2 | Preliminaries

In this chapter, we introduce the basic definitions, on which the remainder of this thesis relies. In particular, we settle the basic notation and define the underlying model of computation. Moreover, we introduce the most common network flow problems that will be used and extended throughout the thesis and define the notion of approximation algorithms. In the very most cases, we comply with standard notations and definitions, so the familiar reader may skip parts of this chapter at will.

2.1 Basic Notation

Throughout this thesis, we denote by $\mathbb{R}$ ($\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$) the set of *real* (*rational*, *integral*, *natural*) numbers. We denote the corresponding subsets of $\mathbb{R}$ and $\mathbb{Q}$ that contain all non-negative numbers by $\mathbb{R}_{\geqslant 0}$ and $\mathbb{Q}_{\geqslant 0}$, respectively. Similarly, we write $\mathbb{R}_{>0}$ and $\mathbb{Q}_{>0}$, respectively, for the subsets of all positive numbers. The set $\mathbb{N}$ of natural numbers does not contain zero while $\mathbb{N}_{\geqslant 0}$ does. For two sets A and B, we use the notation $A \subseteq B$ ($A \supseteq B$) to denote that A is a subset (superset) of B. If A is a proper subset (superset) of B, we write $A \subset B$ ($A \supset B$). Moreover, we denote the union (intersection) of the two sets A and B by $A \cup B$ ($A \cap B$). Finally, we write $\emptyset$ to denote the empty set.

We use the notation $A \in B^{m \times n}$ to denote that A is a matrix with m rows and n columns, each of which contains elements from the set B. Moreover, we denote by A_{ij} the element in the i-th row and the j-th column of A and use $A_{i\cdot}$ and $A_{\cdot j}$ to denote the i-th row vector and the j-th column vector of A, respectively.

For a function $f\colon \mathbb{N}_{\geqslant 0} \to \mathbb{N}_{\geqslant 0}$, the set $\mathcal{O}(f(n))$ contains all functions $g\colon \mathbb{N}_{\geqslant 0} \to \mathbb{N}_{\geqslant 0}$ with the property that there are constants n_0 and c such that $g(n) \leqslant c \cdot f(n)$ for each $n \geqslant n_0$. Similarly, the set $\Omega(f(n))$ contains all functions $g\colon \mathbb{N}_{\geqslant 0} \to \mathbb{N}_{\geqslant 0}$ with the property that there are constants n_0 and c such that $g(n) \geqslant c \cdot f(n)$ for each $n \geqslant n_0$. The intersection of $\mathcal{O}(f(n))$ and $\Omega(f(n))$ is denoted by $\Theta(f(n)) := \mathcal{O}(f(n)) \cap \Omega(f(n))$.

Finally, we denote the *logarithm* of the number a to the basis b by $\log_b a$. Whenever the basis of the logarithm is omitted, it can be assumed to be 2 without loss of generality. We denote the natural logarithm of a by $\ln a$ (i.e., the logarithm to the basis e, where

M. Holzhauser, *Generalized Network Improvement and Packing Problems*, DOI 10.1007/978-3-658-16812-4_2

e is Euler's number). Moreover, we let $\operatorname{sgn}\colon \mathbb{R} \to \{-1, 0, 1\}$ denote the *sign function*, which returns -1, 0, or 1 depending on whether the given argument is negative, zero, or positive.

2.2 Theory of Computation

In this section, we give a short definition of the computational model that is used throughout this thesis and introduce the tools that will be used to measure the computational complexity of the upcoming problems. Again, we will thereby stick to basic definitions and notations such that the experienced reader may skip parts of this section at will. In-depth treatments of the upcoming topics can be found in (Garey and Johnson, 1979; Grötschel et al., 1993; Motwani and Raghavan, 1995; Papadimitriou, 1994; Blum et al., 1989).

Instance Encoding

Throughout this thesis, we assume that the instances of the problems that investigate are encoded by using a "reasonable" *encoding scheme* into a string over some alphabet Σ. The set of all strings over Σ will be denoted by Σ^*. The *encoding length* or *size* of a problem instance I, denoted by $|I|$, is then the length of such a string. Although the number of symbols in Σ will be of no great importance for our results (as long as $|\Sigma| \geqslant 2$), we will assume a *binary encoding* $\Sigma = \{0, 1\}$. As a result, an integer of value n will, e.g., have a size of $\lceil \log_2 |n| + 1 \rceil$ bits in any instance (Grötschel et al., 1993; Garey and Johnson, 1979).

Computational Models

The complexity results stated throughout this thesis are based on the *random access machine (RAM) model*[1234], which is an alternative to the classical *Turing machine* (Papadimitriou, 1994) that is capable of an infinite set of registers, each of which can store one integer of arbitrary size and sign. A random access machine supports a set of instructions such as direct and indirect addressing of registers, jumping and branching, comparisons, as well as arithmetic operations such as addition, subtraction, multiplication and division of numbers. More precisely, we will stick to the *unit-cost RAM*, in which each of these operations can be performed in constant time, independent of the size of the involved integers (in contrast to the *log-cost RAM*, which accounts for

the size of the operands). This simplification leads to a "too powerful" model in comparison to the Turing machine since we can generate very large numbers too quickly using repeated multiplication. Nevertheless, an equivalence between the models (up to a logarithmic factor) holds in case that the encoding length of the involved integers is polynomially bounded by the encoding length of the instance (Motwani and Raghavan, 1995). Since the numbers that are used in the Chapters 3 – 6 fulfill this property, we can use the unit-cost RAM without loss of generality. The *(worst case) time complexity* or *(worst case) running time* of an algorithm is a function $f\colon \mathbb{N}_{\geqslant 0} \to \mathbb{N}_{\geqslant 0}$ such that $f(|I|)$ denotes the maximum number of instructions that are executed by the random access machine at an input of size $|I|$ (Grötschel et al., 1993).

Note that a random access machine is only capable of handling integral (or, more general, rational) numbers. In Chapter 7, however, we will be confronted with real numbers, which may have an infinite (explicit) representation. For this reason, we will stick to the *Blum-Shub-Smale model (BSS model)* in the corresponding chapter, which is basically a random access machine that is extended by the possibilities to store real rather than integral numbers in its registers and to evaluate rational functions on the register contents at unit cost (Blum et al., 1989; Blum, 1998).

Decision Problems

A *decision problem* Π is a problem that, for each instance I, either gives the answer Yes or No. That is, the problem Π can be seen as a subset of $\Sigma^* \times \{0,1\}$ such that, for each instance I, either $(I,0)$ or $(I,1)$ is contained in Π (Grötschel et al., 1993). We say that a decision problem Π has a *time complexity of* $f(|I|)$ or *is solvable in* $f(|I|)$ *time* if there is an algorithm with time complexity $g(|I|)$ for $g \in \mathcal{O}(f(|I|))$ that decides whether or not some problem instance I of size $|I|$ is a Yes-instance of Π. A decision problem Π is *solvable in (weakly) polynomial time* if it has a time complexity of $p(|I|)$ for some polynomial $p\colon \mathbb{N}_{\geqslant 0} \to \mathbb{N}_{\geqslant 0}$. Moreover, if Π has a time complexity of $p(m)$ and uses $q(|I|)$ space, where m denotes the number of integers that occur in the problem instance (regardless of the magnitude of the integers) and p and q are two polynomials, it is *solvable in strongly polynomial time*. Conversely, if Π is not solvable in polynomial time but has a time complexity that is polynomial in $|I|$ and the absolute value of the integers in I, we call it *solvable in pseudo-polynomial time* (Grötschel et al., 1993; Garey and Johnson, 1979).

The class $\mathcal{P}$ consists of all decision problems that are solvable in (weakly or strongly, but not pseudo-) polynomial time. The class $\mathcal{NP}$ consists of all decision problems that are *verifiable* in polynomial time, i.e., the set of problems Π for which there is a decision

problem $\Pi' \in \mathcal{P}$ such that, for each YES-instance I of Π and for some polynomial p', there is a *certificate* $x \in \Sigma^*$ with $|x| \leqslant p'(|I|)$ and $((I, x), 1) \in \Pi'$ (Grötschel et al., 1993).[1]

A decision problem Π is *polynomial-time reducible* to a decision problem Π' if there is a function t that transforms an instance I of Π into an instance $I' := t(I)$ of Π' in polynomial time such that I is a Yes-instance of Π if and only if I' is a Yes-instance of Π'. A decision problem Π is said to be $\mathcal{NP}$*-hard* if every problem in $\mathcal{NP}$ is polynomial-time reducible to Π. If, in addition, $\Pi \in \mathcal{NP}$, the decision problem is said to be $\mathcal{NP}$*-complete*. If the decision problem remains $\mathcal{NP}$-complete even if the numbers that occur in each problem instance I are polynomially bounded by $|I|$, the problem is *strongly* $\mathcal{NP}$*-complete*. Otherwise, we call the decision problem *weakly* $\mathcal{NP}$*-complete*. Unless $\mathcal{P} = \mathcal{NP}$, an $\mathcal{NP}$-complete decision problem is not solvable in polynomial time (Garey and Johnson, 1979).

Optimization Problems

In many problems that occur in practice, one is not only interested in the information whether or not a given instance is a YES-instance (as in the case of decision problems) but wants to determine the somewhat best solution among all feasible solutions to the underlying problem with respect to some quality measurement. In an *optimization problem*, the aim is to determine an *optimal solution* x out of the set of *feasible solutions* X that maximizes (minimizes) some *objective function* $z\colon X \to \mathbb{R}$ in case of a *maximization problem* (*minimization problem*). The definition of time complexity as well as the notion of weakly, strongly, and pseudo-polynomial time solvability can then be applied to the case of an optimization problem without further ado. For the sake of convenience, we usually say that an optimization problem is *weakly or strongly* $\mathcal{NP}$*-hard (*$\mathcal{NP}$*-complete) to solve* if the corresponding decision problem that asks if there is a feasible solution $x \in X$ with $z(x) \geqslant k$ ($z(x) \leqslant k$) in case of a maximization problem (minimization problem) for some given value k is weakly or strongly $\mathcal{NP}$-complete to solve, respectively.

In a k*-criteria optimization problem*, the aim is to optimize a *set* of objective functions $z^{(j)}\colon X \to \mathbb{R}$ for $j \in \{1, \ldots, k\}$ over the set X of feasible solutions, each of which can either be a maximization or a minimization objective. The *objective space* Z is then given by $Z := \left\{ \left(z^{(1)}(x), \ldots, z^{(k)}(x) \right)^T : x \in X \right\}$. The notion of an "optimal solution" becomes ambiguous in case of a k-criteria optimization problem if $k \geqslant 2$. Instead, we call a solution $x \in X$ *efficient* if, for every other solution $x' \neq x \in X$, there is an index $j \in \{1, \ldots, k\}$ such that $z^{(j)}(x') < z^{(j)}(x)$ ($z^{(j)}(x') > z^{(j)}(x)$) in case of a maximization (min-

1 An alternative definition of the class $\mathcal{P}$ ($\mathcal{NP}$) is that it contains all problems that are solvable in polynomial time on a *deterministic* (*non-deterministic*) Turing machine (Garey and Johnson, 1979).

imization) objective. The set $\mathcal{P} := \left\{ \left(z^{(1)}(x), \dots, z^{(k)}(x) \right)^T : x \in X \text{ and } x \text{ is efficient} \right\} \subseteq Z$ is then called the *pareto frontier* of the optimization problem. The corresponding set of efficient solutions is sometimes called pareto frontier as well.

2.3 Graph Theory

Throughout this thesis, we consider *directed (multi-)graphs* (or simply *graphs*) $G = (V, E)$, which are induced by a finite *node set* V and a finite multiset E, called *edge set*, that contains ordered pairs of $V \times V$. The elements in V and E are referred to as *nodes* and *edges*, respectively. We denote the cardinalities of V and E by n and m, respectively. At some points, we restrict our considerations to *simple graphs* in which E is a set rather than a multiset. If the edge set contains two-element subsets of V rather than ordered pairs of $V \times V$, the graph is called a *undirected*.

For each edge $e = (v, w) \in E$, we call v the *starting node* and w the *end node* of e and say that e *heads from* v *to* w. Similarly, we call e both an *outgoing edge* of v and an *ingoing edge* of w or simply say that e *leaves* v and *reaches* w. Two nodes $v \in V$ and $w \in V$ are furthermore called *adjacent* if there is an edge $e \in E$ that heads from v to w or vice versa. Likewise, we call a node v and an edge e *incident* if v is the starting or the end node of e. Moreover, we call two edges e and e' *adjacent* if they are both incident to the same node v.

For each node $v \in V$, we denote by $\delta^+(v) := \{e \in E : e \text{ leaves } v\}$ and $\delta^-(v) := \{e \in E : e \text{ reaches } v\}$ the set of *outgoing edges* and *ingoing edges* of v, respectively. Accordingly, we call $|\delta^+(v)|$ and $|\delta^-(v)|$ the *out-degree* and *in-degree* of node v, respectively. For a set $V' \subseteq V$, we write $\delta^+(V')$ ($\delta^-(V')$) to denote the set of edges $e = (v, w) \in E$ with $v \in V'$ and $w \notin V'$ ($v \notin V'$ and $w \in V'$).

For a given graph $G = (V, E)$, we call each graph $H = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ a *subgraph* of G. For a given subset $V' \subseteq V$ of the node set, we call $G[V']$ the *subgraph induced by* V', which consists of all nodes $v \in V'$ and all edges $e \in E \cap (V' \times V')$. Similarly, for a subset $E' \subseteq E$ of the edge set, the graph $G[E']$ denotes the *subgraph induced by* E', i.e., the graph with edge set E' and the node set V' that contains all end nodes of edges in E'.

A sequence $P := (e_1, \dots, e_k)$ of edges in E in which the end node of e_i and the starting node of e_{i+1} coincide for each $i \in \{1, \dots, k-1\}$ is called a *path (of length* k*)*. If v_0 denotes the starting node of e_1 and v_k the end node of e_k, then P is also referred to as a v_0-v_k-path. Moreover, we say that P *connects* v_0 *and* v_k and that v_k *is reachable from* v_0 *(via* P*)*. If all of the involved nodes are distinct, we call P a *simple path*. A path $C := (e_1, \dots, e_k)$

in which the end node of e_k and the starting node of e_1 coincide is called a *circuit (of length* k*)*. Furthermore, if the paths $P_1 := (e_1, \dots, e_{k-1})$ and $P_2 := (e_2, \dots, e_k)$ are both simple, we call C a *simple cycle (of length* k*)simple cycle* or just *cycle*. We call a (simple) path, circuit, or cycle *undirected* if we obtain a corresponding (simple) path, circuit, or cycle by reverting the direction of one or more of the contained edges.

For two distinguished nodes $s, t \in V$, an s-t-*cut* (S, T) is a partition of the node set into two disjoint sets S and T such that $s \in S$ and $t \in T$. We denote by $\delta^+(S)$ ($\delta^-(S)$) the set of *forward edges* (*backward edges*) in the cut. Usually, we also use the set of forward edges $\delta^+(S)$ to refer to the s-t-cut (S, T).

We call a graph $G = (V, E)$ *connected* if, for each two nodes $v_1, v_2 \in V$, there is a (possibly undirected) v_1-v_2-path P in G. Accordingly, we call G *strongly connected* if there is both a directed path from v_1 to v_2 and a directed path from v_2 to v_1 for every pair of nodes $v_1, v_2 \in V$. Each maximal subgraph (with respect to the edge set) of G that is (strongly) connected is called a *(strongly) connected component*.

A graph $G = (V, E)$ with the property that its node set V can be partitioned into two sets V_1 and V_2 such that $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$ is called *bipartite*. A graph G is called *acyclic* if it does not contain cycles. A *topological sorting* of a graph $G = (V, E)$ with node set $V = \{v_1, \dots, v_n\}$ is a sequence $(v_{i_1}, \dots, v_{i_n})$ with $i_j \neq i_l$ for $j \neq l$ (i.e., a ordering of the node set) such that $i_j < i_l$ for each edge $e = (v_{i_j}, v_{i_l}) \in E$. As it is well-known, a graph is acyclic if and only if it has a topological sorting (cf., e.g., (Ahuja et al., 1993)).

A *tree* is a connected graph that does not contain undirected cycles. A graph G in which each connected component is a tree is called a *forest*. We call a tree $T = (V, E)$ *rooted* if there is some distinguished *root node* $r \in T$. In such a rooted tree T, we say that a node $v \in V$ is on *level* k if it is connected to the root node by an undirected path P_v of length k. Every node $w \neq v$ on this path P_v is called an *ancestor* of v while v itself is a *successor* of w. If, in addition, there is an edge from v to w or from w to v in T, we call v a *child node* of w and w the *parent node* of v. Each node v that has no children is called a *leaf node* or simply *leaf* while every other node of T (including the root node) is called an *inner node*. If every inner node of the tree has exactly two children, we call T a *binary* tree. Furthermore, if every node is reachable from the root node via a directed path, we call the tree an *out-tree*. Conversely, if the root node is reachable from every node via a directed path, we call the tree an *in-tree*. The subgraph of a tree $T = (V, E)$ that is induced by a node $v \in V$ and all of its successors is called a *subtree of* T. Finally, for a graph $G = (V, E)$, we call a tree T a *spanning tree of* G if it is a subgraph of G and contains all nodes in V.

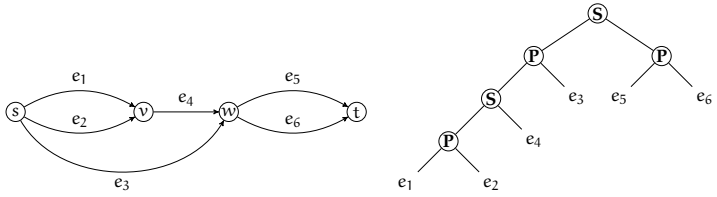
Another important class of graphs that is used throughout this thesis is the class of *series-parallel graphs*. Each series-parallel graph $G = (V, E)$ is associated with a *source* $s \in V$ and a *sink* $t \in V$ and can be recursively defined as follows:

Single edge: Each graph that consists of a single edge $e = (s, t)$ is series-parallel with source s and sink t (denoted by $G = e$).

Parallel composition: For two series-parallel graphs G_1, G_2 with sources s_i and sinks t_i, $i \in \{1, 2\}$, the graph G that is obtained by identifying s_1 with s_2 and t_1 with t_2 is series-parallel with source $s_1 = s_2$ and sink $t_1 = t_2$ (denoted by $G = G_1 \mid G_2$).

Series composition: For two series-parallel graphs G_1, G_2 with sources s_i and sinks t_i, $i \in \{1, 2\}$, the graph G that is obtained by identifying t_1 with s_2 is series-parallel with source s_1 and sink t_2 (denoted by $G = G_1 \circ G_2$).

In particular, note that series-parallel graphs are acyclic and connected according to the above definition. If, additionally, in each series composition of G_1 and G_2 at least one of G_1 or G_2 consists of a single edge, the graph is called *extension-parallel*. A *decomposition tree* T of a series-parallel graph G is a binary tree in which the leaves correspond to single edges of G and each inner node v either corresponds to a series or a parallel composition of the two series-parallel graphs that are induced by the leaves of the two subtrees of v. Such a decomposition tree T of a series-parallel graph G with m edges and n nodes contains m leaves, $n - 2$ inner nodes that correspond to series compositions, and $m - n + 1$ inner nodes that correspond to parallel compositions. Moreover, such a decomposition tree can be constructed from a given series-parallel graph in $\mathcal{O}(m)$ time (cf. Valdes et al. (1982)). A series-parallel graph G and a corresponding decomposition tree T are depicted in Figure 2.1. Note that this graph is not extension-parallel while the subgraph that is induced by the nodes $\{s, v, w\}$ is.



(a) Series-parallel graph G. (b) Decomposition tree T.

Figure 2.1: A series-parallel graph (left) and a possible decomposition tree (right). Inner nodes representing parallel compositions (series compositions) are denoted by the letter **P** (**S**).

Throughout this thesis, we assume that each graph $G = (V, E)$ is given in the *adjacency-list representation*, in which a list $\mathrm{Adj}(v)$ is connected with each node $v \in V$ containing the outgoing edges $\delta^+(v)$ of v.

2.4 Network Flow Problems

In this section we give a short introduction to the field of *network flow problems*. In general, such problems aim at finding the "best" way to send some amount of a commodity from one point in a network to another one. Thereby, we will concentrate on the definitions and complexities of the four possibly most important network flow problems, namely the *shortest path problem*, the *maximum flow problem*, the *minimum cost flow problem*, and the *maximum generalized flow problem*. For an in-depth treatment of these topics, we refer the reader to Ahuja et al. (1993) and Wayne (1999).

In its most general form, each node $v \in V$ in a network flow problem is associated with an integral number b_v that represents the *supply* of the corresponding node. Moreover, each edge $e \in E$ has both a non-negative *lower capacity* $l_e \geqslant 0$ and a (possibly infinite) *upper capacity* or just *capacity* $u_e \geqslant l_e$. A *pseudoflow* is a function $x\colon E \to \mathbb{R}$ that assigns a value $x_e := x(e) \in [l_e, u_e]$ to each edge, which represents the amount of goods that are transported along e from the starting node to the end node of e. For a pseudoflow x, the *excess of a node* $v \in V$ if given as $\mathrm{excess}_x(v) := \sum_{e \in \delta^-(v)} x_e - \sum_{e \in \delta^+(v)} x_e$ and describes the difference of the incoming amount of flow and the outgoing amount of flow at v. Similarly, the *imbalance of a node* $v \in V$ is given as $\mathrm{excess}_x(v) + b_v$ and describes the deviation of the excess from the demand $-b_v$ of the node. If a pseudoflow x fulfills $\mathrm{excess}_x(v) \geqslant -b_v$ at each $v \in V$, it is called a *preflow*. Moreover, if x fulfills the *flow conservation constraint* $\mathrm{excess}_x(v) = -b_v$ at each node $v \in V$, the pseudoflow is called a *feasible flow* or simply *flow*. The *flow value* $\mathrm{val}(x)$ of a pseudoflow x is defined as the amount of flow that remains at the sink, i.e., $\mathrm{val}(x) := \mathrm{excess}_v(t)$. Note that, in order to allow feasible flows, we require that $\sum_{v \in V} b_v = 0$. However, as it is well-known, we can assume without loss of generality that $l_e = 0$ and that u_e is finite for each $e \in E$. Furthermore, we may assume that there is a distinct *source* $s \in V$ as well as a distinct *sink* $t \in V$ such that $b_s > 0$, $b_t < 0$, and $b_v = 0$ for each $v \in V \setminus \{s, t\}$ (Ahuja et al., 1993). As it is well known that the polyhedron described by the above constraints is *integral*, we can assume without loss of generality that the optimal flow is integral as long as the input data is integral as well (Ahuja et al., 1993).

For a flow x in a network that is based on a graph $G = (V, E)$, the *residual network* $G(x)$ contains at most two edges for each $e = (v, w) \in E$ in the original graph G: Unless

$x_e = u_e$, there is some *forward edge* $e^{(1)}$ heading from v to w with a (positive) capacity of $u_{e^{(1)}} := u_e - x_e$. Moreover, unless $x_e = 0$, a *backward edge* $e^{(2)}$ heads from w to v with a (positive) capacity of $u_{e^{(2)}} := x_e$.

Shortest Paths

For a given graph with edge labels c_e for each $e \in E$, the *shortest path problem* in general aims at finding a (directed) path P between two nodes $v, w \in V$ with a minimum *length* $c(P) := \sum_{e \in P} c_e$ among all such paths. In the *single-source shortest path problem*, one seeks to find a shortest path from some node $v \in V$ to every other node $w \in V \setminus \{v\}$. Conversely, in the *all-pairs shortest path problem*, the task is to determine a shortest path between every pair of nodes in V. In particular, the shortest path problem is in fact a network flow problem since it can be seen as the problem of shipping one unit of a good from one point in the network to each of the $n - 1$ other points in the *cheapest* possible way. As we will see later, the problem can be seen as a special case of the more general minimum cost flow problem.

At present, the best bound for the single-source shortest path problem in a simple graph with non-negative lengths c_e is given by

$$SP(m, n, C) \in \mathcal{O}\left(\min\left\{m + n \log n, m \log\log C, m + n\sqrt{C}\right\}\right),$$

with $C := \max_{e \in E} c_e$, where the corresponding bounds are due to Fredman and Tarjan (1987), Johnson (1981), and Ahuja et al. (1990), respectively. The best strongly polynomial time bound is consequently given by $SP(m, n) \in \mathcal{O}(m + n \log n)$. Moreover, in an acyclic graph, the single-source shortest path problem can be solved in $\mathcal{O}(m + n)$ time (Ahuja et al., 1993). Algorithms for the all-pairs shortest path problem will be investigated in Chapter 4.

Maximum Flows

In the *maximum flow problem*, the task is to determine a flow that sends the maximum possible amount of flow from the source s to the sink t of the network without violating any edge capacity. More precisely, the aim is to find a feasible flow x with maximum flow value $\mathrm{val}(x)$ among all feasible flows or, equivalently, to determine the largest value $b_s = -b_t$ that allows a feasible flow x. Hence, we are able to leave out the flow conservation constraints for s and t and can maximize over $-b_t = \mathrm{excess}_x(t)$

instead. Stated as a linear program, we then obtain the following formulation for the maximum flow problem:

$$\max \sum_{e \in \delta^-(t)} x_e - \sum_{e \in \delta^+(t)} x_e \tag{2.1a}$$

$$\text{s.t.} \sum_{e \in \delta^-(v)} x_e - \sum_{e \in \delta^+(v)} x_e = 0 \qquad \text{for all } v \in V \setminus \{s, t\}, \tag{2.1b}$$

$$0 \leqslant x_e \leqslant u_e \qquad \text{for all } e \in E. \tag{2.1c}$$

Note that we can transform each instance of the maximum flow problem on a multigraph into an equivalent instance on a simple graph in linear time *without* increasing the number of nodes and edges by simply aggregating the capacities of all parallel edges between two nodes into the capacity of a new artificial edge and deleting the previous edges in $\mathcal{O}(m)$ time.

The maximum flow problem is probably the network flow problem with the longest history of steady improvements. The first algorithm for the maximum flow problem was introduced in 1956 by Ford and Fulkerson (1956) with a pseudo-polynomial running time of $\mathcal{O}(nmU)$ for $U := \max_{e \in E} u_e$. The underlying idea of repeatedly sending flow on s-t-paths with positive capacity in the residual network was later independently refined by Dinic (1970) and Edmonds and Karp (1972), resulting in strongly polynomial running times of $\mathcal{O}(n^2m)$ and $\mathcal{O}(nm^2)$, respectively. Another class of algorithms, called *push-relabel algorithms*, in which flow is augmented along single edges rather than full s-t-paths, was introduced by Karzanov (1974) and Goldberg and Tarjan (1986). It resulted in running times of $\mathcal{O}(nm \log_{m/(n \log n)} n)$ and $\mathcal{O}(\min\{n^{2/3}, m^{1/2}\} \cdot m \log(n^2/m) \log U))$ due to King et al. (1994) and Goldberg and Rao (1998), respectively. In 2013, after there was no significant progress in the field of maximum flows for about 15 years, Orlin (2013) was able to give an affirmative answer to the long standing open question whether there is an algorithm with a running time in $\mathcal{O}(nm)$ by combining the ideas of King et al. (1994) and Goldberg and Rao (1998) with a new algorithm for sparse graphs. At present, the best bound for the maximum flow problem is given by

$$MF(m, n, U) \in \mathcal{O}\left(\min\left\{\min\{n^{2/3}, m^{1/2}\} \cdot m \log(n^2/m) \log U, nm\right\}\right)$$

due to Goldberg and Rao (1998) and Orlin (2013), respectively. The best strongly polynomial time bound is given by $MF(m, n) \in \mathcal{O}(nm)$ due to Orlin (2013).

Minimum Cost Flows

The *minimum cost flow problem* is the most fundamental network flow problem, particularly since it subsumes both the shortest path and the maximum flow problem and is strongly related to the maximum generalized flow problem that is described in the next subsection (Truemper, 1977). For given *edge costs* $c_e \in \mathbb{Z}$ for each $e \in E$, we seek to obtain a flow x that minimizes the total *flow costs* $c(x) = \sum_{e \in E} c_e \cdot x_e$ among all feasible flows. Note that we obtain the shortest path problem by setting $b_s := 1$ and $b_t := -1$ (if the flow is integral, which we can assume without loss of generality as described above) and the maximum flow problem by setting $c_e = -1$ for $e \in \delta^-(t)$ and $c_e = 0$ for $e \in E \setminus \delta^-(t)$.

Throughout this thesis, we usually restrict our considerations to the case that no flow value is prescribed, i.e., we drop the flow conservation constraints both for the source and the sink of the network (similar to the case of *minimum cost circulations*, cf. (Ahuja et al., 1993)). This assumption, however, does not constrain the capabilities of the model, which can be seen as follows: On the one hand, it can be shown that every minimum cost flow with a flow value of F decomposes into *some* flow with value F (which can be found by a maximum flow computation in $\mathcal{O}(nm)$ time as shown before) and a minimum cost circulation in the residual network of this flow. On the other hand, we can model a desired flow value of $F \geqslant 1$ by adding an artificial edge $\overline{e}$ heading from the original sink t to a new artificial sink t' with capacity $u_{\overline{e}} := F$ and cost $c_{\overline{e}} := -(mCU + 1)$. Since the absolute cost of any flow is bounded by mCU, the cost of a minimum cost flow in this transformed network is then smaller than $-(F-1)(mCU+1)$ if and only if it has a value of F and is minimum among all such flows. Note that this transformation is only connected with no loss of generality when talking about exact algorithms. For approximation algorithms (see Section 2.5), the transformation has an influence on the approximation guarantee of an algorithm. Nevertheless, as one usually needs to make assumptions in order to guarantee a non-negative or non-positive objective function value for a proper definition of an approximation guarantee, we restrict our considerations on the flow model described above.

Hence, we can formulate the minimum cost flow problem as a linear program as follows:

$$
\begin{aligned}
\min \sum_{e \in E} c_e \cdot x_e & \\
\text{s.t.} \sum_{e \in \delta^-(v)} x_e - \sum_{e \in \delta^+(v)} x_e &= 0 && \text{for all } v \in V \setminus \{s,t\}, \\
0 \leqslant x_e &\leqslant u_e && \text{for all } e \in E.
\end{aligned}
$$

Note that the total flow costs are non-positive for each minimum cost flow in this setting since the all-zero flow is always feasible.

Similarly to the case of the maximum flow problem, the minimum cost flow problem has a long history of steady research and improvements. At present, the best bound $\mathrm{MCF}(m, n, C, U)$ for the minimum cost flow problem is given by

$$\mathrm{MCF}(m, n, C, U) \in \mathcal{O}(\min\{nm \cdot \log(n^2/m) \log(nC), nm \cdot \log\log U \log(nC), \\ m \log n \cdot (m + n \log n)\}).$$

These bounds are due to Goldberg and Tarjan (1987), Ahuja et al. (1992), and Orlin (1993), respectively. The best strongly polynomial time bound $\mathrm{MCF}(m, n)$ is achieved by Orlin's *enhanced capacity scaling algorithm* (cf. Ahuja et al. (1993) and Orlin (1993)) and is given by $\mathrm{MCF}(m, n) \in \mathcal{O}(m \log n \cdot (m + n \log n))$.

Note that each of these bounds only applies to *simple* graphs. Clearly, we can convert every multigraph into a simple graph by replacing each edge $e = (v, w)$ by two edges $e_1 = (v, v')$ and $e_2 = (v', w)$ for an artificial node v'. This transformation increases the number of edges from m to $2m$ and the number of nodes from n to $n + m$. However, in Chapter 4, we will see that we can avoid this transformation for the case of the enhanced capacity scaling algorithm, which in turn yields a time bound of $\mathcal{O}(m \log m \cdot (m + n \log n))$ for the problem on multigraphs.

Maximum Generalized Flows

The is an extension of the traditional maximum flow problem in which the implicit assumption of flow being conserved when it traverses an edge is dropped. Instead, the flow x_e that enters some edge e will have a value of $\gamma_e \cdot x_e$ for some *gain factor* $\gamma_e > 0$ when it leaves that edge. For different values of these gain factors, one can model effects like evaporation in a gas pipeline (if $\gamma_e < 1$) or money exchanges among different currencies (Wayne, 1999). Clearly, the resulting polyhedron is no longer integral, so we cannot assume a maximum generalized flow to be integral as well. The maximum generalized flow problem can be formulated as a linear program as follows:

$$\begin{aligned}
\max \quad & \sum_{e \in \delta^-(t)} \gamma_e \cdot x_e - \sum_{e \in \delta^+(t)} x_e & & \\
\text{s.t.} \quad & \sum_{e \in \delta^-(v)} \gamma_e \cdot x_e - \sum_{e \in \delta^+(v)} x_e = 0 & & \text{for all } v \in V \setminus \{s, t\} \\
& 0 \leqslant x_e \leqslant u_e & & \text{for all } e \in E.
\end{aligned}$$

Note that the capacity of an edge bounds the flow that *enters* an edge (rather than the flow that *leaves* the edge). At present, the best bound for the maximum generalized flow problem is given by

$$\mathrm{MGF}(n, m, B) \in \mathcal{O}\left(\min\left\{nm \cdot (m + n \log n) \log B), m^5\right\}\right)$$

if the capacities are integers between 1 and B and the gain factors are fractions of numbers between 1 and B. The first time bound is due to (Radzik, 2004) while the strongly polynomial time bound is due to Végh (2013), who recently showed that the generalized maximum flow problem can be solved in strongly polynomial time $\mathrm{MGF}(n, m) \in \mathcal{O}(m^5)$.

2.5 Approximation Algorithms

For a maximization (minimization) problem Π with non-negative objective function z, an algorithm is called an *approximation algorithm with performance guarantee* $\alpha \in [1, \infty)$ or simply *α-approximation algorithm* if, for each instance I of Π, it computes a feasible solution x with objective value $z(x) \geqslant \frac{1}{\alpha} \cdot z(x^*)$ ($z(x) \leqslant \alpha \cdot z(x^*)$) in polynomial time, where x^* denotes an optimal solution of I.

An algorithm that receives an instance I of a maximization (minimization) problem Π and a real number $\varepsilon \in (0, 1)$ as its input is called a *polynomial-time approximation scheme (PTAS)* if, on input (I, ε), it computes a feasible solution x with objective value $z(x) \geqslant (1 - \varepsilon) \cdot z(x^*)$ ($z(x) \leqslant (1 + \varepsilon) \cdot z(x^*)$) with a running time that is polynomial in the encoding size $|I|$ of I. If this running time is additionally polynomial in $\frac{1}{\varepsilon}$, the algorithm is called a *fully polynomial-time approximation scheme (FPTAS)*. Similarly, for a k-criteria optimization problem Π with objective functions $z^{(j)}$ for $j \in \{1, \ldots, k\}$, we call an algorithm a *k-criteria fully polynomial-time approximation scheme (k-criteria FPTAS)* if, on input (I, ε) with $I \in \Pi$ and $\varepsilon \in (0, 1)$, it computes a feasible solution x with objective value $z^{(j)}(x) \geqslant (1 - \varepsilon) \cdot z^{(j)}(x^*)$ for each maximization objective $z^{(j)}$ and $z^{(j)}(x) \leqslant (1 + \varepsilon) \cdot z^{(j)}(x^*)$ for each minimization objective $z^{(j)}$.

The notion of a k-criteria FPTAS is strongly related to the concept of *ε-approximate pareto frontiers*: For some instance I of a k-criteria optimization problem, the ε-approximate pareto frontier $\mathcal{P}(\varepsilon)$ is a subset of the feasibility set X such that, for each $x \in X$, there is a point $x_P \in \mathcal{P}(\varepsilon)$ that fulfills $z^{(j)}(x_P) \geqslant \frac{1}{1+\varepsilon} \cdot z^{(j)}(x)$ for each maximization objective and $z^{(j)}(x_P) \leqslant (1 + \varepsilon) \cdot z^{(j)}(x)$ for each minimization objective (cf. (Papadimitriou and Yannakakis, 2000)). Intuitively, the ε-approximate pareto frontier is a "sufficiently good" approximation of the real pareto frontier $\mathcal{P}$ with respect to the precision

parameter ε. Note that, for $\varepsilon \in (0,1)$, the fact that $z^{(j)}(x_p) \geqslant \frac{1}{1+\varepsilon} \cdot z^{(j)}(x)$ also implies that $z^{(j)}(x_p) \geqslant (1-\varepsilon) \cdot z^{(j)}(x)$ as in the definition of an FPTAS above.

Finally, an optimization problem Π is said to be $\mathcal{NP}$-*hard to approximate* if the existence of an approximation algorithm with a specific performance guarantee α would imply that an $\mathcal{NP}$-hard decision problem is solvable in polynomial time.

http://www.springer.com/978-3-658-16811-7

Generalized Network Improvement and Packing Problems

Holzhauser, M.

2016, XVI, 213 p. 26 illus., Softcover
ISBN: 978-3-658-16811-7