Introduction

Urban traffic congestion is increasing day by day. Examples can be found around the world. Growing cities and increasing population doubled the traffic volume in the last two decades in Europe and North America and an even higher rise has to be expected for the urban regions in Asia or South America in the next years. In Germany, the population travels about 1000 billion kilometers every year, and 85 percent of this distance is covered by individual motor car traffic [25, 56]. The city of São Paulo, Brazil, is famous for its record-breaking traffic jams. The 20 million inhabitants own about six million vehicles. On an evening in June 2009, the traffic congestion the city reached a new record of 293 kilometers in total [47]. In August 2010, there was a 60-mile, nine-day traffic jam near Beijing, China, that even made headlines in Europe.

Traffic congestion causes delays which add up to huge costs for society and business. The urban mobility report 2009 [136] states a total loss of 4.2 billion hours and 87.2 billion dollars for the 439 urban areas in the United States in only one year. Wasted fuel of 2.8 billion gallons, noise, and pollution accumulate. A huge problem has to be solved.

But what can mathematics do to support the quest for stress-free, environment-friendly, and safe traveling? In this thesis we will have a close look at two familiar systems for traffic control–*guideposts* and *traffic signals*. If you do not like traffic jams you are invited to read on and find out how an optimization of guideposts and traffic signals can be used to direct and improve inner-city traffic flow.

Guideposts. Guideposts have been used for a long time. They provide guidance, especially in unfamiliar regions. With increasing mobility, we cannot imagine traveling without guideposts and we can find them everywhere. Even modern GPS-based satellite navigation systems can be seen as small, virtual guideposts inside our cars. Everything seems clear–just follow the guideposts!

But we missed an important question. Where should we install these guideposts? And what consequences arise from our choice?

Assume we want to find a certain point of interest in an unfamiliar network. Fortunately, this network is equipped with guidepost pointing towards our destination wherever a routing decision has to be made. Further, we may assume that these guideposts point in an unique direction, i.e. they exclude all but one road at each intersection. Nothing would be more confusing than two guidepost naming the same destination but pointing in two different directions. Most likely, other traffic participants with the same destination will follow these guideposts, too. If we meet one of them, she will make the same routing decisions just like us. Thus, we will travel on the same route until we reach our destination. Consequently, a group of road users starting at the same origin will share the same path, even if there would exist several alternatives. The capacity of this path limits the amount of traffic participants that can reach the common destination. With road users starting all over the network, a bad choice of guideposts may lead to congestion, although the traffic flow in the network is far away from the network's capacity at free route choice.

Traffic signals. With increasing car traffic, traffic signals managing the right of way at intersections became more and more important. However, the *red* traffic light seems to be dominant. But sometimes, we arrive at a traffic signal and it switches to *green* just in time, so that we can go on without stopping. And once in a blue moon[1], we get even four or five green lights in a row.

Such a traffic signal coordination is a difficult task. Of course, coordinating one road in one direction is rather easy, but with traffic in the opposite direction or traffic in a whole street network it becomes considerably harder.

Even more, changing the coordination also means changing travel times. After a while road users will learn about the fastest routes in the network and they will switch to these routes. This new distribution of traffic in the network may completely disturb our fine-tuned coordination.

Obviously, guideposts and traffic signals are important tools for controlling traffic and traffic control is the backbone in the management of traffic flows in our cities. The optimal use of these signals is essential when we are going to resolve traffic congestion. However, it is sometimes not even clear what 'optimal' means in this context. Guideposts are often installed with respect to the shortest distance towards the destination. Their influence on congestion is poorly studied. In contrast, traffic signal coordination has been investigated for a long time and many approaches and models have been proposed. But these models also recommend various definitions of optimality. The two most common objectives are *minimizing the delay/waiting time of vehicles facing red lights* and *minimizing the number of stops*. Furthermore, the majority of the approaches reveal some deficits like unrealistic modeling of inner-city traffic flows or no guarantee for an optimal solution.

Contribution

In this thesis, we tackle traffic congestion with the help of network flow theory from two sides. First, we advance guideposts from a theoretical point of view and introduce *confluent flows*. A flow is called confluent if the flow uses at most one outgoing arc at each node. Unlike previous results we consider heterogeneous arc capacities. We will focus on $\mathcal{NP}$-hardness results for maximum confluent flows, an approximation algorithm for graphs with treewidth bounded by a constant k, and polynomial time-algorithms for special graph classes.

Second, we advance traffic signals. Since this discussion is actually a practical one – most results presented in this part are an outcome of the ADVEST project that emerged

[1] A *blue moon* refers to the third full moon in a season with four full moons. A season with four full moons is very rare, this happens only once every 2 or 3 years.

between BTU Cottbus, TU Berlin, TU Braunschweig, and PTV AG[2] – we start our contribution with a new model for the simultaneous optimization of traffic signal coordination and traffic assignment. This combined approach accounts the feedback between red lights and route choices. We answer the time dependance of traffic signals by a cyclically time-expanded network. This time expansion will also allow capturing several other characteristics of inner-city traffic like platoons of cars and exact arrival times of these platoons at the intersections. Still, viewing inner-city traffic as a periodic process limits the time horizon of the expansion and leads to a compact formulation of the problem as a mixed-integer program. Solving the MIP yields a guarantee or at least bounds for the optimal solution. We investigate our approach with the help of real-world data and state-of-the-art simulation tools.

Outline of the thesis

In Chapter 1 we will fix the notation and terminology and present basic definitions. We assume the reader to be familiar with the basic concepts in graph theory, complexity theory as well as linear programming. However, for later reference and as a short refreshing of knowledge we recall some of the most important facts. For additional information we refer to [2, 66, 100, 122, 147].

In Chapter 2 we will derive the concept for flows in networks with guideposts. For that, we will introduce *confluent flows* and conclude some basic properties, e.g. the underlying tree structure. We will study complexity result for both the *transshipment* and the *maximum flow* variant. We also present *arc-confluent flows* and discuss *cuts* for confluent flows.

Afterwards, we present *polynomial time algorithms* for restricted graph classes, e.g. trees, planar graphs with at most k terminals on the boundary, and graphs without $K_{2,3}$ as a minor in Chapter 3. The relation between confluent flows and trees will lead to a pseudo-polynomial time solution for maximum confluent flows on graphs with *treewidth* bounded by a constant k. We use this result to develop a *fully polynomial time approximation scheme* (FPTAS) for confluent flows on this kind of graphs.

Due to the various approaches for traffic signal optimization we start with a short survey on this topic in Chapter 4. We will use this survey to make the reader familiar with concepts in traffic engineering and with terms related to traffic signals. We will also discuss the advantages and disadvantages of the considered approaches to motivate our new model. Herewith, the ground for the next two chapters should be prepared. Additional information can be found, e.g., in [69, 141].

In Chapter 5 we use the concept of dynamic flows and the periodicity of traffic signals to develop a cyclically time-expanded network. The model is completed by modeling intersections, traffic signals and traffic assignment. As a main result of this chapter we show how this model can be used to optimize traffic signal coordination and traffic assignment simultaneously. Aiming for a realistic modeling we also discuss the conse-

[2]PTV AG is a traffic planning company from Karlsruhe, Germany. It is well known for its traffic planning and simulation software VISUM and VISSIM.

quences of our approach to travel times and link performance in detail and derive further properties of the model.

Finally, Chapter 6 is designated for the simulation of inner-city traffic and the practical evaluation of the proposed model. We introduce the reader to two traffic simulation tools, namely VISSIM and MATSim. In detail, we consider the real-world inner-city networks of Cottbus, Braunschweig, Portland, and Denver. In particular, we emphasize the advantages of our simultaneous optimization of signal coordination and traffic assignment by comparing to a decomposed successive version of our approach.

About this thesis

A lot of results in this thesis were obtained during the ADVEST project, granted by the German ministry of education and research (BMBF). This also reflects in the thesis. First, some results were already published, see [53, 52, 98, 96, 97]. Second, the aims of the project lead to different kinds of results. On the one hand confluent flows were studied theoretically and are better understood now. But due to the combinatorial complexity practical applicability is – in the moment – poor. One the other hand, in a more experimental approach, a new model for simultaneous traffic signal coordination and traffic assignment was created, implemented, improved and tested with help of simulation tools. Hereby, a model of high practicability was developed, but it is difficult to prove the impact of the model also mathematically. Hence, the first part of the thesis will perhaps be more interesting for readers who focus on combinatorial optimization. The second part may more appeal to readers who are interested in the modeling of real world problems.

1 Basic Definitions and Notation

In this chapter we introduce and fix the basic notation for this thesis. Many fields of discrete mathematics are touched. First, we introduce the graph notation and we present some classical graph problems that we will refer to later. Due to the wide area of graph theory this description cannot be complete. For an introduction to graph theory we suggest, e.g., [152] or [49]. A good textbook on network flows is, for example, [2]. Good textbooks, covering network flows and other combinatorial optimization strategies, are [36, 100, 138]

Furthermore, we fix the notations for algorithms, complexity and approximation. Again, we can only give a short overview. For additional information, we refer to [66] and [9].

Linear Programming and *Integer Programming* are two basic approaches to solve network flow problems and combinatorial optimization problems. We will introduce both techniques in section 1.4 and suggest [147, 153] for further reading.

Please note that the following chapters also provide their own introductions and terms specific to these chapters are defined there.

1.1 Graphs

In this work we consider finite graphs $G = (V, E)$ where $V = V(G)$ is the *vertex set* and elements $v \in V$ are called *vertices* or *nodes*. $E = E(G)$ is the *edge set* of G. We consider both *undirected* and *directed* graphs (*digraphs*). In the case of undirected, loop free graphs the edge set is a subset of V^2, i.e. $E \subseteq \{\{u, v\} : u, v \in V,\ u \neq v\}$.

To denote directed edges, we also call them *arcs*. E is termed *arc set* A. A consists of ordered pairs of nodes, i.e. $A \subseteq V \times V = \{(u, v) : u, v \in V\}$. Therefore, each arc $a \in A,\ a = (u, v)$ is directed from its $\text{tail}(a) = u$ to its $\text{head}(a) = v$. For $v \in V$, we use $\delta^-(v) = \{a \in A :\ v = \text{head}(a)\}$ for the set of incoming arcs and $\delta^+(v) = \{a \in A :\ v = \text{tail}(a)\}$ for the set of outgoing arcs. A graph is called *bi-directed* if it contains for each arc $a = (u, v)$ also the arc in the opposite direction $a' = (v, u)$. A directed graph can be made undirected by simply deleting the directions of the arcs. To make an (undirected) graph a bi-directed one we add both directions for each edge/arc.

The cardinalities of the node and edge sets are denoted by $n = |V|$ and $m = |E|$. A graph with n vertices that contains all possible edges, is called a *complete graph* and denoted by K_n. Obviously, the complete graph has $m = \binom{n}{2} = \frac{n(n-1)}{2}$ edges. Sometimes, a graph is allowed to contain *multi-edges*, i.e., parallel edges. Hence, E, or A respectively, is defined as a multi-set in this case and G is called *multi-graph*. The *induced subgraph* on a vertex set $V' \subseteq V$ is denoted by $G[V']$. The induced arc set of $G[V']$ is denoted by $A[V']$.

A sequence $W = (a_1, \ldots, a_k)$, $a_i \in A$, of arcs is called a *walk* if it fits head to tail, i.e. $\text{head}(a_i) = \text{tail}(a_{i+1})\ \forall i \in \{1, \ldots, k-1\}$. For short, we will use $\text{tail}(W) := \text{tail}(a_1)$ and $\text{head}(W) := \text{head}(a_k)$. $V[W]$ is used for the set of vertices that occur in the arcs of W. To simplify matters, we use $a \in W$ to denote that the arc a is contained in the sequence of arcs in walk W. A *path* P is a walk which passes through every vertex at

most once. A walk/path where the tail of the first arc and the head of the last arc coincide is called *cycle/circuit*. For $u, v \in V$, $\mathcal{P}_{u,v}$ denotes the set of all paths with tail u and head v. The *length* of a path (with respect to *unit edge lengths*) is the number of arcs in its sequence. Two paths P_1 and P_2 are (arc) *disjoint* if $A[V[P_1]] \cap A[V[P_2]] = \emptyset$. They are *node disjoint* if $V[P_1] \cap V[P_2] = \emptyset$. The composition of two paths $P_1 = (a_1, \ldots, a_k)$ and $P_2 = (b_1, \ldots, b_l)$ is defined as $P_1 \circ P_2 = (a_1, \ldots, a_k, b_1, \ldots, b_l)$.

Similarly, walks, paths, cycles, and circuits can be defined for undirected graphs.

A graph is *strongly connected* if for each pair of vertices (u, v) there exists a path P from u to v, i.e. $\text{tail}(P) = u$ and $\text{head}(P) = v$. A graph is *weakly connected* if its corresponding bi-directed graph is strongly connected. A node set $U \subseteq V$ is (strongly/weakly) connected if the induced graph $G[U]$ is (strongly/weakly) connected. The inclusion maximal (strongly/weakly) connected subgraphs of G are called *(strongly/weakly) connected components*.

A *cut* in G is an arc set C such that $G \backslash C = (V, A \backslash C)$ has at least one connected component more than G. The *value* of a cut is simply the number of arcs in the cut. An *s-t-cut* is defined as a partition of V into two subsets C_1 and C_2, such that $s \in C_1$ and $t \in C_2$. Let $C = \{a \in A : \text{tail}(a) \in C_1 \wedge \text{head}(a) \in C_2\}$ then there exists no directed path from s to t in $G \backslash C$. The value of the s-t-cut is $|C|$, i.e. the number of forward arcs from C_1 to C_2.

1.2 Flows and Networks

Flows have been a major planning tool for many applications ever since Ford and Fulkerson [58] studied them. Before considering flows with additional routing constraints in the following sections, standard flow is introduced here.

1.2.1 Definitions

Although there is consens what a *flow* in a *network* should be, we will use a slightly different and modular approach for defining flows[3].

A *flow function* is a non-negative function on the arc set $x : A \to \mathbb{R}_0^+$. In most practical applications the maximal flow on each arc is limited by a *capacity* bound, i.e., a maximal flow value that cannot be exceeded on this arc. The capacities are given by a function $u : A \to \mathbb{R}_0^+ \cup \{\infty\}$. A flow is *feasible* if $0 \leq x(e) \leq u(e)\ \forall e \in A$ holds. A graph together with capacities is called a *network* $G = (V, A, u)$. For some results in this thesis, we will limit the capacity function to integer values, i.e., $u : A \to \mathbb{N}_0$. This is no restriction for most applications, since rational values can simply be scaled to integer values and irrational numbers cannot exactly be represented in our computers anyway[4].Furthermore, a flow function is *integral* if it has only integral values.

[3] In some of the algorithms especially in Section 3, we will not be able to fulfill all requirements of a flow at once. A modular definition admits step-by-step procedures. For example, we will define and derive *preflows* with a slightly different flow conservation constraint.

[4] Irrational input may yield a unexpected behavior, the algorithm of Ford and Fulkerson is a prime example.

The *inflow* and *outflow* of a flow function x at a given node v are

$$\text{inflow}_x(v) := \sum_{a \in \delta^-(v)} x(a)$$

$$\text{outflow}_x(v) := \sum_{a \in \delta^+(v)} x(a)$$

and the *balance* of a flow function x at a node v is defined as net flow out of this node

$$\text{bal}_x(v) := \text{outflow}_x(v) - \text{inflow}_x(v).$$

We can now distinguish between three kinds of nodes. At some nodes $v \in V$ flow may enter into the network, i.e., the outflow is higher than the inflow. Therefore, these nodes have positive balance and we call them *sources*. The set of all sources is denoted S^+. At other nodes, the balance may be negative and flow may leave the network. These nodes are called *sinks* and the set of all sinks is accordingly denoted S^-. All nodes in S^+ and S^- are also called *terminals* and we demand $S^+ \cap S^- = \emptyset$. For all other nodes v we require that x satisfies the *flow conservation* constraint, i.e., $bal_x(v) = 0$.

Definition 1.1 (Flow). *A* flow *in a network $G = (V, A, u)$ with sources S^+ and sinks S^- is a feasible flow function $x : A \to \mathbb{R}_0^+$ that*

1. *satisfies the flow conservation at all non-terminals,*
2. *has non-negative balance at all sources,*
3. *has non-positive balance at all sinks.*

An *s-t-flow* is a flow with a single source $s \in V$ and a single sink $t \in V$. A *circulation* is a flow where $bal_x(v) = 0$ holds for all $v \in V$ and thus, there are no sources or sinks.

The *value* of a flow x is the amount of flow reaching the sinks, i.e.,

$$val(x) = -\sum_{v \in S^-} bal_x(v).$$

This is equal to $val(x) = \sum_{v \in S^+} bal_x(v)$ due to flow conservation at the non-terminals. For most applications we need to restrict the balances of the terminals. We use a *supply/demand function* $d : V \to \mathbb{R}$. All sources have positive values $d(v) > 0$, while all sinks have negative values $d(v) < 0$. For non-terminals we require $d(v) = 0$. When we talk about the *supply* of a source or the *demand* of a sink this refers to the absolute values of the supply/demand function.

We can now study two variants of flow problems. In the case of a *transshipment* the supplies and demands should exactly be *satisfied*. This means $bal_x(v) = d(v)\ \forall v \in V$. Of course, this can only be achieved if $\sum_{v \in V} d(v) = 0$. In a weaker variant a source s may send up to $d(s)$ and a sink t can accept up to $d(t)$ units of flow. A flow *obeys* the supply/demand function if $0 \leq bal_x(v)d(v) \leq d(v)^2\ \forall v \in V$.

These two variants lead to two problems.

Problem 1.2. *Consider a network $G = (V, A, u)$ with sources S^+, sinks S^-, and a matching supply/demand function d.*

1. *Is there a flow x that satisfies d?* (Transshipment Problem)
2. *What is the maximum value of a flow x obeying d?* (Max Flow)

Note that both problems can also be formulated as a circulation problem. In many problems one can typically add a *supersource* s^* and a *supersink* t^* without changing the problem. More precisely, one connects the supersource with all sources, i.e., one adds the arcs (s^*, s) $\forall s \in S^+$ with capacities $u((s^*, s)) = d(s)$. Respectively, the same is done for the supersink with arcs (s, t^*) $\forall s \in S^-$ with capacities $u((s, t^*) = -d(s)$. Connecting supersink and supersource as well, we require $x((t^*, s^*)) = \sum_{v \in S^+} d(v)$ for the transshipment or we try to maximize $x((t^*, s^*))$ for a maximum circulation. Furthermore, flow conservation applies in every node.

This general definition of flows can be extended by adding *costs* to each arc, i.e., there is another function $c : A \to \mathbb{R}_0^+$. These *arc costs* can be interpreted as the length of an arc or a toll that has to be paid for using this arc. The total cost of flow on an arc e is $f(e)c(e)$ and the total cost of a flow in a network is given by $\sum_{e \in A} c(e)f(e)$. We can now vary the above flow problems by adding an additional constraint, which limits the overall costs of a flow. Or we can look for the cheapest flow among all flows with maximum flow value.

Problem 1.3. *Consider a network $G = (V, A, u)$ with sources S^+, sinks S^-, a matching supply/demand function d, and a cost function c.*

1. *What is the minimum cost of a transshipment?* (Min Cost Transshipment)
2. *What is the minimum cost of a maximum flow?* (Min Cost Flow)

One can also think of negative costs, i.e., getting money for using an arc. But this complicates the computation of shortest paths and Min Cost Flow. Shortest paths can have negative infinite length if the graph contains a cycle with negative cost sum. On the other hand a minimum circulation can be used to calculate a Max Flow. Just add supersource and supersink as above with $c((t^*, s^*)) = -1$, and all other costs $c(e) = 0,\ e \neq (t^*, s^*)$.

Up to now, we considered only homogeneous flows. In many applications one has to deal with several goods with different origins and destinations in the same network. This can be modeled by *multicommodity flows*. For each commodity $i, i \in \{1, \ldots, N\}$ we implement a separate flow function $x_i : A \to \mathbb{R}_0^+$. We require flow conservation as above for each flow x_i. The capacity of an arc is shared among the commodities, $\sum_{i=1}^{N} x_i(e) \leq u(e)$ $\forall e \in A$. All problems given above can be formulated in a multicommodity variant [2].

1.2.2 Important results

In 1927, Menger presented his famous result which shows the relationship between disjoint paths and cuts.

Theorem 1.4 (Menger, 1927). *Let $G = (V, A)$ be a graph and $s, t \in V$. The maximum number of edge disjoint paths from s to t is equal to the minimum value of an s-t-cut.*

Introducing arc weights, i.e. arc capacities, this result can be extended to flows. Hereby, the *capacity* of an s-t-cut is $\sum_{e \in C} u_e$ where C is the set of forward arcs from $C_1 \ni s$ to $C_2 \ni t$.

Theorem 1.5 (Maximum-flow Minimum-cut). *The maximum value of an s-t-flow is equal to the minimum capacity of an s-t-cut.*

This theorem was proven by P. Elias, A. Feinstein, and C.E. Shannon in 1956, and independently also by L.R. Ford, Jr. and D.R. Fulkerson in the same year.

Several algorithms for finding a maximum s-t-flow have been developed since the 1950s. To name a few approaches, Ford and Fulkerson constructed an algorithm that is related to their constructive proof of the maximum-flow minimum-cut theorem. This algorithm uses *augmenting* paths and was improved by Dinic and by Edmonds and Karp. Goldberg and Tarjan presented a push-relabel-algorithm, which was improved by Ahuja and Orlin. Note that, despite the bunch of combinatorial algorithms for s-t-flows, no exact combinatorial algorithm is known for multicommodity flows, yet.

The successively increasing flow value in the algorithm of Ford and Fulkerson also leads to an important observation.

Corollary 1.6 (Integrality theorem). *If the capacity function u is integral, there exists an integral maximum flow.*

Note that this result is not true for multicommodity flows.

There is also another important link between flows and paths. Let x be an s-t-flow in a network $G = (V, A, u)$ with $s, t \in V$. Then there exists up to $m = |A|$ s-t-paths or cycles, such that x can be represented as a nonnegative linear combination of these paths or cycles. Moreover, if x is integral, the linear combination can be realized with integral coefficients. This result is also known as *path decomposition*.

1.3 Algorithms and Complexity

In the previous section we just mentioned some algorithms for maximum flows and many more algorithms for restricted flows will follow in this thesis. But when talking about algorithms, we have to discuss *running times* and *complexity*. Again, we can only provide an overview.

An *algorithm* is a finite list of instructions. These instructions perform operations on the given data, but they need not be performed in linear order. Each instruction also determines which instruction is next or may even stop the algorithm. Therefore, the algorithm itself is fixed, but the input data may vary.

Obviously, the running time of an algorithm may depend on the size of the input. We will not discuss memory models or machine models here. For a detailed survey on deterministic Turing machines see [66].

For short, we measure the size of the input data as its lengths in some *binary encoding*. The running time of an algorithm is measured in the number of *elementary arithmetic operations* (addition, subtraction, multiplication, division, comparison), that are performed until the algorithm stops. Even inputs of the same size may lead to different behavior of the algorithm. Considering the worst case, the *time complexity function* for an algorithm is the largest amount of time for each possible input length, that is needed by the algorithm to solve an instance of this size.

1.3.1 Polynomial-time algorithms

If the number of elementary operations is bounded by a polynomial in the input size, the algorithm is called *polynomial-time algorithm* or *efficient algorithm*. We can describe the running time of an algorithm with help of the $\mathcal{O}$-notation. A function $f(n)$ is in $\mathcal{O}(g(n))$ if there exists a constant factor c and $n_0 \in \mathbb{N}$ such that $|f(n)| \leq c|g(n)| \; \forall n \geq n_0$. Therefore, an algorithm is efficient, if its time complexity function is in $\mathcal{O}(p(n))$ for a polynomial p and input length n.

To simplify matters we do not measure the size of a graph or network in some binary encoding. The size of a graph depends on the number n of nodes and the number m of edges/arcs. Therefore, the input size for graph related problems is in most cases $n+m$.

With help of this notation, we can now compare algorithms. For example, the algorithm for Max Flow from Edmonds and Karp with shortest augmenting paths has a time complexity of $\mathcal{O}(nm^2)$, while the algorithm from Goldberg and Tarjan has time complexity $\mathcal{O}(nm \log(n^2/m))$. Remember that $m < n^2$.

Note that huge constant factors are maybe ignored in this run time analysis. Also, the running time may depend on some additional implementation tricks. The popular Dijkstra's algorithm for shortest path calculation (cf. [50]) can simply be implemented with a time complexity of $\mathcal{O}(n^2)$. Using a more sophisticated data structure, namely Fibonacci heaps, the time complexity can be reduced to $\mathcal{O}(n \log n + m)$ [2].

Sometimes, an input value itself appears in the running time of an algorithm instead of its logarithm (its size). If the complexity bound is also a polynomial of this parameter, the algorithm is called *pseudo-polynomial*.

For other problems, it is useful to describe the input with several parameters. This allows a finer analysis of their inherent complexity. Not only the size of the input data influences the running time of the algorithm, but the structure of the data may be important as well. Therefore, for some hard problems one can compute an answer in a time that is polynomial in the size of the input and exponential in a parameter k. If k is small and fixed, then such problems can still be considered manageable. The corresponding algorithms are called *fixed-parameter tractable*.

1.3.2 $\mathcal{P}$, $\mathcal{NP}$, and co-$\mathcal{NP}$

For some problems one has not found polynomial time algorithms yet. This leads to the question whether some problems are harder than other problems. The classes $\mathcal{P}$, $\mathcal{NP}$, and co-$\mathcal{NP}$ are collections of decision problems. A decision problem is a problem that

can be answered by YES or NO. More precisely, we consider a finite alphabet Σ and the set Σ^* of all finite words of letters from Σ. A problem Π is an arbitrary subset of Σ^*. For a given $x \in \Sigma^*$ we have to decide whether $x \in \Pi$.

Problems in $\mathcal{P}$ are considered to be 'easy' problems, because they can be solved on a deterministic Turing machine in polynomial time with respect to the length of the input.

A problem is in $\mathcal{NP}$ when it can be answered on a nondeterministic Turing machine in polynomial time. Clearly, $\mathcal{P} \subseteq \mathcal{NP}$. An equivalent definition for $\mathcal{NP}$ requires that a given certificate (YES-solution) can be verified on a deterministic Turing machine in polynomial time. For example, it is hard to decide whether a graph contains a Hamiltonian circuit, i.e., a cycle that visits all nodes exactly once. But when a cycle is given, it is easy to check, whether it is a Hamiltonian circuit.

A problem is in co-$\mathcal{NP}$ when its complement is in $\mathcal{NP}$. It holds $\mathcal{P} \subseteq \mathcal{NP} \cap \text{co-}\mathcal{NP}$. But it is an open problem whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$. Assuming the second case holds, $\mathcal{NP}$-hard problems are not solvable in an acceptable time.

1.3.3 $\mathcal{NP}$-complete problems

A problem in $\mathcal{NP}$ is said to be $\mathcal{NP}$-complete if each problem in $\mathcal{NP}$ can be reduced to this problem. Hereby, a reduction is a polynomial time algorithm that transforms a problem into another problem such that both problems have the same answer. Hence, $\mathcal{NP}$-complete problems are also referred as the hardest problems in $\mathcal{NP}$. A problem is $\mathcal{NP}$-hard if and only if there is an $\mathcal{NP}$-complete problem that is reducible in polynomial time to this problem. In other words, a problem is $\mathcal{NP}$-hard when it is at least as hard as the hardest problems in $\mathcal{NP}$. Note that an $\mathcal{NP}$-hard problem needs not to be in $\mathcal{NP}$. It could be even harder, undecidable or it may not even be a decision problem. This notation dates back to Cook, who proved that there exists a formal language with $\mathcal{NP}$-complete problems [35]. Building on that, Karp [85] showed for 21 popular combinatorial problems (e.g. the TRAVELLING SALESMAN PROBLEM (TSP)) that they are $\mathcal{NP}$-complete.

1.3.4 Complexity of optimization problems

We defined complexity for decision problems, i.e., problems with a YES or NO answer. But in most problems introduced up to now like MAX FLOW or MIN COST FLOW, we are looking for a minimum or maximum of some objective.

Assume, we minimize a function $f(x)$ over $x \in X$. We consider the following decision problem:

Given a value v, is there an $x \in X$ with $f(x) < v$?

We perform a binary search to find the optimal value v. If we have an upper bound on the size of the optimal solution and a polynomial time algorithm for the decision problem, this usually yields a polynomial time algorithm for the optimization problem. On the other hand, if an optimization problem has an $\mathcal{NP}$-complete decision version, then it is $\mathcal{NP}$-hard.

1.3.5 Approximation

Many practical optimization problems are $\mathcal{NP}$-hard problems. Theory states that most real world instances of such problems cannot be solved exactly in an acceptable time, assuming $\mathcal{P} \neq \mathcal{NP}$. But on the other hand, this is not necessary for a lot of applications and an exact solution is of limited use when, e.g., the input data is noisy. In this case a reasonable good solution near to the optimum is sufficient.

A p-approximation algorithm is an algorithm that runs in polynomial time and calculates a solution that is at most a factor p away from the optimum. The factor p is called the *performance ratio* of the approximation. Obviously, to performance ratio for a maximization algorithm is less than 1, while it is greater than 1 for minimization problems.

If we can find polynomial time approximation algorithms with a performance ratio of $p = (1 + \epsilon)$ for minimization problems or $p = (1 - \epsilon)$ for maximization problems $\forall \epsilon \in (0, 1)$, we call the family $\{\mathcal{A}_\epsilon\}_{0<\epsilon<1}$ a *polynomial time approximation scheme* (PTAS). Furthermore, when each algorithm $\mathcal{A}_\epsilon$ in this family has a running time polynomial in the input size and polynomial in $1/\epsilon$ we call the family $\{\mathcal{A}_\epsilon\}_{0<\epsilon<1}$ a *fully polynomial time approximation scheme* (FPTAS).

Approximation algorithms are not only used for hard problems. For example, there exist combinatorial approximation algorithms for Multicommodity Flow (see, e.g., [67]).

1.4 Linear Programming and Integer Programming

1.4.1 Linear programs

Linear programming (LP) is a powerful optimization tool, which we will also use in this thesis. Again, we can only sketch the main ideas here. Fortunately, there exist very good text books and we recommend, e.g., [147, 153].

Whenever we can formulate a problem in the form

$$\begin{aligned} \min \quad & c^\mathsf{T} x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{Q}^n \end{aligned}$$

with the objective $c^\mathsf{T} x$ ($c \in \mathbb{Q}^n$), and some constraints $Ax \leq b$ ($A \in \mathbb{Q}^{m \times n}, b \in \mathbb{Q}^m$) linear programming is an alternative option to solve this problem. For example, the Max Flow problem can be formulated as a linear program. For simplicity, we assume a single source s and a single sink t with infinite capacities:

$$\max \quad \sum_{e=(s,v)} x(e) - \sum_{e=(v,s)} x(e) \tag{1}$$

$$\text{s.t.} \quad u(e) - x(e) \geq 0 \qquad \forall e \in A \tag{2}$$

$$\sum_{e=(u,v)} x(e) - \sum_{e=(v,w)} x(e) = 0 \qquad \forall v \in V \backslash \{s,t\} \tag{3}$$

$$x(e) \geq 0 \tag{4}$$

Hereby, the objective (1) is the outgoing flow of the source. The constraints (2) ensure the capacity bounds. The flow conservation is formulated in (3), i.e., the balance is equal to zero. Of course, all flow values have to be non-negative (4).

Linear programs can be solved by the simplex algorithm, presented by Dantzig [44] in 1947. Despite the simplex algorithm has no polynomial running time in theory, it works well in practice.

In 1979, Khachiyan proved that linear programs can be solved in polynomial time with the ellipsoid method [87]. Therefore, whenever we can formulate a problem Π as a linear program, and this formulation is polynomial in time and size of the original problem, this proves that Π is in $\mathcal{P}$. For example, no combinatorial algorithm is known for the Multicommodity Flow problem. But it is easy to formulate this problem as a linear program similar to the formulation of Max Flow above. Hence, Multicommodity Flow is in $\mathcal{P}$.

Interestingly, the ellipsoid method is inefficient in practice. Today, most solvers for LP also use interior points methods, introduced by Karmarkar [84] in 1984, which are efficient in theory and practice.

The existence of a dual program is an important property of linear programs. As a consequence of Farkas' lemma, each linear program has a dual linear programming formulation and both problems have the same optimal objective value [147]. For example, the path based Max Flow can be formulated as linear program as follows:

$$\begin{aligned} \max \quad & \sum_{P \in \mathcal{P}_{s,t}} x_P & & \\ \text{s.t.} \quad & \sum_{P: e \in P} x_P \leq u(e) & & \forall e \in A \\ & x_p \geq 0 & & \forall P \in \mathcal{P} \end{aligned}$$

Note that the set $\mathcal{P}_{s,t}$ is very large in general. It is not a good idea to list all paths between two nodes. However, this formulation is very useful when only considering a rather small set of active paths. This can be achieved, e.g., by a column generation approach. The dual formulation is:

$$\begin{aligned} \min \quad & \sum_{e \in A} y_e u(e) & \\ \text{s.t.} \quad & \sum_{e \in P} y_e \geq 1 & \forall P \in \mathcal{P} \\ & y_e \geq 0 & \forall e \in E \end{aligned}$$

It is not obvious that this dual linear program always has an optimal solution with integer values for y_e. One will need a result concerning total unimodular matrices to prove this. So, assume, we have found this integer solution. This optimal solution of the dual determines a minimum s-t-cut in the graph, i.e., the set of arcs in the cut is $\{e : y_e = 1\}$. All paths between s and t are cancelled by deleting this set of arcs. Furthermore, the objective is the weighted sum over this set of arcs.

Solving primal and dual LP simultaneously yields bounds for the optimal solution. Furthermore, if both solutions have equal value, then optimality is proved.

1.4.2 Integer programs

If some of the variables have to be integral, finding an optimal solution is much harder. Simply dropping the integrality constraints and rounding does not need to yield good solutions in general. In fact, a special case, the decision version of an integer program with binary variables, is one of Karp's 21 $\mathcal{NP}$-complete problems.

Integer Programming and Mixed Integer Programming are based on linear programming, but they use various strategies to handle integrality constraints. Typically, one solves the relaxation of the problem, i.e., the integrality constraints are dropped. If the solution is not integral, the problem is modified and solved again. These modifications include adding new constraints or splitting the problem. Two strategies are briefly introduced here. For further reading, we recommend [153].

Cutting plane methods This method was first used by Dantzig et al. [45] to solve an instance of the Travelling Salesman Problem, containing 49 major cities in the United States. Gomory [70] generalized this approach to arbitrary integer programs. A cutting plane is an additional constraint that refines the relaxation of an integer program. If the optimal solution of the relaxation is not integral, then there exists a linear inequality separating the optimum from the integer feasible set. This inequality can be added to the relaxation. Obviously, the current optimum is no longer feasible. This process is repeated until an optimal integer solution is found.

Branch&Bound The branch&bound method was introduced by Land and Doig [104]. This enhanced enumeration method consists of two steps. In the branching step the problem is split into two smaller problems. For example, assume the value of a variable x is not integral in the optimal solution, i.e. $x = r$ and $r \notin \mathbb{Z}$. We may now consider two new problems:

- one with the additional constraint $x \leq \lfloor r \rfloor$
- the other one with the additional constraint $x \geq \lceil r \rceil$

Obviously, the union of these two sub-problems is the original problem. Now, a recursive branching is executed. In each step, a new variable and a new threshold is chosen, depending on the branch in the last step. In the bounding step upper and lower bounds for each sub-problem are calculated. If we consider a minimization problem and a lower bound of some sub-problem X is greater than the upper bound of another sub-problem Y, then X can be safely discarded from the branching (pruning), since Y will always yield better solutions. Again, this is repeated until an optimal integer solution is found.