a) & \forall a \in A, \\ \text{(ii)} & f(a) \leq c(a) & \forall a \in A, \\ \text{(iii)} & \text{excess}_f(v) \geq b_d(v) & \forall v \in R, \\ \text{(iv)} & \text{excess}_f(v) \leq b_c(v) & \forall v \in R. \end{array}$$

For this situation, an existence result similar to the one formulated as Theorem 2.33 can be established (see for example [21, p. 175].

2.7.3 Traversal Times

In the majority of applications, it will require a certain amount of time for the 'flow' (whatever it represents) to traverse the arc (u, v). In other words, starting at u, a unit of flow will need a certain time $t(a)$ to reach v. Sometimes this time might be negligible or irrelevant. In other cases it might be the only matter of interest.

In a model respecting those traversal times, the flow capacity bounds are often thought of as per-unit time bounds. That is, in each of the discrete time states $T_1, T_2, \ldots$, a given arc can at most (or least) carry flow as specified by its assigned capacity bounds. 'Storing capacities' on the vertices is another variation: in this case, one can decide whether flow is sent immediately from one vertex to the next, or stored for a while, for later processing.

These topics are examined in [8, Section III.7–III.9], [22, Section 6.2], [7, p. 82–84], or [21, Section 12.5.c]. Basically it can be said that all of these 'dynamic flow' models can be reduced to a 'static' one by adding arcs and vertices.

2.7.4 'Lossy' and 'Gainy' Arcs

The situation becomes a little different if one desires to include 'lossy' or 'gainy' arcs into the model. These arcs could for example represent leakage in a system of pipelines or waterpipes, loss of quality during the transmission of a message, or well-invested money in an econometrical model.

Formally, this situation can be described by replacing balance-condition (2.3) by

$$\sum_{a \in \delta^{\text{in}}(v)} \mu(a) f(a) - \sum_{a \in \delta^{\text{out}}(v)} f(a) = b(v).$$

Here $\mu \in [0, \infty)^A$, the $\mu(a)$'s being certain 'multipliers', representing the degree of loss or gain in the respective arc.

In [1], this type of network is called a *generalized network*. For such a network, the Max-Flow Problem is significantly more difficult to solve (i.e., the solution takes longer) than in the classical (loss- and gainless) case. Among the fastest algorithms for it is the "Generalized Network Simplex Algorithm". This is a specialization of Dantzig's general Simplex Algorithm for generalized networks. It works similar to the Network Simplex Algorithm described in Section 4.4.1.

A detailed study can be found in [1, Chapter 15]. Other places where this topic is treated (under the name of "signal flows") are [24, Chapter 8], and [7, Chapter 3].

Chapter 3

A Linear Programming Approach to Maximal Flow

In this chapter, I am going to describe a rather different approach to the Max-Flow Min-Cut topic. Conditions (2.5), (2.4), and (2.3) of Definition 2.3 are defining a certain polyhedron in $P \subseteq \overline{\mathbb{R}}^n$ (for some $n \in \mathbb{N}$). The problem of finding a maximal flow in a network is the same as finding a vector $f \in P$ in the corresponding polyhedron that maximizes the objective function value. These findings are elaborated in Section 3.1.

Then, in Section 3.2, I am first going to discuss in detail the correspondence between feasible flows for a given flow-quintuple (S, T, b, c, d) in a given digraph D and feasible solutions of a certain linear program. This program will be regarded as the *primal* one.

The second step will be to show a similar correspondence between $S-T$-cuts in the same graph and certain feasible solutions of the associated dual program. The Max-Flow Min-Cut Theorem will then, after considering finiteness of the optima of the so constructed programs, follow as a consequence of the Duality Theorem of Linear Programming, which was stated in Section 1.4.5.

3.1 Flow and Cut Optimization are Linear Programs

Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. In order to construct a corresponding linear program, an objective function needs to be specified, as well as the constraints subject to which that function is to be optimized (i.e., which solutions are the feasible ones), and the underlying vector space.

In this section, the capacity bounds c and d, respectively flows f, should be thought of as vectors in $\overline{\mathbb{R}}^A$, respectively $\mathbb{R}$, and the balance-vector b as a vector in $\overline{\mathbb{R}}^{R}$[1]. Let us make the following definition:

$$u \coloneqq (u_a)_{a\in A} \quad \text{with} \quad u_a \coloneqq \begin{cases} 1 & \text{if } a \in \delta^{\text{out}}(S) \\ -1 & \text{if } a \in \delta^{\text{in}}(S) \\ 0 & \text{otherwise} \end{cases}.$$

Moreover, define $\Delta = (\delta_{v,a})_{(v,a)\in R\times A}$ to be the $R \times A$-incidence matrix, as defined in Section 1.3.2, on page 23. I_A is the identity matrix on A, defined in Section 1.4.1. The dimension of Δ is $|R| \times |A|$ and the dimension of I_A is $|A| \times |A|$. Therefore:

$$\begin{aligned} b &\in \overline{\mathbb{R}}^R, \\ c,\, d &\in \overline{\mathbb{R}}^A, \\ u &\in \{-1,0,1\}^A, \\ \Delta &\in M_{R,A}(\{-1,0,1\}), \\ I_A &\in M_{A,A}\{0,1\}. \end{aligned}$$

The linear program of our interest in the current chapter is the following:

$$\max_{f\in\mathbb{R}^A}\{u^{\mathrm{T}} f \mid d \le f \le c,\, \Delta f = b\}. \tag{3.1}$$

I will refer to a linear program in the form of (3.1) as a *network flow linear program* or as a linear program in *network flow form*. Observe that this is a special case of the form in which linear programs were introduced in Definition 1.21. It is obtained from (1.5) by setting

$$\begin{aligned} \dim(x) = \dim(z) &\coloneqq 0, \\ \dim(y) &\coloneqq |A|, \\ \alpha &\coloneqq c, \\ \beta &\coloneqq b, \\ \gamma &\coloneqq d, \\ \varepsilon &\coloneqq u, \\ B = H &\coloneqq I_A, \\ E &\coloneqq \Delta. \end{aligned}$$

[1] Remember that $R \coloneqq V\backslash(S \cup T)$.

If the general version of the Duality Theorem of Linear Programming, Theorem 1.28, is applied to (3.1), this yields the following equation:

$$\begin{aligned}&\max_{f\in\mathbb{R}^A}\{u^{\mathrm{T}}f \mid d \le f \le c,\, \Delta f = b\}\\ =&\min_{x,z\in\mathbb{R}^A,\, y\in\mathbb{R}^R}\{x^{\mathrm{T}}c + y^{\mathrm{T}}b + z^{\mathrm{T}}d \mid x \ge \mathbf{0}_A,\, z \le \mathbf{0}_A,\, x^{\mathrm{T}} + y^{\mathrm{T}}\Delta + z^{\mathrm{T}} = u^{\mathrm{T}}\}.\end{aligned}$$

Writing this slightly differently, and using the transpose $\Delta^{\mathrm{T}} = (\delta_{v,a})_{(a,v)\in A\times V}$ of the incidence matrix Δ, the following statement results. Remember that it holds true if and only if at least one of the optima is finite.

$$\max_{f\in\mathbb{R}^A}\{u^{\mathrm{T}}f \mid d \le f \le c,\, \Delta f = b\} \tag{3.2}$$

$$=\min_{x,y\in\mathbb{R}^A,\, z\in\mathbb{R}^R}\{c^{\mathrm{T}}x - d^{\mathrm{T}}y + b^{\mathrm{T}}z \mid x,\, y \ge \mathbf{0}_A,\, x - y + \Delta^{\mathrm{T}}z = u\}. \tag{3.3}$$

Now, what exactly is the connection between (3.2) and (S, T, b, c, d)-flows in D, and between (3.3) and $S-T$-cuts in D? This question is the topic of the next section. We will see that the last equation, (3.2)=(3.3), is actually exactly the Max-Flow Min-Cut Theorem.

Before I can deduce this fact, one more powerful concept from the duality theory of linear programming needs to be introduced: complementary slackness. This concept gives necessary and sufficient conditions for a pair of solutions to a primal-dual pair of linear programs to be optimal. I formulate it for a primal linear program in network flow form, i.e., for the Primal-Dual Pair (3.2) and (3.3).

3.1 Lemma (Complementary Slackness)**.** *Let f, (x, y, z) be a pair of solutions for the Linear Programs (3.2) and (3.3) respectively. They are optimal solutions if and only if for all $a \in A$ the following two conditions hold:*

$$x_a > 0 \Longrightarrow f_a = c_a, \tag{3.4}$$

$$y_a > 0 \Longrightarrow f_a = d_a. \tag{3.5}$$

Proof. Since $d \le f \le c$ and $x, y \ge 0$, Conditions (3.4) and (3.5) hold for all $a \in A$ if and only if the following equation holds for all $a \in A$:

$$x_a(c_a - f_a) + y_a(f_a - d_a) = 0. \tag{3.6}$$

The left-hand side of (3.6) is ≥ 0, therefore (3.6) holds for all $a \in A$ if and only if

$$\sum_{a\in A} x_a(c_a - f_a) + \sum_{a\in A} y_a(f_a - d_a) = 0,$$

which can also be written as

$$x^{\mathrm{T}}c - x^{\mathrm{T}}f + y^{\mathrm{T}}f - y^{\mathrm{T}}d = 0. \tag{3.7}$$

Let us compute the value of the objective function of the Program (3.2):

$$u^{\mathrm{T}}f = (x^{\mathrm{T}} - y^{\mathrm{T}} + z^{\mathrm{T}}\Delta)f = x^{\mathrm{T}}f - y^{\mathrm{T}}f + z^{\mathrm{T}}\Delta f = x^{\mathrm{T}}f - y^{\mathrm{T}}f + z^{\mathrm{T}}b.$$

The pair f, (x, y, z) is a pair of optimal solutions if and only if the values of their objective functions coincide, that is, if and only if

$$x^{\mathrm{T}}f - y^{\mathrm{T}}f + z^{\mathrm{T}}b = x^{\mathrm{T}}c - y^{\mathrm{T}}d + z^{\mathrm{T}}b. \tag{3.8}$$

Clearly, Equation (3.8) is equivalent with Equation (3.7). This completes the proof. □

3.2 The Max-Flow Min-Cut Theorem Revisited

I am now going to re-prove the Max-Flow Min-Cut Theorem. It is a special case of the Duality Theorem of Linear Programming.

3.2 Theorem (Maximum-Flow Minimum-Cut)**.** *Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. Suppose that*

$$\mathcal{F}_{S,T}^{b,c,d}(D) \neq \emptyset, \tag{3.9}$$
$$\mathcal{C}_{S,T}(D) \neq \emptyset, \tag{3.10}$$
$$c < \infty, \tag{3.11}$$
$$d > -\infty. \tag{3.12}$$

Then

$$\max_{f \in \mathcal{F}_{S,T}^{b,c,d}(D)} \mathrm{value}(f) = \min_{U \in \mathcal{C}_{S,T}(D)} \mathrm{capacity}(U). \tag{3.13}$$

Proof. Let $\Delta = (\delta_{v,a})_{(v,a) \in R \times A}$ and u be defined as in the last section. The first things that needs to be shown, is that

$$\max_{f \in \mathbb{R}^A} \{u^{\mathrm{T}}f \mid d \leq f \leq c,\, \Delta f = b\} = \max_{f \in \mathcal{F}_{S,T}^{b,c,d}(D)} \mathrm{value}(f). \tag{3.14}$$

But this becomes almost obvious after a quick look at the definitions of u and Δ: we have $u^{\mathrm{T}}f = \mathrm{value}(f)$, and

$$\Delta f = b \Longleftrightarrow \text{balance-condition (2.3) holds for all } v \in R.$$

The left-hand side of (3.14) is bounded as f lies between d and c, and because the objective function is linear. Therefore, the conditions of Theorem 1.28 are satisfied. As described at the end of the previous section, this yields

$$\max_{f\in\mathbb{R}^A}\{u^{\mathrm{T}}f \mid d\le f\le c,\ \Delta f = b\} \tag{3.15}$$

$$= \min_{x,y\in\mathbb{R}^A,\, z\in\mathbb{R}^R}\{c^{\mathrm{T}}x - d^{\mathrm{T}}y + b^{\mathrm{T}}z \mid x,\, y\ge \mathbf{0}_A,\, x - y + \Delta^{\mathrm{T}}z = u\}. \tag{3.16}$$

It remains to show, in order to re-prove the Max-Flow Min-Cut Theorem, that (3.16) equals

$$\min_{U\in\mathcal{C}_{S,T}(D)}\Big(c(\delta^{\mathrm{out}}(U)) - d(\delta^{\mathrm{in}}(U)) + \sum_{v\in U\setminus S} b(v)\Big). \tag{3.17}$$

First, I will show that (3.16) $\le$ (3.17). Choose an $S-T$-cut $U\subseteq V$. Remember it satisfies $U\supseteq S$ and $U\cap T=\emptyset$. Define

$$x := (x_a)_{a\in A} \quad \text{with} \quad x_a := \begin{cases} 1 & \text{if } a\in\delta^{\mathrm{out}}_D(U)\\ 0 & \text{otherwise}\end{cases},$$

$$y := (y_a)_{a\in A} \quad \text{with} \quad y_a := \begin{cases} 1 & \text{if } a\in\delta^{\mathrm{in}}_D(U)\\ 0 & \text{otherwise}\end{cases}, \text{ and}$$

$$z := (z_v)_{v\in R} \quad \text{with} \quad z_v := \begin{cases} 1 & \text{if } v\in U\\ 0 & \text{otherwise}\end{cases}.$$

Clearly, x and y are $\in\mathbb{R}^A$ and $\ge\mathbf{0}_A$, and $z\in\mathbb{R}^R$. Now choose $a=(v_1,v_2)\in A$. Then $\delta_{v_1,a}=1$, $\delta_{v_2,a}=-1$, and $\delta_{v,a}=0$ for all other vertices $v\in V$. To see that the vector $x-y+\Delta^{\mathrm{T}}z$ equals the vector u, the two can be compared coordinate-wisely. This amounts to the verification of

$$x_a - y_a + (\Delta^{\mathrm{T}}z)_a = u_a. \tag{3.18}$$

Here $(\Delta^{\mathrm{T}}z)_a = \sum_{v\in R}\delta_{v,a}z_v = \sum_{v\in R\cap\{v_1,v_2\}}\delta_{v,a}z_v$ is the a-th entry of the $|A|$-dimensional vector $(\Delta^{\mathrm{T}}z)$. It has to be distinguished between a few cases.

- Firstly, suppose $a\in\delta^{\mathrm{out}}(S)$, i.e., $v_1\in S\subseteq U$ (so $v_1\notin R$, consequently z_{v_1} is not defined), and $v_2\notin S$. Then $u_a=1$. Since $v_1\in U$, we have $a\notin\delta^{\mathrm{in}}(U)$, hence $y_a=0$.
  - If $a\in\delta^{\mathrm{out}}(U)$, then $x_a=1$ and $v_2\notin U\setminus S$. Hence z_{v_2} is either $=0$ or not defined, and the sum on the left-hand side of (3.18) equals 1, as desired.

  - If, on the other hand, $a \notin \delta^{\text{out}}(U)$, then $x_a = 0$. $v_1 \in U$ implies $v_2 \in U$. Consequently, $v_2 \in U \backslash S$. As $U \cap T = \emptyset$, $v_2 \in R$. Therefore, $z_{v_2} \in U \cap R$ and hence $z_{v_2} = 1$. Again, the sum on the left-hand side of (3.18) equals 1.

- Secondly, suppose $a \in \delta^{\text{in}}(S)$, i.e., $v_1 \notin S$, $v_2 \in S \subseteq U$ (then $v_2 \notin R$, and z_{v_2} is not defined). It follows that $u_a = -1$. Since $v_2 \in U$, we have $a \notin \delta^{\text{out}}(U)$, so $x_a = 0$.
  - If $a \in \delta^{\text{in}}(U)$, then $y_a = 1$ and $v_1 \notin U \backslash S$. Hence $z_{v_1} = 0$ and the sum on the left-hand side of (3.18) equals -1, as desired.
  - If, on the other hand, $a \notin \delta^{\text{in}}(U)$, then $y_a = 0$. $v_2 \in U$ implies $v_1 \in U$. Consequently, $v_1 \in U \backslash S$, so $z_{v_1} = 1$. As $U \cap T = \emptyset$, $v_1 \in R$. The sum on the left-hand side of (3.18) equals -1.

- If $v_1, v_2 \in S$, then $u_a = 0 = x_a = y_a$. z_{v_1} and z_{v_2} are not defined.

- The last case is $v_1, v_2 \notin S$. Here also, $u_a = 0$.
  - If $v_1, v_2 \notin U$, then $x_a = y_a = 0$. z_{v_1} and z_{v_2} are either $= 0$ or not defined. The sum on the left-hand side of (3.18) equals 0.
  - If $v_1, v_2 \in U$, then $x_a = y_a = 0$. As $v_1, v_2 \in U \backslash S$, $z_{v_1} = z_{v_2} = 1$. As $U \cap T = \emptyset$, $v_1, v_2 \in R$. We have $\delta_{v_1,a} = -1$ and $\delta_{v_2,a} = 1$. Hence, the sum on the left-hand side of (3.18) equals 0.
  - If $a \in \delta^{\text{in}}(U)$, z_{v_1} is $= 0$ or not defined, $z_{v_2} = 1$, $x_a = 0$, $y_a = 1$. We have $v_2 \in U \cap R$. The sum on the left-hand side of (3.18) equals 0.
  - If $a \in \delta^{\text{out}}(U)$, $z_{v_1} = 1$, z_{v_2} is $= 0$ or not defined, $x_a = 1$, $y_a = 0$. We have $v_1 \in U \cap R$. The sum on the left-hand side of (3.18) equals 0.

This establishes the validity of (3.18). Furthermore, we clearly have

$$c^{\mathrm{T}}x - d^{\mathrm{T}}y + b^{\mathrm{T}}z = c(\delta^{\text{out}}(U)) - d(\delta^{\text{in}}(U)) + \sum_{v \in U \backslash S} b(v).$$

Thus for every $S-T$-cut U in D, we have found a feasible vector (x, y, z) of the dual linear program (3.16) such that the value of its objective function equals capacity(U). It can be concluded that

$$\begin{aligned} &\min_{x,y \in \mathbb{R}^A,\, z \in \mathbb{R}^R} \{c^{\mathrm{T}}x - d^{\mathrm{T}}y + b^{\mathrm{T}}z \mid x,\, y \geq \mathbf{0}_A,\, x - y + \Delta^{\mathrm{T}}z = u\} \\ \leq\ &\min_{U \in \mathcal{C}_{S,T}(D)} \Big(c(\delta^{\text{out}}(U)) - d(\delta^{\text{in}}(U)) + \sum_{v \in U \backslash S} b(v) \Big). \end{aligned}$$

To show that the opposite inequality holds as well, let $x, y \in \mathbb{R}^A$ with $x, y \geq 0$, and $z \in \mathbb{R}^R$ such that $x - y + \Delta^{\mathrm{T}} z = u$. The theorem is proved if an $S-T$-cut U with capacity$(U) \leq c^{\mathrm{T}}x - d^{\mathrm{T}}y + b^{\mathrm{T}}z$ is found.

To this end, let f, (x, y, z) be a pair of optimal solutions for (3.15) respectively (3.16). Define the following set of vertices of D:

$$U \coloneqq S \cup \{u \in R \mid z < 0\}.$$

Then U is an $S-T$-cut, because $U \supseteq S$ and $U \cap T = \emptyset$. I shall show the following two statements:

$$x_a > 0 \quad \text{for all } a \in \delta^{\mathrm{out}}(U), \tag{3.19}$$

$$y_a > 0 \quad \text{for all } a \in \delta^{\mathrm{in}}(U). \tag{3.20}$$

From the Lemma 3.1 about complementary slackness and from the optimality of the pair of solutions, it can then be concluded that

$$f_a = c_a \quad \text{for all } a \in \delta^{\mathrm{out}}(U), \tag{3.21}$$

$$f_a = d_a \quad \text{for all } a \in \delta^{\mathrm{in}}(U). \tag{3.22}$$

Using (3.21) and (3.22), followed by an application of the elementary Equation (2.17) from the proof of Theorem 2.20, the definition of u, and ultimately the fact that the value of the respective objective functions coincide for a pair of optimal solutions, we can calculate the capacity of the cut U:

$$\begin{aligned}\text{capacity}(U) &= c(\delta^{\mathrm{out}}(U)) - d(\delta^{\mathrm{in}}(U)) + b(U \backslash S) \\ &= f(\delta^{\mathrm{out}}(U)) - f(\delta^{\mathrm{in}}(U)) + b(U \backslash S) = \text{value}(f) = u^{\mathrm{T}} f \\ &= c^{\mathrm{T}}x - d^{\mathrm{T}}y + b^{\mathrm{T}}z.\end{aligned}$$

Hence, the theorem is proved if (3.19) and (3.20) can be shown. As (x, y, z) is a feasible solution of the dual Program (3.16), we have $x - y + \Delta^{\mathrm{T}} z = u$. For the arc $a = (v_1, v_2) \in A$, this means that

$$x_a - y_a + \sum_{v \in \{v_1, v_2\} \cap R} \delta_{v,a} z_v = u_a. \tag{3.23}$$

We examine (3.23) for various arcs. To simplify the notation, set $z_v \coloneqq 0$ for $v \in S \cup T$. Let $a = (v_1, v_2) \in \delta^{\mathrm{out}}(U)$. Then $v_1 \in U$, i.e., either $v_1 \in S$ or $z_{v_1} < 0$, and $v_2 \notin U$, i.e., $z_{v_2} \geq 0$. Furthermore, $\delta_{v_1,a} = 1$ and $\delta_{v_2,a} = -1$.

- First, suppose $v_1 \in S$. Then from (3.23), we infer $x_a - y_a - z_{v_2} = 1$, or, equivalently, $x_a - y_a = 1 + z_{v_2} > 0$. As $y_a \geq 0$, it can be concluded that $x_a > 0$.

- Otherwise, $v_1 \in U \backslash S$. Then (3.23) yields $x_a - y_a + z_{v_1} - z_{v_2} = 0$, which can be written as $x_a - y_a = z_{v_2} - z_{v_1} > 0$. Again, $y_a \geq 0$ implies $x_a > 0$.

Now let $a = (v_1, v_2) \in \delta^{\text{in}}(U)$. Then $v_1 \notin U$, i.e., $z_{v_1} \geq 0$, and $v_2 \in U$, i.e., $v_2 \in S$ or $z_{v_2} < 0$. Moreover, $\delta_{v_1,a} = 1$ and $\delta_{v_2,a} = -1$.

- Suppose that $v_2 \in S$. Then (3.23) implies $x_a - y_a + z_{v_1} = -1$, or in other words, $x_a - y_a = -1 - z_{v_1} < 0$. As $x_a \geq 0$, we must have $y_a > 0$.
- Otherwise, $v_2 \in U \backslash S$. Then, from (3.23), it can be inferred that $x_a - y_a + z_{v_1} - z_{v_2} = 0$. Writing this differently yields $x_a - y_a = z_{v_2} - z_{v_1} < 0$. Because of $x_a \geq 0$, this necessitates $y_a > 0$.

We have thus shown the validity of (3.19) and (3.20), and this completes the proof. □

Chapter 4

Algorithms for Flow Maximization

Since its inception, network flow theory has spawn a reputable number of algorithms for finding a maximum flow in a network. They have diversely been classified. Most of the algorithms available maintain feasibility of either the primal or the dual variables throughout its execution. This is the feature that will be used in this chapter to distinguish between two major groups of algorithms: primal and dual ones.

This is by no means the only classification available of algorithms for solving the Maximum Flow Problem, but maybe the most widespread one. However, there are variants focussing on different aspects of the algorithm. For example Ahuja, Magnati, and Orlin in [1] distinguish between "augmenting path algorithms" and "preflow-push algorithms", which yields almost the same classification that is used in this text.

Section 4.1 presents a few of the most popular graph theoretical problems: the Min-Cost Flow Problem, the Shortest Path Problem, and finally the one of most relevance for this thesis, the Max-Flow Problem. In Section 4.2, three well-known algorithms that solve the Max-Flow Problem are discussed. They all belong to the first category of "primal" or "augmenting path" algorithms. The algorithm examined in Section 4.3 on the other hand, is an example of a "dual", or a "preflow-push" algorithm. The last section of this chapter, Section 4.4 comprises the specialization of the general Simplex Method of linear programming to the case of network flows, which results in the potent Network Simplex Algorithm.

4.1 Max-Flow and Related Problems

In this thesis, I have limited the material to selections from quite a specific topic: maximal flows in networks. Of course there are *a lot of* related subjects, in particular from combinatorial optimization, that would deserve being included, in order to give a more complete picture of network flows. Some of them are indeed very closely connected with flow maximization. With regard to the applications to be discussed in the next chapter, I want to mention at least briefly some other aspects of network flow theory that are most widespread in applications, in- and outside of mathematics.

4.1.1 The Min-Cost Flow Problem

The following is a well-investigated and -documented, and rather general problem, that has a very large range of applications. In a sense, it can be regarded as one of the most fundamental problems in network theory. It is the common generalization of the Maximum Flow and the Shortest Path Problem, both of which are popular research areas with numerous applications. In order to formulate it, I give a definition first:

4.1 Definition (General Cost Function)**.** Let $D = (V, A)$ be a digraph, and (S, T, b, c, d) a flow-quintuple for D. A function $\text{cost}\colon \overline{\mathbb{R}}^A \to \overline{\mathbb{R}}$ is called a (*general*) *cost function*. One says $f \in \overline{\mathbb{R}}^A$ is *of cost* K if $\text{cost}(f) = K$.

The (fairly general) formulation of the Min-Cost Flow Problem that I want to use now reads as follows:

MIN-COST FLOW PROBLEM

Input
- A digraph $D = (V, A)$
- A flow-quintuple (S, T, b, c, d)
- A number $W \in \overline{\mathbb{R}}$
- A cost function cost

Output
- An answer to whether or not $\{f \in \mathcal{F}_{S,T}^{b,c,d} \mid \text{value}(f) = W\} = \emptyset$ and
- If not, an (S, T, b, c, d)-flow f with $\text{value}(f) = W$ that minimizes $\text{cost}(f)$

In this most general setting, not much can be said about the solution. This is, because the solution obviously completely depends upon the nature of

the function cost, as is exemplified by a simple exchange of cost by $(-\text{cost})$. Therefore, usually restrictions are imposed upon cost. The most common version is as follows.

4.2 Definition (Linear Cost Function)**.** Let $D = (V, A)$ be a digraph, and (S, T, b, c, d) a flow-quintuple for D. Let $k \in \overline{\mathbb{R}}^A$. The function

$$\text{cost}\colon \mathcal{F}_{S,T}^{b,c,d} \to \overline{\mathbb{R}},\ f \mapsto \sum_{a \in A} k_a f(a)$$

is called a (*linear*) *cost function*.

For the solution of this much more specialized version (let us call it the *Linear Min-Cost Flow Problem*), polynomial time algorithms similar to the ones for the Max-Flow Problem (discussed below) exist. Another popular variant is to allow convex functions depending on $f(a)$ for the k_a's of the above definition, and not restrict them to be constants (for a discussion, see for example [1, Chapter 14]). In the following special case of the Min-Cost Flow Problem, the cost of an arc is thought of in terms of distance between its two ends.

4.1.2 The Shortest Path Problem

Paths belong to the most fundamental and most important objects in graph theoretical optimization. They are of direct practical relevance, coming up when searching for any kind of connection. Being able to find short connections is of obvious interest in many areas, ranging from personal route planing to macroeconomics. Here 'short' need not mean geographic distance, but may also rather concern costs, travel time, a combination of some of these, etc.

Another aspect of (short) paths that makes them worth studying is that finding them often arises as a component of another combinatorial problem. In particular, some of the algorithms that find maximum flows in a network do so by actually repeatedly finding (shortest) paths as sub-procedures.

4.3 Definition. Let $D = (V, A)$ be a digraph and $S, T \subseteq V$ two subsets. Let $l \in \overline{\mathbb{R}}^A$. The function

$$\text{length}\colon \mathcal{P}_{S,T} \to \overline{\mathbb{R}},\ P \mapsto \sum_{a \in A(P)} l_a$$

is called a *length function*. A *shortest* $S-T$-path is a path $P \in \mathcal{P}_{S,T}$ that minimizes length(P).

Shortest Path Problem

Input
- A digraph $D = (V, A)$
- Subsets $S, T \subseteq V$
- A vector $l \in \overline{\mathbb{R}}^A$

Output
- An answer to whether or not $\mathcal{P}_{S,T} = \emptyset$ and
- If not, a shortest $S - T$-path P (with respect to length)

This task is contained in the Linear Min-Cost Flow Problem. Straightforwardly, it can be verified that it is obtained from it by taking:

$$\begin{aligned} b &:= \mathbf{0}_V, \\ c &:= \mathbf{1}_A, \\ d &:= \mathbf{0}_A, \\ W &:= 1, \\ k &:= l. \end{aligned}$$

From any integer output-flow f of the Min-Cost Flow Problem with these parameters, the shortest path P one was looking for is obtained as the one with arc set $A(P) = \{a \in A \mid f(a) \neq 0\}$ (the arc set $A(P)$ obviously determines P).

The difficulty of the Shortest Path Problem naturally heavily depends upon the function length, i.e., upon the vector l. While the problem is relatively easy to solve in the case of, say $l = \mathbf{1}_A$, it is much harder to solve if l might have negative components. I want to describe a simple, but efficient, algorithm by Berge and Moore that finds shortest paths in the case of unit lengths, i.e., $l = \mathbf{1}_A$. Several of the algorithms introduced in the next section use an algorithm of this kind as a subroutine. This description is adopted from [21, p. 88f.]:

Berge-Moore Algorithm

Input
- A connected digraph $D = (V, A)$
- Subsets S, $T \subseteq D$

Output A shortest $S - T$-path in D

① `Let` $V_0 := S$ `and set` $i = 0$.

② `Check whether or not` $T \cap \left(V \setminus \bigcup_{j=0}^{i} V_j\right) \neq \emptyset$.

- If yes, go to ③.
- Otherwise, go to ④.

③ Define V_{i+1} to be the set of vertices $v \in V \setminus \bigcup_{j=0}^{i} V_j$ for which $(u, v) \in A$ for some $u \in V_i$. Store one such arc for every $v \in V_{i+1}$. Reset $i = i + 1$ and go to ②.

④ The arcs used so far form subtree of a shortest path tree of D which contains all vertices of T. For a vertex $v \in V_i$, there is a path from S to v in D of length i. Let $i_0 \coloneqq \min\{i \mid V_i \cap T \neq \emptyset\}$ and choose $t_0 \in V_{i_0} \cap T$. Then any path from S to t_0 in the shortest path tree is a shortest $S - T$-path in D.

It can be shown that this algorithm finds a shortest $S-T$-path in time $O(|A|)$ (see [21, p. 88f.]). It is hence one of the fastest algorithms available for this task.

4.1.3 The Max-Flow Problem

The remainder of this section is dedicated to a problem that is another special case of the Min-Cost Flow Problem:

MAX-FLOW PROBLEM

Input
- A digraph $D = (V, A)$
- A flow-quintuple (S, T, b, c, d)

Output
- An answer to whether or not $\mathcal{F}_{S,T}^{b,c,d} = \emptyset$ and
- If not, a maximal (S, T, b, c, d)-flow f

As shown in the previous sections, this problem is equivalent to the task of finding a minimum cut. (This does of course not mean that in certain cases one of the two might not be better suited for a direct algorithmic approach than the other.)

To see that the Max-Flow Problem is included in the Linear Min-Cost Flow Problem, choose two new vertices s_0, $t_0 \notin V$ and construct a slightly bigger network from the one given as input of the Max-Flow Problem as

follows:

$$
\begin{aligned}
V' &:= V \cup \{s_0\} \cup \{t_0\},\\
a_0 &:= (s_0, t_0),\\
A' &:= A \cup \{a_0\},\\
D' &:= (V', A'),\\
S' &:= S \cup \{s_0\},\\
T' &:= T \cup \{t_0\},\\
b' &:= b,\\
c'(a) &:= \begin{cases} c(a) & \text{if } a \in A \\ \infty & \text{if } a = a_0 \end{cases},\\
d'(a) &:= \begin{cases} d(a) & \text{if } a \in A \\ \infty & \text{if } a = a_0 \end{cases},\\
W &:= \infty\\
k_a &:= \begin{cases} -1 & \text{if } a \in \delta^{\text{out}}(S) \\ 1 & \text{if } a \in \delta^{\text{in}}(S) \\ 0 & \text{otherwise} \end{cases}.
\end{aligned}
$$

Every feasible flow in this augmented network has a value of $= \infty$,[1] only the costs $K = \sum_{a \in A} k_a f(a)$ are varying. There is a bijective correspondence between flows of cost K in the augmented network and the flows of value K in the original network, as is not difficult to see. Hence, a solution of the Min-Cost Flow Problem for the augmented setting gives a solution of the originally posed Max-Flow Problem.

The Shortest Path Problem and the Max-Flow Problem are in a sense complementary (with respect to the Min-Cost Flow Problem): in the former, there are no capacity constraints. The aim is just to minimize certain costs. In the latter, costs are irrelevant, the end is solely maximization of flow, subject to given capacities and certain demands.

Now, let us have a look at some of the standard algorithms for finding solutions to the Max-Flow Problem.

[1] In Definition 2.3, I refrained from allowing flows assuming infinite values on the arcs. This was mainly because most of the theory of linear programming is usually formulated for real (finite) functions. Still, most of the results presented in this text could easily be formulated for 'infinite' flows without major changes.

4.2 Primal Algorithms

In the integer case, all of the three problems described in the previous section consist of choosing some object from a finite set of possibilities. Therefore, they can theoretically be solved by listing all possibilities and picking the best one. Obviously, this might take a lot of time. In particular, this 'algorithm' does not have a running time that is polynomial in the number of arcs and vertices of the network: the number of feasible flows in the Max-Flow Problem is for example not polynomial in $|A|$ and $|V|$, since it depends on the arc capacities.

The first algorithm (deserving this label) for finding a maximum flow in a network was given by L.R. Ford Jr. and D.R. Fulkerson in 1955. In its original form, it has two main drawbacks: firstly, there are examples (specifically constructed, not taken from practice) for which disadvantageous choices of the augmenting path[2] can lead to non-polynomial running time. Secondly, if not all of the edge-capacities are rational, then even worse, the algorithm is not guaranteed to stop at all. (It should not go unmentioned that this second problem is a mere theoretical one; computers deal only with rational numbers.) Fortunately, both of these issues can be dealt with by quite simple rules regarding which path is to be chosen as an augmenting path. Examples of such rules were given by E.A. Dinic[3] as well as by J. Edmonds and R.M. Karp. They guarantee a bound on the number of steps the Ford-Fulkerson Algorithm needs to find a maximum flow which is polynomial in the size of $|V|$ and $|A|$, regardless of the arc-capacities or other input-data.

Since then, a number of variants as well as quite different algorithms have been introduced by various authors. Since the Max-Flow Problem can be formulated as a linear program (as described in Chapter 3), all algorithms available for linear programming can be utilized for solving it. As one might expect though, due to their generality, few of them can compete with specifically designed 'combinatorial' algorithms in terms of running time. That is, unless they are used in a somewhat more specialized version. In fact, there can be shown equivalence up to different degrees between some general linear programming algorithms and other more combinatorial ones.

The description of the algorithms in this section is rather informal and should just give an overall idea of how implementations would basically work.

[2] see Section 2.4

[3] another transcription of the name of this Russian mathematician that is sometimes found in the literature is "Dinits"

4.2.1 The Ford-Fulkerson Algorithm

The first algorithm I want to have a closer look at is the classical "labelling algorithm" by Ford and Fulkerson which I already mentioned several times. I start with giving a compact overview of its structure:

FORD-FULKERSON ALGORITHM

Input
- A digraph $D = (V, A)$
- A flow-quintuple (S, T, b, c, d) for D
- An initial feasible flow $f \in \mathcal{F}_{S,T}^{b,c,d}$

Output A flow $f_{\max} \in \mathcal{F}_{S,T}^{b,c,d}$ of maximal value

① Compute D_f[4].

② Look for an $S-T$-path P in D_f.

- If none exists stop. $f_{\max} \coloneqq f$ is maximal.
- If a path P is found, go to ③.

③ Compute a suitable ε and augment f along P by ε, i.e., reset $f \coloneqq f + \chi^P \varepsilon$. Then go to ①.

How to choose ε as large as possible, such that f remains feasible after the augmentation in ③, was described in the proof of Proposition 2.24. The main questions left open are how to find an initial flow and how to efficiently find and choose an augmenting path. The former was discussed in Section 2.6: basically any of the algorithms of the current chapter can be applied to some starting flow (not necessarily an (S, T, b, c, d)-flow) in order to find a feasible flow or to decide that there is none. For the moment, let us focus on the latter.

In fact, step ② is a well-known graph theoretical problem itself (namely the Shortest Path Problem with $l = \mathbf{0}_a$), for which several algorithms exist. For example, the Berge-Moore Algorithm from the prvious section, or the algorithm "FINDPATH" described by Papadimitriou and Steiglitz (cf. [18, p. 198]) both find an $S-T$-path in a digraph, if one exists. They are both basically implementations of an idea called *breadth-first search* (*BFS*), which represents a commonly used method for finding paths in a graph, be it directed or not.

[4] the residual graph; see Definition 2.22 on page 63

Complementary in a sense is *depth-first search* (*DFS*), another very well-known method of path searching. It dates back to 1882, when M. Tremaux suggested it as a method to traverse mazes (see [13, p. 234]). The problem of finding any $S-T$-path is usually somewhat easier than our original problem of finding a maximal $S-T$-flow. (Fortunately! Otherwise the above described procedure would not be a very useful one.) This algorithmic scheme, of splitting more complex problems repeatedly into related simpler problems, reappears in a lot of the algorithms presented in this chapter.

I now want to (very roughly) estimate the complexity of Ford and Fulkerson's Algorithm: as mentioned before, there are search-algorithms that can be implemented such that they find an $S-T$-path in time $O(|A|)$ (for example the Berge-Moore Algorithm). This is (more or less) the time needed to perform one iteration of the algorithm. How many iterations will there be? If all input data are assumed to be integral (if they are rational, they can simply be rescaled, using least common multiples; if irrational numbers occur, the algorithm need not terminate at all, as mentioned before), then in each iteration the flow value is augmented at least by 1. Consequently, the maximal number of iterations necessary to obtain a maximum flow equals the difference between the value of the initial flow and the maximum flow value. This difference can be estimated using Theorem 2.20 for any cut, e.g., S. Let f_0 respectively $f_{\max}$ be the initial feasible flow respectively any maximum flow. Set $U \coloneqq \max\{c(a) - d(a) \mid a \in A\}$. Then

$$\begin{aligned}\operatorname{value}(f_{\max}) - \operatorname{value}(f_0) &\leq \operatorname{capacity}(S) - \operatorname{cocapacity}(S)\\ &= \big(c(\delta^{\text{out}}(S)) - d(\delta^{\text{in}}(S))\big) - \big(d(\delta^{\text{out}}(S)) - c(\delta^{\text{in}}(S))\big)\\ &= \big(c(\delta^{\text{out}}(S)) - d(\delta^{\text{out}}(S))\big) + \big(c(\delta^{\text{in}}(S)) - d(\delta^{\text{in}}(S))\big)\\ &\leq |A|U + |A|U = O(|A| \cdot U)\end{aligned}$$

The time the algorithm needs is the time needed for one iteration times the number of iterations. Thus, a time complexity of $O(|A|^2 \cdot U)$ is established. (If $|U| = \infty$, the algorithm obviously need not terminate.) Here one problem becomes apparent: the bound involves the arc capacities. If some of them are for example of size $2^{|A|}$, the running time is not polynomial in $|A|$ and $|V|$.

Ford and Fulkerson's Algorithm of often termed "labelling algorithm", the reason for which is that the computation in ① and of ε in ③ can be implemented as a procedure of successively assigning certain labels to the vertices of D. That process, as originally described in [8, p. 11f.] (and since then in hundreds of other books), also yields some $S-T$-path P in D_f (if there is one). The problem is that the path found in that way is not chosen carefully enough to obtain a polynomial-time bound for the worst-case

running time. Ford and Fulkerson were already aware of the fact that their algorithm need not terminate and gave a counterexample (cf. [8, p. 21f.]). For slightly terminology-wise slicker formulations of the same example, see [18, p. 124–128] or [21, p. 153f.]. A few different examples can be found in [1, p. 205f.]. Examples that illustrate the exponential worst-case running time can also be found in the first two books ([18, p. 200], respectively [21, p. 153]), as well as in [14, p. 156]. As mentioned before, all of these difficulties are resolved by a slight variation of step ②:

4.2.2 The Edmonds-Karp Algorithm

Edmonds and Karp were the first to give a polynomial-time algorithm for the Max-Flow Problem, namely in 1972. This was achieved by quite a simple idea. Here is an outline of the algorithm:

EDMONDS-KARP ALGORITHM

Input
- A digraph $D = (V, A)$
- A flow-quintuple (S, T, b, c, d) for D
- An initial feasible flow $f \in \mathcal{F}_{S,T}^{b,c,d}$

Output A flow $f_{\max} \in \mathcal{F}_{S,T}^{b,c,d}$ of maximal value

① Compute D_f.

② Look for an $S-T$-path P in D_f.

- If none exists stop. $f_{\max} \coloneqq f$ is maximal.
- If a path is found, choose some shortest $S-T$-path P and go to ③.

③ Compute a suitable ε and augment f along P by ε. Then go to ①.

So everything stays the same as in the Ford-Fulkerson Algorithm, except for the type of the (easier) combinatorial problem which the original one (of finding a maximum flow) is reduced to. It is a little more difficult this time: finding a *shortest* $S-T$-path, instead of just *any* $S-T$-path. Here 'shortest' means containing a minimum number of edges. The reward for this additional effort is the following:

4.4 Theorem. *Let $D = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for D. Let some (S, T, b, c, d)-flow f be given. Then the Edmonds-Karp Algorithm, initiated with f, stops after at most $|V| \cdot |A|$ iterations and yields a maximal flow.*

A proof can for example be found in [14, p. 160f.], or [21, p. 153f.]. A breadth-first search algorithm can (as mentioned before) be implemented in time $O(|A|)$, which hence is also a bound for the time needed for each iteration of the algorithm. The resulting overall running time for the Edmonds-Karp Algorithm can therefore be estimated by $O(|V| \cdot |A|^2)$. Note that, as desired, this does not depend on any arc capacities.

The problem of finding a shortest path in a (di)graph is a very well documented one (as described in the previous section). Besides the Berge-Moore Algorithm, there are many more algorithms available. For example, the above mentioned "FINDPATH" by Papadimitriou and Steiglitz can be implemented in such a way (in breadth-first manner) that it delivers a shortest $S-T$-path. In fact, all breath-first algorithms produce a shortest path tree. Another example of this is given by the popular shortest path algorithm by Dijkstra—documented, among many other publications, in [21, Section 7.2], [1, Section 4.5], [2, Section 2.3.3], [13, Section 3.6], or [14, p. 141f.].

Depth-first search can in general not be used for implementation of Edmonds and Karp's Algorithm, since it is not guaranteed to find a path that is a shortest one. For thorough discussions see [21, Chapters 6–8], [2, Chapter 2], or [13, Chapter 3].

4.2.3 The Dinic Algorithm

Dinic observed that Ford and Fulkerson's Algorithm could be speeded up if in each iteration one would augment f not only along a simple path in D_f, but use a whole flow g in D_f for augmentation. Achieved in this way is the following: in Edmonds and Karp's Algorithm, it can be proved that the length of a shortest $S-T$-path in D_f is non-decreasing from one iteration to the next (See [2, p. 115], [13, p. 167], or [21, p. 154f.]). It hence runs in 'phases' of increasing shortest $S-T$-path-length, during each of which the length of a shortest $S-T$-path remains the same. Using Dinic's technique, it can be achieved that the length of a shortest $S-T$-path in D_f increases in *every* iteration of the algorithm. As the length any path lies between 0 and $|A|$, the algorithm consequently yields a maximum flow in at most $|A|$ iterations.

How does augementation along flows work? Let $f \in \mathcal{F}_{S,T}^{b,c,d}(D)$. Define a

capacity function c_f on A_f as follows:

$$\begin{cases} c_f(a) \coloneqq c(a) - f(a) & \text{if } a \in A^{f<c} \\ c_f(a^{-1}) \coloneqq f(a) - d(a) & \text{if } a \in A^{f>d} \end{cases}$$

Then, for any $(S, T, \mathbf{0}_R, c_f, \mathbf{0}_{A_f})$-flow g in D_f, it can be shown that $f' \in \mathcal{F}^{b,c,d}_{S,T}(D)$, for the flow

$$f' \colon A \to \mathbb{R},\ a \mapsto f(a) + g(a) - g(a^{-1}).$$

(Herein, set $g(a) = 0$, respectively $g(a^{-1}) = 0$, if those expressions are not defined, i.e., if $a \notin A_f$, respectively $a^{-1} \notin A_f$.) With this definition, Ford and Fulkerson's Algorithm, as well as Edmonds and Karp's, can be viewed as special cases in which the augmenting flows are nonzero only on single $S-T$-paths. The question hence is how to make a choice of g cleverer than that. It turns out that the *sum of all* 'single-pathed' augmenting flows used in one 'phase' of Edmonds and Karp's Algorithm is a flow g which is a good choice. Of course, one would not want to have to execute Edmonds and Karp's Algorithm in order to find out what such a g would be. Fortunately, it can be shown that the flows of this kind are exactly so-called *blocking flows* in the so-called *level graph* of D_f.

4.5 Definition. Let $D = (V, A)$ be a digraph. A function measuring the distances between different vertices of D is defined as follows: let S, $T \subseteq V$. Then set

$$\operatorname{dist}_D(S, T) \coloneqq \begin{cases} \text{the length of a shortest } S-T\text{-path} & \text{if } \mathcal{P}_{S,T}(D) \neq \emptyset \\ \infty & \text{if } \mathcal{P}_{S,T}(D) = \emptyset \end{cases}.$$

If $S = \{s\}$, or $T = \{t\}$, then the braces are left out, as usually.

4.6 Definition. For a digraph $D = (V, A)$ and a flow-quintuple (S, T, b, c, d) for D, the *level graph* D^L_f of D_f is the digraph

$$\Big(V, \big\{(u, v) \in A_f \mid \operatorname{dist}_{D_f}(S, v) = \operatorname{dist}_{D_f}(S, u) + 1\big\}\Big).$$

Also define c^L_f to be the restriction of c_f to $A(D^L_f)$.

4.7 Definition. For a digraph $D = (V, A)$ and a flow-quintuple (S, T, b, c, d) for D, a flow $f \in \mathcal{F}^{b,c,d}_{S,T}(D)$ is called a *blocking flow* if $(V, A^{f<c})$ contains no $S-T$-path; or equivalently if for every flow $f' \in \mathcal{F}^{b,c,d}_{S,T}(D)$ the inequality $f \leq f' \leq c$ implies $f' = f$.

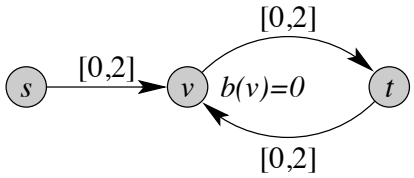

Figure 4.1: A blocking flow need not be maximal.

4.8 Example. Note that a blocking flow need not be of maximum value. Take a look a Figure 4.1. Let the $s-t$-flow f be defined as follows:

$$f((s,v)) = f((t,v)) = 1,\ f((v,t)) = 2.$$

f is a blocking flow, but value$(f) = 1$ and f is not maximal: the following $s-t$-flow f_{max} has greater value (in fact, it is the unique maximal $s-t$-flow in the graph):

$$f_{max}((s,v)) = f_{max}((v,t)) = 2,\ f_{max}((t,v)) = 0.$$

Dinic Algorithm

Input
- A digraph $D = (V, A)$
- A flow-quintuple (S, T, b, c, d) for D
- An initial feasible flow $f \in \mathcal{F}_{S,T}^{b,c,d}$

Output A flow $f_{\max} \in \mathcal{F}_{S,T}^{b,c,d}$ of maximal value

① `Compute` D_f^L.

② `Check if there is a blocking flow` $g \neq \mathbf{0}_{A(D_f^L)}$ `in` D_f^L `which is feasible with respect to` $(S, T, \mathbf{0}_R, c_f^L, \mathbf{0}_{A_f^L})$.

- `If none exists stop.` $f_{\max} \coloneqq f$ `is maximal.`
- `If such a` g `is found, go to ③.`

③ `Augment` f `along` g. `Go to ①.`

A worst-case running time bound for Dinic's Algorithm is given by $O(|V|^2 \cdot |A|)$. This result is shown, e.g., in [21, p. 154f.], [2, p. 116f.], or [1, p. 221–223].

I did not explain explicitly how to find a blocking flow; however with subroutines for this task other than the one suggested by Dinic, better running times can be obtained: sleator described a method for finding blocking flows that improved the running time of the above algorithm to $O(|V| \cdot |A| \cdot \log |V|)$ (cf. [14, p. 162f.]).

4.2.4 Other Primal Algorithms

There are other variants of the ideas exploited by Edmonds and Karp's or Dinic's Algorithm. For example, Papadimitriou and Steiglitz proposed the following algorithm in [18, Section 9.4]: in each iteration the residual graph is constructed, just like in all of the above algorithms. Only, instead of looking for a shortest augmenting path or a blocking flow in order to augment the current feasible solution, they give a clever rule for finding a *maximal* flow in the layered residual graph, which is then used for augmentation. The running time bound they derive for their algorithm is $O(|V|^3)$.

Another possibility is to look for a path in the residual graph along which a maximum augmentation is possible, usually dubbed *fattest augmenting path*. To this end, one has to look for a path such that the 'bottleneck' (i.e., the minimum residual capacity) is maximized. This can be achieved using a suitable variant of Dijkstra's shortest path algorithm.

4.3 Dual Algorithms

All of the algorithms discussed so far worked after a common scheme: it follows from Definition 2.3 and Proposition 2.23, that for a function $f\colon A \to \mathbb{R}$, to be an (S, T, b, c, d)-flow of maximum value is equivalent to satisfying the following conditions:

$$\text{(i)} \quad \text{excess}_f(v) = b(v) \qquad \forall v \in R, \tag{4.1}$$

$$\text{(ii)} \quad f(a) \leq c(a) \qquad \forall a \in A, \tag{4.2}$$

$$\text{(iii)} \quad f(a) \geq d(a) \qquad \forall a \in A, \tag{4.3}$$

$$\text{(iv)} \quad \nexists\ S - T\text{-path in } D_f. \tag{4.4}$$

The algorithms of Ford and Fulkerson, Edmonds and Karp, and Dinic all start with a function f satisfying Conditions (i)–(iii) (i.e., f is a feasible (S, T, b, c, d)-flow), and maintain them in each iteration, until Condition (iv) is finally fulfilled as well. Since Conditions (i)–(iii) correspond to feasibility of the primal problem (finding a maximum flow), which is maintained at all time, these algorithms can be thought of as *primal* ones.

Another approach was introduced by Goldberg and Tarjan. If Condition (iv) is satisfied, then for the set $U \subseteq V$ of all vertices of D reachable from S in D_f (as considered in Proposition 2.23), we have $U \in \mathcal{C}_{S,T}$. This is because clearly every vertex $v \in S$ is reachable in D_f from S (so $U \supseteq S$), and by Condition (iv), the same is true for none of the vertices $v \in T$ (hence $U \cap T = \emptyset$). So validity of (iv) corresponds to feasibility of the dual problem (i.e., finding a minimum $S-T$-cut). As this feasibility is maintained throughout, this algorithm (and algorithms having a similar concept) is often referred to as a *dual* algorithm.

Goldberg and Tarjan's Algorithm starts with a function f satisfying Conditions (ii)–(iv), maintains them throughout, and stops as soon as Condition (i) holds as well. At every stage, a feasible dual solution can be obtained directly from f, therefore Goldberg and Tarjan's Algorithm can be considered a dual algorithm.

4.3.1 The Goldberg-Tarjan Algorithm

A few notions need to be introduced first, in order to describe Goldberg and Tarjan's Algorithm:

4.9 Definition. Let $D = (V, A)$ be a digraph, and (S, T, b, c, d) be a flow-quintuple for D. A function $f\colon A \to \mathbb{R}$ is called an (S, T, b, c, d)-*preflow for* D if it satisfies

$$\text{(i')} \quad \text{excess}_f(v) \geq b(v) \qquad \forall v \in R, \tag{4.5}$$

$$\text{(ii)} \quad f(a) \leq c(a) \qquad \forall a \in A, \tag{4.6}$$

$$\text{(iii)} \quad f(a) \geq d(a) \qquad \forall a \in A. \tag{4.7}$$

A vertex $v \in R$ with $\text{excess}_f(v) > b(v)$ is called *active*. Vertices which are not active are called *inactive*.

Then, obviously, an (S, T, b, c, d)-preflow is an (S, T, b, c, d)-flow if and only if there are no active vertices in V.

4.10 Definition. Let $D = (V, A)$ be a digraph and (S, T, b, c, d) be a flow-quintuple for D. Let f be an (S, T, b, c, d)-preflow. A function $\text{label}\colon V \to \mathbb{Z}_+$ is called a *distance labelling* if

$$\text{label}(s) = |V| \qquad \forall s \in S, \tag{4.8}$$

$$\text{label}(t) = 0 \qquad \forall t \in T, \tag{4.9}$$

$$\text{label}(v) \leq \text{label}(w) + 1 \qquad \forall (v, w) \in A_f. \tag{4.10}$$

4.11 Lemma. *Let $D = (V, A)$ be a digraph and (S, T, b, c, d) be a flow-quintuple for D. Let f be a preflow. Then a distance labelling exists if and only if there is no $S - T$-path in D_f. (Which, if f happens to be a flow, is by Proposition 2.23 equivalent to f being of maximal value.)*

Proof. Suppose there is an $S - T$-path P in D_f. Let label be any function satisying (4.8) and (4.10). P is a path, hence it traverses every vertex at most once. Therefore, $|V(P)| \leq |V|$. The first vertex s of P is in S, so $\text{label}(s) = |V|$ by (4.8). Now (4.10) implies that if one walks along the vertices of P, the value of label can decrease at most by 1 from one vertex to the next. For the last vertex $t \in T$ of P, this necessitates

$$\text{label}(t) \geq \text{label}(s) - (|V(P)| - 1) \geq |V| - |V| + 1 = 1.$$

This is clearly a contradiction to (4.9), such that label cannot be a distance labelling.

Conversly suppose that D_f contains no $S - T$-path. Let $U \subseteq V$ be the set of all vertices reachable from S in D_f. As then $U \cap T = \emptyset$, it is not hard to verify that the following function is a distance labelling:

$$\text{label}(v) := \begin{cases} |V| & \text{if } v \in U \\ 0 & \text{otherwise} \end{cases}.$$

□

Goldberg-Tarjan Algorithm

Input
- A digraph $D = (V, A)$
- A flow-quintuple (S, T, b, c, d) for D
- An initial feasible flow $f_0 \in \mathcal{F}_{S,T}^{b,c,d}$

Output A flow $f_{\max} \in \mathcal{F}_{S,T}^{b,c,d}$ of maximal value

① Initialize

$$\text{label}(v) \coloneqq \begin{cases} |V| & \texttt{if } v \in S \\ 0 & \texttt{otherwise} \end{cases}$$

$$f(a) \coloneqq \begin{cases} c(a) & \texttt{if } a \in \delta^{\texttt{out}}(S) \\ f_0(a) & \texttt{otherwise} \end{cases}$$

② Check if there is an active vertex in $v \in R$.

- If none exixts, stop. $f_{\max} \coloneqq f$ is maximal.
- If such a v is found, go to ③.

③ While v is active, iteratively do the following until v becomes inactive: check if there is an arc $a = (v, w) \in A_f$ with $\text{label}(w) = \text{label}(v) - 1$.

- If none exists, call the subroutine RELABEL with input v.
- If such an a is found, call the subroutine PUSH with input a.

When v becomes inactive, go to ②.

RELABEL(v)

① Reset $\text{label}(v) \coloneqq \text{label}(v) + 1$.

PUSH(a)

① - If $a = (v, w) \in A$, reset $f(a) \coloneqq f(a) + \varepsilon$ with

$$\varepsilon \coloneqq \min\big\{c(a) - f(a), \text{excess}_f(v) - b(v)\big\}.$$

- If $a^{-1} = (w, v) \in A$, reset $f(a^{-1}) \coloneqq f(a^{-1}) - \delta$ with

$$\delta \coloneqq \min\big\{f(a^{-1}) - d(a^{-1}), \text{excess}_f(v) - b(v)\big\}.$$

Now, let us have a look at whether this algorithm really does what it is hoped to do, i.e., whether it works correctly and finds a maximum flow:

4.12 Proposition. *The function f is a preflow at all stages of the algorithm. In particular, when the algorithm stops, f is a flow.*

Proof. The flow f_0 given as input is a preflow trivially. In step ①, while moving from f_0 to f, obviously Conditions (4.6) and (4.7) remain valid. What is altered are only flow-values at arcs *a entering* some vertices $v \in R$ (i.e., arcs $a \in \delta^{\text{in}}(v)$). At arcs where an alteration occurs, f is augmented, resulting in an augmentation of $\text{excess}_f(v)$. But this maintains (4.5).

It is straightforward to verify that ε and δ in the procedure PUSH are chosen just right, in order to keep (4.5), (4.6), and (4.7) valid. A run of RELABEL is irrelevant for f being a preflow or not. □

4.13 Proposition. *The function* label *is a distance labelling at all stages of the algorithm. In particular, when the algorithm stops with flow f as output, f is of maximum value.*

Proof. For the initial f, the residual graph D_f does not contain an $S-T$-path. So by Lemma 4.11, the initial label is a distance labelling. Let $v \in R$. By (4.10), for all vertices $w \in \delta^{\text{out}}_{D_f}(v)$, we have $\text{label}(w) \geq \text{label}(v) - 1$. If the procedure RELABEL(v) is called, then only because there are no vertices $w \in \delta^{\text{out}}_{D_f}(v)$ with $\text{label}(w) = \text{label}(v) - 1$. By integrality of label, this implies $\text{label}'(w) = \text{label}(w) \geq \text{label}(v) = \text{label}'(v) - 1$ for all $w \in \delta^{\text{out}}_{D_f}(v)$, with label′ being the updated function. This is (4.10) for the updated function. For vertices in $S \cup T$, the value of label is never changed.

In the procedure PUSH(a), the only thing that could happen related to label is that arcs might be added to A_f (or be erased from it; but this makes Condition (4.10) only easier to satisfy). But PUSH(a) is only called for $a = (v, w)$ with $\text{label}(w) = \text{label}(v) - 1$. The arc (w, v) is the only one that could be added to A_f, and for this arc, (4.10) holds valid: $\text{label}(w) = \text{label}(v) - 1 \leq \text{label}(v) + 1$.

So label is a distance labelling initially and maintains this property at every iteration of the algorithm. □

It is thus established that Goldberg and Tarjan's Algorithm works as it should. It is the fastest of the algorithms that have been considered in this chapter. In fact, it is one of the fastest algorithms available. Running time bounds that can be proved for various clever implementations include $O(|V|^2 \cdot \sqrt{|A|})$, or alternatively $O(|V| \cdot |A| \cdot \log \frac{|V|^2}{|A|})$. For a complexity analysis see for example [14, p. 165–168] or [21, p. 157–159].

4.4 Algorithms Derived from Methods from Linear Programming

The Simplex Algorithm was the first algorithm developed that could solve a general linear program of the form (1.4):

$$\max\{c^{\mathrm{T}}x|Ax \leq b\}.$$

By now, there are a lot of variants and different implementations of it. In addition, several algorithms of thoroughly different structures have been conceived, such as the ellipsoid method mentioned before. One feature all these algorithms have in common is, that in their most general formulation, suited to solve *any* linear program, they do not have running times useful in practice when applied to highly specialized linear programs such as the combinatorial problems considered in Section 4.1. In these cases, the performances are greatly inferior when compared with the algorithms described in Sections 4.2 and 4.3.

The reason for this is quite obvious: being able to handle so many problems simultaneously, they cannot exploit the particular structure of specific combinatorial problems. An example illustrating this can be found in [23, p. 190f.].

However, there is a 'streamlined' version of the general Simplex Algorithm, called the *Network Simplex Algorithm*, that is designed specifically for solving network flow problems and hence delivers much better running times.

4.4.1 The Network Simplex Algorithm

In this Section, I want to demonstrate how to efficiently apply the Simplex Algorithm, as described in Section 1.4.3, to the Max-Flow Problem introduced in Section 4.1.3. Let therefore the digraph $G = (V, A)$ be given (the symbol D is unfortunately needed for something else in this section), as well as a flow-quintuple (S, T, b, c, d) for G. Recall that $R = V\backslash S \cup T$. As in Section 3.1, the balance condition can be written as

$$\Delta f = b. \tag{4.11}$$

Here $\Delta = (\Delta_a)_{a\in A}$ is the $|V| \times |A|$ node-arc incidence matrix of G, defined on page 23. Each Δ_a is a $|V|$-dimensional vector, f is $|A|$-dimensional, and b is $|V|$-dimensional. Furthermore, a feasible flow f should satisfy

$$d \leq f \leq c. \tag{4.12}$$

d and c are both $|A|$-dimensional. Finally, the objective function that is to be maximized is

$$u^{\mathrm{T}} f = \sum_{a \in A} u_a f(a), \quad \text{with} \quad u_a \coloneqq \begin{cases} 1 & \text{if } a \in \delta^{\mathrm{out}}(S) \\ -1 & \text{if } a \in \delta^{\mathrm{in}}(S) \\ 0 & \text{otherwise} \end{cases}. \tag{4.13}$$

Basic Solutions

First, I want to discuss the graph theoretical equivalents of bases and basis structures from the Simplex Algorithm. It turns out that there is a one-to-one correspondence between basic solutions and a certain type of flows in G, called *spanning tree solutions*.

4.14 Definition. Let $G = (V, A)$ be a digraph, (S, T, b, c, d) a flow-quintuple for G. Let T be a spanning tree of G. Set $B \coloneqq A(T)$. A (not necessarily feasible) flow f of G satisfying the balance-condition (4.11) is a called a *basic solution* if for every arc $a \in A \backslash B$ one of $f(a) = c(a)$ or $f(a) = d(a)$ holds. If in addition $d \leq f \leq c$, i.e., f is a feasible (S, T, b, c, d)-flow, then f is called a *spanning tree solution*.

If (C, D) is a partition of $A \backslash B$, then the triplet (B, C, D) is called a *spanning tree structure (for the tree T)* of G. If there exists a (feasible) spanning tree solution f such that $f(a) = c(a)$ for all $a \in C$ and $f(a) = d(a)$ for all $a \in D$, then the spanning tree structure is called *feasible*. It can be shown that assigning flow values to all arcs in $A \backslash B$ determines the entire flow (assuming that the balance-condition needs to be respected). In particular, for every spanning tree structure (B, C, D), there exists a unique basic solution f such that $f(a) = c(a)$ for all $a \in C$ and $f(a) = d(a)$ for all $a \in D$.

It will in the sequel prove useful to randomly select one vertex $r \in V$ and to think of the spanning tree T as rooted at r. Set $V' \coloneqq V \backslash \{r\}$. For $a \in A$, define $\Delta'_a \coloneqq (\delta_{v,a})_{v \in V'}$ to be the vector obtained from Δ_a by deleting the row corresponding to the vertex r. Also set $\Delta' \coloneqq (\Delta'_a)_{a \in A}$ and $\Delta'_J \coloneqq (\Delta'_a)_{a \in J}$, for any subset $J \subseteq A$. Let b' be the $|V| - 1$-dimensional vector obtained from b by deleting $b(r)$.

If all the rows of Equation (4.11) are summed, the following equation is obtained:

$$\sum_{v \in R} b(v) = \sum_{v \in R} \Big(\sum_{a \in \delta^{\mathrm{in}}(v)} f(a) - \sum_{a \in \delta^{\mathrm{out}}(v)} f(a) \Big) = \sum_{a \in \delta^{\mathrm{in}}(R)} f(a) - \sum_{v \in \delta^{\mathrm{out}}(R)} f(a).$$

This is just a re-statement of the conclusion of Lemma 2.2 for $U = R$. But Lemma 2.2 is valid in general. Hence, the system (4.11) is redundant, and it can be concluded that (at least) one of the constraints is superfluous. This means that the system is linearly dependent. As the maximum number of linearly independent columns is the same as the maximum number of linearly independent rows, the number of linearly independent columns in Δ is less or equal to $|V| - 1$. The opposite inequality needs a little preparation:

4.15 Definition. Two indices can be assigned to the vertices in V' in the following way. Choose $v \in V'$. Let P_v be the unique undirected $r - v$-path in T. Define pred(v) to be the second last vertex of P_v, i.e., the one visited directly before v when following the direction of P_v. pred(v) is referred to as the *predecessor of* v. A vertex w is called a *successor of* v if pred$(v) = w$. Consequently, the vertices without successors are exactly the leaves of the tree. Note that a vertex may have several successors, but every vertex has exactly one predecessor. The *descendants* of v are v itself, its successors, the successors of its successors, and so on.

The second index, thread(v), arises from a traversal of the whole tree, and defines an order of the vertices in V. To this end, a depth-first search starting at r is performed. For $v \in V$, thread(v) is defined to be the vertex visited by the depth-first algorithm just after v. For the last vertex encountered, the thread-index is set to be r. A *thread traversal* of the tree is an order of its vertices according to the thread-index. That is, r is the first vertex, thread(r) the second one, and so on. A *reverse thread traversal* is the reverse of that order. That is, the vertex w with thread$(w) = r$ is the first vertex, the vertex with thread-index w is the second, and so on.

Without going into details of depth-first search, the important property of the so defined thread-index, which is not very hard to show,[5] is the following: in a reverse thread traversal, every vertex is visited *after all of its descendants* (other than the vertex itself) have been visited.

4.16 Proposition. *Let $G = (V, A)$ be a digraph, T a spanning tree of G rooted at r. Let $B = A(T)$. Then the rows and columns of the the matrix Δ'_B can be rearranged in such a way that the resulting matrix is lower triangular, with nonzero-entries only in the main diagonal. Consequently, Δ'_B is nonsingular, its columns are linearly independent.*

Proof. As T is a spanning tree, it has $|V|$ vertices and $|V| - 1$ arcs. Hence, the matrix Δ'_B, obtained from the $|V| \times (|V| - 1)$ node-arc incidence matrix

[5] see for example [1, p. 76]

of T by deleting the row corresponding to the vertex r, has the dimension $(|V|-1)\times(|V|-1)$.

Now the rows of Δ'_B are rearranged according to the reverse thread-traversal. Moreover, each arc $a \in B$ can be assigned uniquely to the vertex v for which a is the last arc of the unique $r-v$-path P_v in T. In that way, the columns of the matrix can also be ordered according to the reverse thread traversal. The resulting matrix is lower triangular: select any vertex $v \in V'$. Suppose that v is the i-th vertex selected by the reverse thread traversal. The i-th arc a selected by the reverse thread traversal is the last arc of P_v. In particular, a is incident to v. This means that the element $\delta_{v,a} = \pm 1$, corresponding to v and a, is a diagonal element in the rearranged matrix. We have $|V'| = |V| - 1$, therefore all of the $|V|-1$ diagonal elements of the rearranged matrix have to be nonzero.

Let $\mathrm{pred}(v) = w$. If $w = r$, then $\delta_{v,a}$ is the only nonzero element in the column Δ'_a corresponding to a. Otherwise, as v is a descendant of w, v will be visited before w in the reverse thread-traversal. So there is another nonzero entry in the column corresponding to a, but it must lie below the diagonal. □

4.17 Proposition. *Let $G = (V, A)$ be a connected digraph. Then there exists a one-to-one correspondence between basis structures of the Max-Flow Problem and spanning tree structures of G.*

Proof. Let (B, C, D) be a spanning tree structure for the tree T rooted at r, with $A(T) = B$. Then, by the previous proposition, the $|V|-1$ columns of Δ'_B are linearly independent. Hence, (B, C, D) is a basis structure of the Max-Flow Problem.

Let conversely (B, C, D) be a basis structure. By Proposition 1.14, there exists a spanning tree T_0 of G. The node-arc incidence matrix $(\Delta_a)_{a\in A(T_0)}$ of T_0, which is a submatrix of Δ, has $|V|-1$ linearly independent columns. Consequently, Δ has at least $|V|-1$ linearly independent columns. But, as mentioned above, Δ has *at most*, and hence *exactly*, $|V|-1$ linearly independent columns. This means that $|B| = |V| - 1$. B corresponds to a subgraph T of G having $|V|-1$ arcs.

Now let K be an undirected cycle in G, assigned with an arbitrary orientation. Then the expression

$$\sum_{a\in A(K)} \chi^K(a)\Delta'_a \tag{4.14}$$

is a linear combination of some of the columns of Δ'. As K is a cycle, for every vertex in $V(K)\backslash\{r\}$ there are exactly two indices a_1, $a_2 \in A(K)$ for which

the entry δ_{v,a_i} of Δ'_{a_i} is $\neq 0$. In fact, it is not hard to see that the sum (4.14) equals zero: either one of these two entries δ_{v,a_i} is 1 and the other one equals -1, with $\chi^K(a_1) = \chi^K(a_2)$, or $\delta_{v,a_1} = \delta_{v,a_2}$ are both 1 or -1, with $\chi^K(a_1) = 1$ and $\chi^K(a_2) = -1$, or the other way round. This means that the columns of Δ' corresponding to a cycle in G are linearly dependent. As (B, C, D) is a basis structure, Δ'_B has only linearly independent columns. Therefore T cannot contain a cycle and hence is a forest. By Proposition 1.14, T is a tree. □

Henceforth, fix a root node r for the spanning tree T and set $B \coloneqq A(T)$. Then the $V' \times B$ node-arc incidence matrix Δ'_B is exactly the basis matrix used in the Simplex Method. From now on, Δ'_B will be assumed to be in lower triangular form, the rows and columns ordered according to the reverse thread traversal, as in the proof Proposition 4.16. The main steps of the Network Simplex Algorithm can be overviewed as follows:

Network Simplex Algorithm

Input
- A digraph $G = (V, A)$
- A flow-quintuple (S, T, b, c, d) for G
- An initial feasible spanning tree structure (B, C, D)

Output A flow $f_{\max} \in \mathcal{F}_{S,T}^{b,c,d}$ of maximal value

① `Compute the flow` f `and the reduced costs` $(u_a^\pi)_{a\in A}$ `corresponding to the current spanning tree structure.`

② `Look for arcs in` C `with reduced cost less than zero or arcs in` D `with reduced cost greater than zero.`

- `If none exists, stop.` $f_{\max} \coloneqq f$ `is maximal.`
- `Otherwise, select one of them as the entering arc and got to ③.`

③ `Select a leaving arc.`

④ `Update` (B, C, D) `and go to ①.`

Finding an Initial Feasible Flow

I now want to explain how the technique described in Section 1.4.3, on page 37, for avoiding the problem of having to find an initial feasible solution, translates to the case of networks.

Introducing new variables means introducing new arcs. This can be done in the following way: a new vertex $\overline{v}$ without balance-condition is added ($\overline{v}$ can be thought of as being a source or a sink vertex, i.e., an element of S or T). Then some arcs are added as well: for $v \in R$, let

$$\overline{b}(v) \coloneqq b(v) - \sum_{a \in \delta^{\text{in}}(v)} d(a) + \sum_{a \in \delta^{\text{out}}(v)} d(a).$$

Then add the arc $(v, \overline{v})$ for every vertex $v \in R$ with $\overline{b}(v) > 0$, and the arc $(\overline{v}, v)$ for every vertex $v \in R$ with $\overline{b}(v) < 0$. Write $\overline{A}$ for the set of new arcs. The capacity intervals of the new arcs are $[0, M]$, where M needs to be chosen sufficiently large ($\geq \max\{|\overline{b}(v)| \mid v \in R\}$), and the objective function has to be altered suitably:

$$U(f) \coloneqq \sum_{a \in A} u_a f(a) - \sum_{a \in \overline{A}} \overline{u}_a f(a),$$

where $\overline{u}_a$ are large numbers. Then the following flow is a feasible initial flow for the augmented network:

$$f(a) \coloneqq \begin{cases} d(a) & \text{if } a \in A \\ \overline{b}(v) & \text{if } a = (v, \overline{v}) \\ -\overline{b}(v) & \text{if } a = (\overline{v}, v) \end{cases}.$$

Computing the Flow Corresponding to a Given Spanning Tree Structure

Let (B, C, D) be a spanning tree structure. Then the Simplex Algorithm would, as described in Section 1.4.3, compute the values of the arcs in B by premultiplying the following equation by $(\Delta'_B)^{-1}$:

$$\Delta'_B f_B = b' - \Delta'_C c_C - \Delta'_D d_D. \tag{4.15}$$

Since Δ'_B is lower triangular, in the case of networks the calculation of f_B can be performed relatively simply: the first row of Δ'_B contains only one nonzero entry, namely in the first column. This entry is either 1 or -1, as all the nonzero entries of Δ'_B, because Δ'_B is a node-arc incidence matrix. The

value of the first entry of f_B can therefore be obtained by multiplying the first row of the right-hand side of (4.15) by the inverse of the nonzero entry of the first row of Δ'_B (which is the same as that entry itself, 1 or -1).

Proceeding further, there are at most two nonzero entries in the second row of Δ'_B, in the first and in the second column. But as the first entry of f_B (corresponding to the first column of Δ'_B) has already been determined, the value of the second entry of f_B can be calculated just as easily.

Using this approach of forward substitution, the whole vector f_B can be computed very economically, the only arithmetic operations utilized being additions and subtractions.

How do these calculations translate into combinatorial terminology? The first row of Δ'_B contains only one nonzero entry. This implies that it corresponds to a node v_1 that is incident to only one arc a_1 of the spanning tree, i.e., v_1 is a leaf node. As b_{v_1} is known, and as there is only one arc a incident to v_1 for which f_a is variable (all other arcs a of G incident to v_1 are in C or D, and hence the flow on these arcs is bound to be c_a respectively d_a), to obtain a feasible flow f that satisfies the balance-condition for v_1, it is necessary to set

$$\begin{aligned} f_{a_1} \coloneqq b(v_1) &- \Big(\sum_{a \in \delta^{\text{in}}_G(v_1) \cap C} c_a + \sum_{a \in \delta^{\text{in}}_G(v_1) \cap D} d_a \Big) \\ &+ \Big(\sum_{a \in \delta^{\text{out}}_G(v_1) \cap C} c_a + \sum_{a \in \delta^{\text{out}}_G(v_1) \cap D} d_a \Big), \text{ or} \\ f_{a_1} \coloneqq -b(v_1) &+ \Big(\sum_{a \in \delta^{\text{in}}_G(v_1) \cap C} c_a + \sum_{a \in \delta^{\text{in}}_G(v_1) \cap D} d_a \Big) \\ &- \Big(\sum_{a \in \delta^{\text{out}}_G(v_1) \cap C} c_a + \sum_{a \in \delta^{\text{out}}_G(v_1) \cap D} d_a \Big), \end{aligned}$$

depending on whether v_1 is the head or the tail of a_1. As they are no longer being needed, v_1 and a_1 can now be deleted from the spanning tree, obtaining a new, smaller tree. The procedure described above arithmetically amounts to successively calculating the flow for leaf nodes of a tree, and generating a new tree by deleting the leaf and its unique incident tree arc.

Reduced Costs

Due to the triangularity of the basis matrix, the calculation of the reduced costs of the non-basic variables for a given spanning tree structure is simplified in a way similar to what was described in the previous section. The general Simplex Algorithm finds the simplex multipliers by multiplying the

following equation by $(\Delta'_B)^{-1}$ from the right:

$$\pi^{\mathrm{T}}\Delta'_B = u_B^{\mathrm{T}}.$$

As the last column of Δ'_B contains only one nonzero entry, namely in the last row (equalling 1 or -1), the last entry of π can be obtained immediately. As the first but last column of Δ'_B has at most two nonzero entries, namely in the last two rows, and as the value of the last entry of π is already known, its second last entry can now be computed easily. Iterating this approach of backward substitution, the vector π can be determined by a number of additions and subtractions.

Choosing Entering and Leaving Arcs

If a non-tree arc e violating its optimality condition is selected as the entering arc and added to T, exactly one (undirected) cycle K is formed in T. Choose the orientation of K such that e is traversed in forward direction. Suppose that $e \in C$ (the case $e \in D$ being similar). If the value of f_e is decreased by the amount θ, then in order to maintain the balance constraints, the amount of flow has to be reduced by θ in the forward arcs of K (the arcs oriented in the same direction as K), and an additional amount of θ has to be sent through the backward arcs (the arcs oriented in the opposite direction as K). This shows that, according to Equation (1.30) on page 40, the coefficients $\overline{\delta_{a,e}}$ have to be equal to 1 for all forward arcs a of K, -1 for all backward arcs a of K, and 0 for all other arcs a of T.

The value of f_e can be decreased until the value f_a of the flow in some arc $a \in A(K)$ reaches the boundaries of its feasibility interval. Then one of the arcs now at its upper or lower bound is selected as the leaving arc l (this could possibly again be e).

Running Time

The running time of the Network Simplex Algorithm heavily depends on the details of the pivot rules used as entering and leaving arc criteria. In 1951, Dantzig implemented the Simplex Algorithm for network flows such that he achieved a running time complexity of $O(|V|^2 \cdot |A| \cdot U)$, where U is a constant depending on the arc-capacities and -demands.

Developing a polynomial-time Network Simplex Algorithm was one of the major research topics in network flow theory for quite some time. However, by now there are pivot rules that guarantee a polynomial performance of $O(|V|^2 \cdot |A|)$, which is almost as good as the Goldberg-Tarjan Algorithm.

Once again, it deserves mention that this running time, that is vastly superior when compared with the best available bounds achieved for the general Simplex Algorithm (which are non-polynomial), is mainly due to the special structure of networks and the triangularity of the node-arc incidence matrix.

4.4.2 Variants

I want to only mention that there are a few other algorithms which are closely related to the Network Simplex Algorithm and which can analogously be derived from Simplex Methods of linear programming.

The *Parametric Network Simplex Algorithm* stems from the Parametric Simplex Algorithm, whereas the *Dual Network Simplex Algorithm* is an application of the Dual Simplex Algorithm to networks. Both of these algorithms start with and maintain a solution that is possibly infeasible but always satisfies the optimality conditions. Therefore, they can be regarded as dual algorithms.

Chapter 5

Applications

Network flows are among the most widespread applied fields in all of mathematics. They can be found in fairly different contexts. I would like to start out this chapter in Section 5.1 with a few examples of inner-mathematical topics where network flow techniques have been applied advantageously. In Section 5.2, in order to round off this thesis, I give a very brief overview of the historical origins of network flow theory, as well as a résumé of three well-known examples of how this theory is applied for modelling 'real world' situations.

5.1 Combinatorial Applications

In this section, I am going to use the Max-Flow Min-Cut Theorem 2.27 to give proofs to a few well-known combinatorial min-max problems. This strategy was already introduced by Ford and Fulkerson ([8, Chapter II]). The following are problems in which the minimum of some quantity equals the maximum of some other quantity. I will approach all of these problems by constructing a digraph $D = (V, A)$ and a suitable quintuple (S, T, b, c, d), then checking that the conditions of Theorem 2.27 are satisfied, and finally examining the correspondence between (S, T, b, c, d)-flows in D and the quantity which is to be maximized, as well as between $S - T$-cuts in D and the quantity which is to be minimized.

5.1.1 Menger's Theorem

The following theorem was first proved by Menger in 1927. Ford and Fulkerson gave a proof using network flows in [8, p. 55]. Menger's Theorem exists in arc- and vertex-disjoint versions, for both digraphs and undirected graphs.

5.1 Theorem (Menger's Theorem—directed arc-disjoint version)**.** *Let $D = (V, A)$ be a digraph and $S, T \subseteq V$. Suppose $S, T \neq \emptyset$ and $S \cap T = \emptyset$. Then the maximum number of pairwise arc-disjoint $S - T$-paths equals*

$$\min_{U \in \mathcal{C}_{S,T}(D)} |\delta^{out}(U)|.$$

Proof. Set

$$\begin{aligned} b &\coloneqq \mathbf{0}_R, \\ c &\coloneqq \mathbf{1}_A, \\ d &\coloneqq \mathbf{0}_A. \end{aligned}$$

Then $f = \mathbf{0}_A \in \mathcal{F}_{S,T}^{b,c,d}(D) \neq \emptyset$. $\mathcal{C}_{S,T}(D)$ is nonempty because $S \cap T = \emptyset$. Consequently, Conditions (2.29)–(2.32) of the Max-Flow Min-Cut Theorem 2.27 are satisfied.

Clearly, capacity$(U) = |\delta^{\text{out}}(U)|$ for every $U \in \mathcal{C}_{S,T}(D)$. It therefore only remains to show that the maximum value of an (S, T, b, c, d)-flow is equal to the maximum number of pairwise arc-disjoint $S - T$-paths in D, since then an application of Theorem 2.27 completes the proof.

Suppose there are n pairwise arc-disjoint $S-T$-paths $P_1, \ldots, P_n$. For the starting flow $f = \mathbf{0}_A$, P_1 is a flow-augmenting paths, since it is an $S-T$-path in $D_f = D$. Along P_1, value$(f) = 0$ can be augmented by 1 since then the variable ε in the proof of Proposition 2.24 equals 1. This yields the updated flow

$$f_1(a) \coloneqq \begin{cases} 1 & \text{if } a \in A(P_1) \\ f(a) = 0 & \text{if } a \notin A(P_1) \end{cases}.$$

We have value$(f_1) = 1$. Now, P_2 is also an f-augmenting path in D (since it is an $S-T$-path in $D_f = D$), but because P_1 and P_2 are pairwise arc-disjoint, P_2 must also be an $S-T$-path in D_{f_1}, i.e., an f_1-augmenting path. Iterating this, a sequence $\{f_i\}_{i \in \{1,\ldots,n\}}$ of flows with value$(f_i) = i$ is obtained:

$$f_i(a) \coloneqq \begin{cases} 1 & \text{if } a \in A(P_i) \\ f_{i-1}(a) & \text{if } a \notin A(P_i) \end{cases}.$$

In fact, we have

$$f_n(a) \coloneqq \begin{cases} 1 & \text{if } a \in \bigcup_{i=1}^{n} A(P_i) \\ 0 & \text{otherwise} \end{cases}.$$

As value$(f_n) = n$, it is proved that the maximum value of an (S, T, b, c, d)-flow is not less than the maximum number of pairwise arc-disjoint $S - T$-paths in D.

Conversely, let f be an integer maximal (S, T, b, c, d)-flow in D (maximal in the set of *all* (S, T, b, c, d)-flows), the existence of which is guaranteed by Theorem 2.27. Then f assumes only the values 0 or 1 on all arcs. Set $n := \text{value}(f)$. I am going to show in a somewhat informal way how to construct n pairwise arc-disjoint $S-T$-paths in D. If $n = 0$, there is nothing to prove. If $n = \text{value}(f) > 0$, there must be some arc $a = (u, v) \in \delta^{\text{out}}(S)$ with $f(a) = 1$. As $b(v) = 0$, the unit of flow that enters v through a has to leave v through some arc. In this way, it is possible to construct walks starting in S. If, at some stage of the construction, one comes back to an already visited vertex, a cycle has been formed that would destroy the 'pathiness' of the walk. This problem can always be solved by leaving out some arcs and vertices from the cycle such that the remainder still forms a path: a cycle is formed if some vertex is visited twice during the construction of one path. This vertex can be left again through another, not yet used arc, due to the 0-balance condition. If at some stage S is reached again, the constructed path can be left out. Otherwise, the procedure is continued until T is reached. (At some stage, the path has to reach one of S or T, since the unit of flow it carries has to 'go somewhere'.) There have to be n more paths leaving S that carry a unit of flow to T than paths reentering S. This is because $\text{value}(f) = f(\delta^{\text{out}}(S)) - f(\delta^{\text{in}}(S)) = n$.

Although the technical details are left out, it should hence be plausible that in the described manner, ultimately n arc-disjoint $S - T$-paths can be constructed, which finishes the proof. □

5.1.2 König's Matching Theorem

Another field in which network flow techniques proved useful is the part of graph theory dealing with bipartite graphs. In a certain sense, digraphs are closely connected with undirected bipartite graphs. The basic terminology concerning matchings and vertex covers (that will be the subject of the next theorem) was introduced in Section 1.3.1, on page 12. I first give a central definition:

5.2 Definition. An undirected graph $B = (V, E)$ is called *bipartite* if and only if there is a partition (X, Y) of V such that every element of E can be written in the form $\{x, y\}$, where $x \in X$ and $y \in Y$.

5.3 Remark. A matrix M is called *totally unimodular* if every square submatrix of M has determinant 0, 1, or -1. Undirected bipartite graphs have totally unimodular node-arc incidence matrices, just like directed graphs. In fact, it can be shown that an undirected graph has a totally unimodular

node-arc incidence matrix if and only if it is bipartite. In this sense, directed graphs can be regarded as a generalization of bipartite graphs.

5.4 Theorem (König's Matching Theorem)**.** *Let $B = (V, E)$ be a bipartite graph. Then the maximum size of a matching in B equals the minimum size of a vertex cover in B.*

Proof. Let $\{X, Y\}$ be a partition of V such that every element of E can be written in the form $\{x, y\}$, where $x \in X$ and $y \in Y$. Define the digraph D through $V(D) := \{s\} \cup V \cup \{t\}$ (for some $s,\ t \notin V,\ s \neq t$), and $A(D) := A_1 \cup A_2 \cup A_3$ with

$$\begin{aligned} A_1 &:= \{(s, x) \mid x \in X\}, \\ A_2 &:= \{(x, y) \mid x \in X,\ y \in Y\}, \text{ and} \\ A_3 &:= \{(y, t) \mid y \in Y\}. \end{aligned}$$

Set $b := \mathbf{0}_{R(D)}$, and $d := \mathbf{0}_{A(D)}$. Let us also define a capacity function c for D:

$$c(a) := \begin{cases} 1 & \text{for } a \in A_1 \cup A_3 \\ |V| + 1 & \text{for } a \in A_2 \end{cases}.$$

Since $\mathbf{0}_A \in \mathcal{F}_{S,T}^{b,c,d}(D) \neq \emptyset$ and $s \neq t$, Conditions (2.29)–(2.32) of the Max-Flow Min-Cut Theorem 2.27 are satisfied in this setting. Now let

$$\mathcal{V}_0 := \{Z \in \mathcal{V}(B) \mid \forall y \in Z \cap Y \colon \exists x \in \delta_B(y) \colon x \notin Z\},$$

a certain set of (relatively small, or at least not 'unnecessarily' big) vertex covers of B, as well as

$$\mathcal{C}_0 := \{U \in \mathcal{C}_{s,t}(D) \mid \forall y \in Y \colon (y \in U \Leftrightarrow \exists x \in U \colon (x, y) \in A(D))\},$$

a set of certain cuts of D with relatively small capacity. Note $\delta^{\text{out}}(U) \cap A_2 = \emptyset$ for all $U \in \mathcal{C}_0$. I claim that the mapping

$$g \colon \mathcal{C}_0 \to \mathcal{V}_0,\ U \mapsto \{x \in X \mid x \notin U\} \cup \{y \in Y \mid y \in U\}$$

is a bijection. In order to verify this, the first thing to prove is $g(U) \in \mathcal{V}_0$. Suppose $g(U) \in 2^V \setminus \mathcal{V}$. Choose $\{x, y\} \in E$ with $x \in X$, $y \in Y$, and $\{x, y\} \cap g(U) = \emptyset$. This means $x \in U$, $y \notin U$. As this would imply $(x, y) \in \delta^{\text{out}}(U)$, and as $(x, y) \in A_2$ this is impossible. So $g(U) \in \mathcal{V}$. The other way $g(U)$ could fail to be $\in \mathcal{V}_0$ is that there is a $y \in g(U) \cap Y$ with $x \in g(U)$ for all x with $x \in \delta_B(y)$, or equivalently for all x with $y \in \delta_D^{\text{out}}(x)$. But this would mean $y \in U$, although there is no $x \in U$ such that $(x, y) \in A(D)$, which contradicts the definition of $\mathcal{C}_0$.

g is bijective, because it has an inverse, given by

$$g^{-1}\colon \mathcal{V}_0 \to \mathcal{C}_0,\ Z \mapsto \{s\} \cup (X \backslash Z) \cup (Y \cap Z).$$

We have $g^{-1}(Z) \in \mathcal{C}_0$: obviously $g^{-1}(Z) \in \mathcal{C}_{s,t}(D)$. Furthermore, let $y \in Y$. Suppose $y \in g^{-1}(Z)$. Then $y \in Z$ and consequently, by definition of $\mathcal{V}_0$, there is an $x \in \delta_B(y)$ with $x \notin Z$. This implies $x \in g^{-1}(Z)$, as desired. If on the other hand $y \notin g^{-1}(Z)$, then $y \notin Z$. As Z is a vertex cover, it must be $x \in Z$ whenever $(x, y) \in A(D)$. But then $x \notin g^{-1}(Z)$ for all x with $(x, y) \in A(D)$. This shows $g^{-1}(Z) \in \mathcal{C}_0$. The fact that $g \circ g^{-1} = \mathrm{id}_{\mathcal{C}_0}$ and $g^{-1} \circ g = \mathrm{id}_{\mathcal{V}_0}$ is easy to check.

Note that $\mathcal{C}_0$ contains the $s-t$-cuts of minimum capacity: $A_1 = \delta^{\mathrm{out}}(s)$ is an $s-t$-cut (since $s \in \{s\}$ and $t \notin \{s\}$) of capacity $c(A_1) = |X|$, which is $< c(a)$ for all $a \in A_2$. Therefore, no minimal $s-t$-cut can have a set of outarcs that contains an edge from A_2. Moreover, $\mathcal{V}_0$ contains the minimum-size vertex covers: let $Z \in \mathcal{V}$ containing a $y \in Y$ with $x \in Z$ for all $x \in \delta_B(y)$. Then $Z' \coloneqq Z \backslash \{y\}$ is a vertex cover of size $|Z| - 1$. Hence, as $|g(U)| = |c(\delta^{\mathrm{out}}(U))| = \mathrm{capacity}(U)$ for all $U \in \mathcal{C}_0$, the minimum size of a vertex cover in B equals the minimum capacity of an $s-t$-cut in D.

Similarly, there is a correspondence between matchings in B and integer $s-t$-flows in D: consider the mapping

$$h\colon \mathcal{M}(B) \to \mathcal{F}^{b,c,d}_{s,t}(D) \cap \mathbb{Z}^A,\ M \mapsto f.$$

Herein $f = h(M)$ is the flow

$$f(a) \coloneqq \begin{cases} 1 & \text{if } a = (s, x) \text{ for some } x \in V(M) \\ 1 & \text{if } a = (x, y) \text{ for some } \{x, y\} \in M \\ 1 & \text{if } a = (x, t) \text{ for some } x \in V(M) \\ 0 & \text{in all other cases} \end{cases}.$$

To see that this mapping is a bijection, note that clearly $d \leq f \leq c$ and $f\colon A \to \mathbb{Z}$. The balance-condition (2.3) with $b = \mathbf{0_R}$ is also satisfied: for the vertices v that are endpoints of some edge in M (and hence of exactly one edge in M, because M is a matching), we have $f(\delta^{\mathrm{in}}_D(v)) = f(\delta^{\mathrm{out}}_D(v)) = 1$. For $v \in V(B) \backslash V(M)$ we have $f(\delta^{\mathrm{in}}_D(v)) = f(\delta^{\mathrm{out}}_D(v)) = 0$. For s and t, there is nothing to show. Consequently, we indeed have $f = h(M) \in \mathcal{F}^{b,c,d}_{s,t}(D) \cap \mathbb{Z}^A$.

The inverse of h is

$$h^{-1}\colon \mathcal{F}^{b,c,d}_{s,t}(D) \cap \mathbb{Z}^A \to \mathcal{M}(B),\ f \mapsto E \cap E(\mathrm{supp}(f)).$$

Here $\mathrm{supp}(f)$ denotes the set of arcs a of D such that $f(a) \neq 0$. Recall that $E(A_0)$, for some arc-set A_0, is the set of the corresponding unordered versions

of the pairs of A_0. The fact that $h^{-1}(f)$ is a matching is a consequence of the distribution of capacities in D: each $x \in X$ can be endpoint to at most one edge $\{x, y\} \in E$ with $f((x, y)) \neq 0$ since $f(\delta_D^{\text{in}}(x)) = f((s, x)) \leq 1$ and since $b(x) = 0$. Similarly, each $y \in Y$ can be endpoint to at most one edge $\{x, y\} \in E$ with $f((x, y)) \neq 0$ since $f(\delta_D^{\text{out}}(y)) = f((y, t)) \leq 1$ and since $b(y) = 0$. Finally, the equations $h \circ h^{-1} = \text{id}_{\mathcal{M}(B)}$ and $h^{-1} \circ h = \text{id}_{\mathcal{F}_{s,t}^{b,c,d}(D) \cap \mathbb{Z}^A}$ are easy to check.

By Theorem 2.27, $\mathcal{F}_{s,t}^{b,c,d}(D) \cap \mathbb{Z}^A$ contains a maximum flow (i.e., a flow maximal in the set of *all* (s, t, b, c, d)-flows). Due to our construction, we have

$$\text{value}(h(M)) = h(M)(\delta_D^{\text{out}}(s)) - h(M)(\delta_D^{\text{in}}(s)) = |M| - 0 = |M|.$$

Therefore, the maximum size of a matching in B equals the maximum value of an $s-t$-flow in D. The statement of the theorem is now just the Max-Flow Min-Cut Theorem 2.27 applied to D. □

5.5 Remark. Papadimitriou and Steiglitz gave a complexity bound for finding a maximum size matching by finding a maximum network flow, as in the above proof. They used their max-flow algorithm mentioned in Section 4.2.4. It turns out that this algorithm performs particularly well in the case of networks with unit capacities[1] in which every vertex has indegree 1 or 0, or outdegree 1 or 0. In that way, they derive a running time bound of $O(|V|^{\frac{1}{2}} \cdot |A|)$.

5.1.3 Hall's Marriage Theorem

Yet another example of an interesting combinatorial application of network flow techniques can be found in the theory of transversals. What a transversal is, is clarified by the following definition:

5.6 Definition. Let X be a finite set and $\mathcal{X} = (X_1, X_2, \ldots, X_k)$ a family of subsets of X. A *partial transversal of* $\mathcal{X}$ is a set $Z \subseteq \bigcup_{i=1}^{k} X_i$ such that for every $i \in \{1, \ldots, k\}$ we have $|Z \cap X_i| \leq 1$. So a partial transversal is a subset of X that contains exactly one element from each of $X_{i_1}, \ldots, X_{i_r}$, where $\{i_1, \ldots, i_r\}$ is some subset of $\{1, \ldots, n\}$.

The next theorem is cited from [21, p. 380] (for a 'dual' formulation, see [13, p. 211]), whereas the idea for the proof stems from the construction

[1] The value of a maximum flow in the network constructed in the proof does obviously not change if the large arc capacities on A_2 are changed to 1. Those capacities were only chosen in that way to make the argumentation in the proof easier.

given in [12, p. 56f.], for the 'classical' version of Hall's Theorem, a slightly less general situation. (Usually though, the theorem as formulated below is proved by reduction to the classical one.) Hall was the first to prove his version of the theorem, 1935. It was dubbed "marriage theorem" by Weyl in 1949, because of the interpretation he gave of it.

5.7 Theorem (Defect Form of Hall's Marriage Theorem)**.** *Let $k, n \in \mathbb{N}$, and let $X = \{x_1, \dots, x_n\}$ be a finite set. Let $\mathcal{X} = (X_1, X_2, \dots, X_k)$ be a family of subsets of X. Then the maximum size of a partial transversal of $\mathcal{X}$ is equal to*

$$\min_{Y \subseteq X} \Big(|X \backslash Y| + |\{i \mid X_i \cap Y \neq \emptyset\}| \Big).$$

Proof. I am going to construct a digraph and choose (s, t, b, c, d) such that the Max-Flow Min-Cut Theorem 2.27 can be applied. Choose two new elements $s \neq t$ that are not in X. Set

$$\begin{aligned}
V &\coloneqq \{s\} \cup X \cup \mathcal{X} \cup \{t\},\\
A_1 &\coloneqq \{(s, x_i) \mid i \in \{1, \dots, n\}\},\\
A_2 &\coloneqq \{(x_i, X_j) \mid x_i \in X_j,\ i \in \{1, \dots, n\},\ j \in \{1, \dots, k\}\},\\
A_3 &\coloneqq \{(X_j, t) \mid j \in \{1, \dots, k\}\},\\
A &\coloneqq A_1 \cup A_2 \cup A_3,\\
D &\coloneqq (V, A),\\
b &\coloneqq \mathbf{0}_R,\\
c(a) &\coloneqq \begin{cases} 1 & \text{for } a \in A_1 \cup A_3 \\ n+1 & \text{for } a \in A_2 \end{cases},\\
d &\coloneqq \mathbf{0}_A.
\end{aligned}$$

Note that Conditions (2.29)–(2.32) of the Max-Flow Min-Cut Theorem 2.27 are satisfied, since $f = \mathbf{0}_A \in \mathcal{F}_{s,t}^{b,c,d}(D)$ and $\{s\} \in \mathcal{C}_{s,t}(D)$. We have $\delta^{\text{in}}(s) = \emptyset$ and $c(\delta^{\text{out}}(s)) = c(A_1) = n$.

A minimal $s-t$-cut cannot have an arc from A_2 as one of it outarcs, since then $\{s\}$ would be a smaller cut (i.e., a cut a smaller capacity). Moreover, if a cut U has the arcs $(s, x_{i_1}), \dots, (s, x_{i_r}) \in A_1$ as outarcs (i.e., $x_{i_1}, \dots, x_{i_r} \notin U$), as well as an arc $(X_{j_0}, t) \in A_3$ (i.e., $X_{j_0} \in U$) for an $X_{j_0} \subseteq \{x_{i_1}, \dots, x_{i_r}\}$, in other words, there is no $x \in X \backslash \{x_{i_1}, \dots, x_{i_r}\}$ with $(x, X_{j_0}) \in A$, then a smaller cut can be obtained from U by leaving out X_{j_0}. So if looking for possible candidates for a minimal cut, it suffices to regard cuts constructed as follows: choose $Y \subseteq X$ and set $U \coloneqq \{s\} \cup Y \cup \{X_i \in \mathcal{X} \mid X_i \cap Y \neq \emptyset\}$. The result are only cuts that have no 'unnecessary' arcs from A_3, and

no 'expensive' arcs from A_2 as outarcs. As for these cuts $\text{capacity}(U) = c(\delta^{\text{out}}(U)) = |\delta^{\text{out}}(U)| = |X \backslash Y| + |\{X_i \in \mathcal{X} \mid X_i \cap Y \neq \emptyset\}|$, it is thus shown that

$$\min_{Y \subseteq X} \Big(|X \backslash Y| + |\{i \mid X_i \cap Y \neq \emptyset\}| \Big) = \min_{C \in \mathcal{C}_{s,t}} (D)\, \text{capacity}(U).$$

If I can now also show that the value of a maximum flow equals the maximum size of a partial transversal of $\mathcal{X}$, then an application of Theorem 2.27 proves the statement of the theorem.

To this end, let first Z be a partial transversal of $\mathcal{X}$. It can be written as $Z = \{z_1, \ldots, z_r\}$ with $z_i \in X_{j_i}$ for $i \in \{1, \ldots, r\}$, and with $j_1, \ldots, j_r \in \{1, \ldots, k\}$ all distinct. Define an integer flow f of value $|Z| = r$ by

$$f(a) \coloneqq \begin{cases} 1 & \text{if } a = (s, z_i),\ i \in \{1, \ldots, r\} \\ 0 & \text{for all other arcs in } A_1 \\ 1 & \text{if } a = (z_i, X_{j_i}),\ i \in \{1, \ldots, r\} \\ 0 & \text{for all other arcs in } A_2 \\ 1 & \text{if } a = (X_{j_i}, t),\ i \in \{1, \ldots, r\} \\ 0 & \text{for all other arcs in } A_3 \end{cases}.$$

Let us check that f is indeed an (s, t, b, c, d)-flow: capacity-conditions (2.4) and (2.5) are clearly satisfied. The fact that Z contains at most one element from each X_i (i.e., $h \neq i \Longrightarrow j_h \neq j_i$) ensures that the balance-condition (2.3) is valid, too. Furthermore, we have

$$\text{value}(f) = f(\delta^{\text{out}}(s)) - f(\delta^{\text{in}}(s)) = f(\delta^{\text{out}}(s)) = \sum_{a \in \delta^{\text{out}}(s)} f(a) = \sum_{i=1}^{r} 1 = r.$$

If one is conversely given any integer (s, t, b, c, d)-flow f, a partial transversal Z of $\mathcal{X}$ with $|Z| = \text{value}(f)$ can be obtained as follows. There are exactly $v \coloneqq \text{value}(f)$ indices $j_1, \ldots, j_v \in \{1, \ldots, n\}$ with $f((s, x_{j_i})) = 1$. Let $Z \coloneqq \{x_{j_1}, \ldots, x_{j_v}\}$. Then clearly $|Z| = \text{value}(f)$.

Because of the balance-condition (2.3), applied to the vertices of the set X, there are vertices $X_{k_1}, \ldots, X_{k_v} \in \mathcal{X}$ of D with $(x_{j_i}, X_{k_i}) \in A_2$ (namely the endpoints of the v arcs from A_2 for which $f((x_{j_i}, X_{k_i})) = 1$). This implies $x_{j_i} \in X_{k_i}$ for $i \in \{1, \ldots, v\}$ and hence $Z \subseteq \bigcup_{j=1}^{k} X_j$. Because of (2.3) applied to the vertices of the set $\mathcal{X}$, we have

$$\forall h,\ i \in \{1, \ldots, v\} \colon h \neq i \Longrightarrow X_{k_h} \neq X_{k_i}.$$

But this implies $|Z \cap X_j| \leq 1$ for $j \in \{1, \ldots, k\}$, i.e., Z is indeed a partial transversal of $\mathcal{X}$.

A combination of these two observations shows that the maximum value of an (s, t, b, c, d)-flow equals the maximum size of a partial transversal of $\mathcal{X}$, and the proof is finished. □

5.2 'Real World' Applications

Network flow techniques have not only been applied diversely within the realms of mathematical theory, but are also encountered regularly in different areas of the rest of the world. I want to use this section to first give a brief overview of the origins that have led to the development of the complex and extensive theory of which some parts we had a glimpse at in this thesis.

5.2.1 Historical Notes

The Max-Flow Min-Cut question was originally posed to Ford and Fulkerson by T.E. Harris and General F.S. Ross ([8, p. 1]). They issued a secret report for the U.S. Air Force, entitled *Fundamentals of a Method for Evaluating Rail Net Capacities* and dated 24 October 1955 (cf. [21, p. 166]). This report was downgraded to "unclassified" as late as 1999. The main objective of the report was not to find a flow of maximum value through a given railway network, but rather the dual problem: interdicting it as efficiently as possible (i.e., finding a minimum capacity cut; or in other words: how one has to destroy as few railroad tracks as possible, such that no traffic is possible anymore). In the report a heuristic "flooding technique" due to Boldyreff was described and proposed for usage. The technique did not deliver optimum results for every possible case. Harris and Ross also applied the technique as an example to a simplified model of the Soviet and East European Railroads. The model was an (undirected) graph with 44 vertices (each of which modeled some subdivision of the railroad system) and 105 edges. For the data, they referred to several secret C.I.A. reports. They achieved an estimate for the flow capacity from the Soviet Union to East European "satellite" countries, along with a cut of equal capacity, which they indicated as the 'bottleneck' of the railway system.

It is probably not surprising from today's perspective that the Max-Flow Problem and the Max-Flow Min-Cut Theorem arose from a 'practical problem', considered that the theory of network flows is one of the most applied fields in all of mathematics. Fortunately, since the dark days of its inception, a great number of somewhat more constructive interpretations of networks have been developed and successfully applied. The fields in which these applications can be found are diverse: transportation of goods, scheduling,

assignment, route planing, data security, public transport systems, communication networks, operations research/management sciences, waste water systems, electric engineering, macroeconomics, supply/demand models, dieting, ...

Still, the *literature* on applications is far not as numerous as literature on the theory behind them. A real wealth of examples is for instance found in the comprehensive book *Network Flows. Theory, Algorithms, and Applications* by Ahuja, Magnanti, and Orlin ([1]). Besides a thorough treatment of the underlying theory, it presents around 150 applications of network flow-related theory from all imaginable areas of human life. Mainly various problems from economics that can be modelled by network flows are described in [22] and [7]. Other collections of applications include [24] and [15].

To illustrate the various approaches, I would like to give a few brief, but hopefully illuminating, examples of how network flow theory is employed to model situations arising in economics or other areas.

5.2.2 Supply/Demand

The following is one of the most intuitive interpretations of network flows. It exists in many variations, examples for which can among others be found in [1, p. 169f.], or [9, p. 233].

Supply/Demand Problem
Suppose a certain good is available at some sea ports and desired at others. The stock available and amount desired at every port is known, as well as the maximum amount which can be shipped from some ports to others (e.g., within a fixed period of time). Can all demands be met? Can all the stock available be sold?

It appears to be very suitable to model this situation by a network as follows:

$$\begin{aligned} V &:= \text{a finite set of vertices, such that there is exactly one vertex for every port,} \\ S &:= T := \emptyset, \\ A &:= V \times V, \\ b(v) &:= \text{the demand of the port represented by the vertex } v \in V \text{ minus the amount available at } v, \\ c((u,v)) &:= \text{the maximum amount of the good that can be shipped from the port represented by } u \text{ to the port represented by } v, \\ d &:= \mathbf{0}_A. \end{aligned}$$

For $f \in \mathbb{R}^A$, interpret $f((u,v))$ as the amount being shipped from the port represented by u to the port represented by v. Then it is easy to see that there exists a possibility to ship goods from some ports to some others in a way such that in the end there is exactly as great an amount of the good at every port as desired (and at the same time such that all the available stock is used up) if and only if there is an (S,T,b,c,d)-flow for D.

Moreover, there exists a possibility to ship goods from some ports to some others in a way such that in the end there is *at least* as great an amount of the good at every port as desired if and only if there is an (S,T,b,c,d)-preflow[2] for D (in which case there might be unused stock remaining).

A related but more complex problem arises when the balance-vector b is replaced by a finite family of balance-vectors $b_1,\ldots,b_n\colon V \to \mathbb{R}$. This models the situation when one simultaneously wishes to satisfy demands of n different goods, subject to available supplies and total capacity constraints. This is an example of a so-called *multicommodity flow*. Multicommodity flows are much more difficult to handle then the one-commodity flows described in this piece of work. In the models considered so far, there were possibly several source- or sink-nodes, but it did not matter from which source to which sink the flow was transported. As discussed earlier, this is not essentially different from the situation with only one source and one sink. Multicommodity flows can be thought of having additional requirements specifying from which sources how much flow should be transported to which sinks. This problem turns out to be far more complex.

5.2.3 The Transportation Problem

As a variation to the Supply/Demand Problem described in the previous section, a per-unit cost $k_{(u,v)}$ could be assigned to each arc $a=(u,v)$. Then another natural question is how to minimize the total costs such that all demands are met (assuming that this is possible)? This is exactly the Linear Min-Cost Flow Problem as described in Section 4.1. The total costs for a feasible solution (i.e., for $f \in \mathcal{F}_{S,T}^{b,c,d}$) are

$$\mathrm{cost}(f) = \sum_{(u,v)\in A} k_{(u,v)} f((u,v)).$$

In the following slightly varied bipartite form, this problem is usually referred to as the *Hitchcock Transportation Problem*. It was formulated by F.L. Hitchcock in 1941. A quite extensive discussion in terms of linear programming is for example [10, Chapter 9]. Other variations of it, illustrating

[2] cf. Definition 4.9, on p. 101

different facets, can be found in countless books on applied graph theory, network flows, and linear programming, such as [2, Section 3.12], [6, Chapter 11], [9, p. 232f.], [23, Chapter 6 and 7/§6], [8, Chapter III], [5, Kapitel 14–15 and 20], [24, Chapter 2], [17, Section 1.3.5], ... An actual case study can be found in [10, Section 12.4].

Hitchcock Transportation Problem

Input
- A finite set of factories together with their respective supply production in units
- A finite set of retail stores together with their respective demands in units
- The maximum number of units that can be transported from every factory to each retail store
- Per-unit costs for transportation from each factory to every retail store

Output A transportation schedule subject to the transportational capacities that satisfies all demands from the available supplies and minimizes the total costs

In terms of networks, this problem looks as follows:

$$
\begin{aligned}
X &\coloneqq \text{a finite set of vertices, one for every factory,}\\
Y &\coloneqq \text{a finite set of vertices, one for every retail store,}\\
V &\coloneqq X \cup Y,\\
S &\coloneqq T \coloneqq \emptyset,\\
A &\coloneqq X \times Y,\\
b(v) &\coloneqq \begin{cases} \text{the unit production of factory } v & \text{if } v \in X \\ (-1) \text{ times the unit demand of retail store } v & \text{if } v \in Y \end{cases},\\
c((x,y)) &\coloneqq \text{the maximal number of units that can be transported from the}\\
&\qquad \text{factory represented by } x \text{ to the retail store represented by } y,\\
d &\coloneqq \mathbf{0}_A,\\
k_{(x,y)} &\coloneqq \text{the costs of transporting one unit from the factory represented}\\
&\qquad \text{by } x \text{ to the retail store represented by } y.
\end{aligned}
$$

As mentioned before, this is an instance of the Linear Min-Cost Flow Problem. As the resulting graph D is bipartite, an interesting special case results

for $|X| = |Y|$ and

$$b(v) := \begin{cases} 1 & \text{if } v \in X \\ -1 & \text{if } v \in Y \end{cases}.$$

Finding an integral min-cost flow in this network is the same as finding a minimum perfect matching[3] for D (i.e., a matching $M = \{a_1, \ldots, a_{|X|}\}$ covering all vertices of D such that $\sum_{i=1}^{|X|} k_{a_i} a_i$ is minimal).

This last version of the Hitchcock transportation problem yields, if interpreted appropriately, yet another famous problem of classical combinatorial optimization:

5.2.4 The Assignment Problem

Now regard the vertices in X from the previous section as persons looking for a job or machines available for one, and the vertices from Y as the jobs that are vacant or need to be processed. $k_{(x,y)}$ could be interpreted as a measure of proficiency or efficiency of x for job y. Here smaller values of $k_{(x,y)}$ correspond to higher proficiency/efficiency (alternatively, this correspondence could be reversed, resulting in a maximization problem). Finding an integral feasible flow minimizing the total costs is known as the *Assignment Problem*:

ASSIGNMENT PROBLEM

Input
- n persons looking for a job
- n jobs
- Numbers $k_{(x,y)}$ measuring the aptitude of person x for job y

Output An assignment of every person to a job such that overall optimality is achieved

Studies of this problem are as numerous as for the Hitchcock transportation problem: [2, Section 3.12], [6, Chapter 10], [13, p. 273, Chapter 14], [4, Section 7.1], [3, Chapters 4, 5], [22, Section 6.4], [17, Section III.2.3], ...

[3] matching was defined in Section 1.3.1, on page 12

Bibliography

[1] Ravindra K. Ahuja, Thomas L. Magnati, and James B. Orlin, *Network Flows. Theory, Algorithms, and Applications*, Prentice-Hall, Upper Saddle River, New Jersey, 1993. pages 6, 7, 29, 55, 76, 78, 87, 89, 96, 97, 99, 107, 124

[2] Jørgen Bang-Jensen and Gregory Gutin, *Digraphs: Theory, Algorithms and Applications*, Springer Monographs in Mathematics, Springer-Verlag, London Berlin Heidelberg, 2001. pages 6, 7, 48, 55, 97, 99, 126, 127

[3] Dimitri P. Bertsekas, *Linear Network Optimization: Algorithms and Codes*, The MIT Press, Cambridge (Massachusetts), London, 1991. pages 6, 26, 127

[4] Andreas Brandstädt, *Graphen und Algorithmen*, Leitfäden und Monographien der Informatik, B. G. Teubner, Stuttgart, 1994. pages 7, 55, 127

[5] George B. Dantzig, *Lineare Programmierung und Erweiterung*, Ökonometrie und Unternehmensforschung, no. II, Springer-Verlag, Berlin Heidelberg, 1966. pages 26, 27, 29, 43, 126

[6] Ulrich Derigs, *Programming in Networks and Graphs*, Lecture Notes in Economics and Mathematical Systems, no. 300, Springer-Verlag, Berlin Heidelberg New York, 1988. pages 6, 26, 126, 127

[7] Salah E. Elmaghraby, *Some Network Models in Management Science*, Lecture Notes in Economics and Mathematical Systems. Economics, Computer Science, Information and Control, no. 29, Springer-Verlag, Berlin Heidelberg, 1970. pages 77, 78, 124

[8] L.R. Ford, Jr. and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, New Jersey, 1962. pages 4, 48, 55, 76, 77, 95, 96, 115, 123, 126

[9] Martin Grötschl, László Lovász, and Alexander Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Algorithms and Combinatorics, no. 2, Springer-Verlag, Berlin Heidelberg, 1988. pages 6, 26, 43, 69, 124, 126

[10] G. Hadley, *Linear Programming*, Addison-Wesley Series in Industrial Management, Addison-Wesley Publishing Company, Reading (Massachusetts), London, 1962. pages 26, 29, 125, 126

[11] Harro Heuser, *Lehrbuch der Analysis, Teil 1*, 13. ed., B. G. Teubner, Stuttgart/Leipzig/Wiesbaden, 2000. pages 57

[12] Konrad Jacobs, *Einführung in die Kombinatorik*, Walter de Gruyter, Berlin New York, 1983. pages 7, 121

[13] Dieter Jungnickel, *Graphs, Networks and Algorithms*, second ed., Algorithms and Computation in Mathematics, vol. 5, Springer, Berlin Heidelberg New York, 2005. pages 6, 7, 69, 95, 97, 120, 127

[14] Bernhard Korte and Jens Vygen, *Combinatorial Optimization. Theory and Algorithms*, 2nd ed., Algorithms and Combinatorics, no. 21, Springer-Verlag, Berlin Heidelberg, 2002. pages 6, 26, 43, 96, 97, 100, 104

[15] Christoph Mandl, *Applied Network Optimization*, Academic Press Inc., London, 1979. pages 124

[16] Kurt Marti and Detlef Grötel, *Einführung in die lineare und nichtlineare Optimierung*, Physica-Verlag, Heidelberg, 2000. pages 26, 29

[17] L. George Nemhauser and Laurence A. Wolsey, *Integer and Combinatorial Optimization*, Wiley-Interscience series in discrete mathematics and opimization, John Wiley and Sons, New York Chichester Brisbane Toronto Singapore, 1988. pages 6, 26, 29, 126, 127

[18] Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization. Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982. pages 6, 26, 29, 43, 44, 94, 96, 100

[19] RAND, *Rand Corporation Provides Objective Research Services and Public Policy Analysis*, http://www.rand.org/. pages 4

[20] Walter Rudin, *Analysis*, R. Oldenbourg Verlag, München, 1998. pages 26, 57

[21] Alexander Schrijver, *Combinatorial Optimization. Polyhedra and Efficiency*, vol. A, Algorithms and Combinatorics, no. 24, Springer-Verlag, Berlin Heidelberg New York, 2003. pages 6, 7, 26, 28, 43, 44, 48, 55, 60, 71, 76, 77, 90, 91, 96, 97, 99, 104, 120, 123

[22] H. Steckhan, *Güterströme in Netzen*, Lecture Notes in Economics and Mathematical Systems. Operations Research, no. 88, Springer-Verlag, Berlin Heidelberg New York, 1973. pages 77, 124, 127

[23] James K. Strayer, *Linear Programming and its Applications*, Undergraduate Texts in Mathematics, Springer-Verlag, New York, 1989. pages 26, 105, 126

[24] H. Walther, *Anwendungen der Graphentheorie*, Friedr. Vieweg and Sohn, Braunschweig/Wiesbaden, 1978. pages 4, 78, 124, 126

[25] Wikipedia, *Rand - Wikipedia, the Free Encyclopedia*, http://en.wikipedia.org/wiki/RAND. pages 4

Curriculum Vitae

Timon Thalwitzer

Contact

Email: TimonThalwitzer@gmx.net
Homepage: www.myspace.com/TimonThalwitzer

Personal Details

Gender: Male
Date of birth: 28th of January, 1983
Place of birth: Vienna, Austria
Present Citizenship: Austria

Education

10/2001-10/2008	Diplomstudium Mathematik at the University of Vienna, Austria. Main interests algebra and graph theory
since 2001	Diplomstudium Musikwissenschaft at the University of Vienna, Austria. Main interests history and theory of jazz
09/2006-06/2007	Jazz Drums at the Gustav-Mahler Konservatorium in Vienna, Austria
09/1993-06/2001	Secondary School at the GRG 13 Wenzgasse in Vienna, Austria
09/1989-06/1993	Primary School

Vienna, November 11, 2008
Ort, Datum

Timon Thalwitzer