K-Best Enumeration
Theory and Application

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Denis Benjamin Kurz

Dortmund

2017

Denis Benjamin Kurz
Lehrstuhl 11 – Algorithm Engineering
Fakultät für Informatik
Technische Universität Dortmund
Otto-Hahn-Str. 14
44227 Dortmund

Tag der mündlichen Prüfung	**14. März 2018**
Dekan:	**Prof. Dr.-Ing. Gernot A. Fink**
Gutachter:	**Prof. Dr. Petra Mutzel** (TU Dortmund, Fakultät für Informatik)
	Prof. Dr. Markus Chimani (Universität Osnabrück, Institut für Informatik)

Contents

Abstract

Where an optimal solution does not contain sufficient information about a given problem instance, enumerating good solutions is a common coping strategy. In combinatorial optimization, k-best enumeration, or ranking, has been studied and applied extensively. The k shortest simple path problem in directed, weighted graphs (kSSP), introduced in 1963 by Clarke, Krikorian and Rausen, is particularly well known. Efficient existing algorithms are based on Yen's algorithm for this problem; they all feature a worst-case running time of $\mathcal{O}(kn(m + n \log n))$ on a graph with n vertices and m edges. Vassilevska Williams and Williams show that a polynomially faster kSSP algorithm would also result in an algorithm for the all-pairs shortest path problem with running time $\mathcal{O}(n^{3-\varepsilon})$, which seems unlikely at the moment.

We present a kSSP algorithm that is not based on Yen's algorithm, matching the state-of-the-art running time of $\mathcal{O}(kn(m + n \log n))$. In particular, we do not solve a single instance of the replacement path problem, a basic building block of Yen's algorithm. Instead, we make use of ideas used in Eppstein's algorithm for a similar problem where paths are allowed to be non-simple. Using our algorithm, one may find $\Theta(m)$ simple paths with a single shortest path tree computation and additional $\mathcal{O}(m + n)$ time per path in well-behaved cases. We also propose practical improvements for our algorithm, comprising dynamic edge pruning, lazy shortest path tree computations, and fast path simplicity checks. In a detailed computational study, we demonstrate that well-behaved cases are quite common in random graphs, grids and road networks. We showcase usefulness of the dynamic edge pruning approach on those three graph classes and on network topology graphs. Despite 40 years of heavy research on Yen-based kSSP algorithms, including involved algorithm engineering, our algorithm proves to be faster than existing algorithms by at least an order of magnitude.

However, there is not much room for improvement for the general worst case. We demonstrate that kSSP can be solved considerably faster if the input graph is restricted to have bounded treewidth. Specifically, we propose an algorithm template for enumerating the k best solutions in a bounded treewidth graph for any problem that can be expressed in counting monadic second-order logic. Our template is a generalization of Courcelle's theorem, mainly utilizing dynamic programming and a persistent heap data structure. It enumerates any constant amount of solutions in time $\mathcal{O}(n)$. For general k, it requires $\mathcal{O}(\log n)$ extra time per solution, resulting in a total running time of $\mathcal{O}(n + k \log n)$. Finding the initial solution is parallelizable, so the linear term can be dropped and we obtain a running time of $\mathcal{O}(k \log n)$ in the PRAM model. The class of problems expressible in counting monadic second order logic contains kSSP, matching problems, or the spanning tree problem, but also a number of NP-hard problems like the traveling salesman problem, where we achieve a doubly-exponential speedup.

Zusammenfassung

Bei der klassischen kombinatorischen Optimierung begnügt man sich damit, lediglich eine optimale Lösung zu berechnen. Manchmal werden jedoch Anforderungen an eine Lösung gestellt, die nur schwer formalisierbar sind und deshalb in der Modellierung des Optimierungsproblems nicht berücksichtigt werden. Das Aufzählen der k besten Lösungen, auch *Ranking* genannt, kann dann eine sinnvolle Strategie sein. Zu den kombinatorischen Problemen, deren Ranking-Variante zahlreiche Algorithmiker beschäftigt, zählt das Aufzählen der k kürzesten einfachen Wege in einem gewichteten, gerichteten Graphen (kSSP), vorgestellt 1963 von Clarke, Krikorian und Rausen. Der Algorithmus von Yen, der das Problem auf einem Graphen mit n Knoten und m Kanten in Zeit $\mathcal{O}(kn(m + n \log n))$ löst, dient dabei als Vorlage für alle Algorithmen von praktischer Relevanz. Vassilevska Williams und Williams haben dabei gezeigt, dass polynomielle Laufzeitverbesserungen nur möglich sind, wenn auch das Bestimmen aller paarweisen Distanzen in einem gewichteten, gerichteten Graphen in Zeit $\mathcal{O}(n^{3-\varepsilon})$ möglich ist.

Wir präsentieren einen Algorithmus für das kSSP, der nicht auf dem Algorithmus von Yen basiert, dessen asymptotische Laufzeit von $\mathcal{O}(kn(m+n \log n))$ jedoch einstellt. Insbesondere löst dieser Algorithmus im Allgemeinen keine Instanz der günstigsten Ersatzpfade, einem Grundbaustein des Algorithmus von Yen. Stattdessen verwenden wir Ideen, die auch im Algorithmus von Eppstein genutzt wurden. Der Algorithmus von Eppstein löst ein mit dem kSSP verwandtes Problem, bei dem die ausgegebenen Pfade Kreise enthalten dürfen. Unter günstigen Voraussetzungen entdeckt unser Algorithmus bereits durch eine Berechnung eines Kürzeste-Wege-Baums $\Theta(m)$ verschiedene einfache Pfade. Sie auf Einfachheit zu überprüfen und auszugeben bedarf einer Laufzeit von $\mathcal{O}(m+n)$ pro Pfad. Wir beschreiben außerdem praktische Verbesserungen, die die Menge der zu betrachteten Kanten sukzessive verkleinert, die benötigten Kürzeste-Wege-Bäume *lazy* berechnet sowie die Überprüfung von Pfaden auf Einfachheit beschleunigt. In einem ausführlichen experimentellen Vergleich zeigen wir, dass unser Ansatz den Yen-basierten in der Praxis überlegen ist: Die günstigen Voraussetzungen treffen wir dabei in Zufalls- und Gittergraphen sowie in großen Straßennetzgraphen an. Trotz großer Algorithm-Engineering-Bemühungen zur Verbesserung von Yens Algorithmus erreicht unser Algorithmus auf diesen Graphklassen sowie auf Graphen, die Netzwerktopologien modellieren, eine Beschleunigung von mindestens einer Größenordnung.

Wegen der Erkenntnisse von Vassilevska Williams und Williams sind momentan keine großen Verbesserungen der theoretischen Worst-Case-Laufzeit bei kSSP-Algorithmen im allgemeinen Fall zu erwarten. Wir betrachten daher den Spezialfall der baumweitebeschränkten Graphen. Wir präsentieren einen Meta-Algorithmus, der für eine große Klasse an Graphproblemen zu einem Ranking-Algorithmus für baumweitebeschränkte Graphen adaptiert werden kann. Die Problemklasse umfasst genau die Graphprobleme, die in der monadischen Prädikatenlogik zweiter Stufe mit Modulo-Prädikat formuliert werden können. Es handelt sich um eine Verallgemeinerung von Courcelles Theorem; Schlüsseltechniken sind dynamische Programmierung sowie eine persistente Datenstruktur. Für das Aufzählen einer festen Anzahl an Lösungen wird Zeit $\mathcal{O}(n)$ benötigt. Ist die Anzahl k an Lösungen Teil der Eingabe, benötigt der Algorithmus $\mathcal{O}(\log n)$ Zeit zusätzlich pro Lösung, also insgesamt $\mathcal{O}(n + k \log n)$. Die Berechnung der ursprünglichen Lösung kann dabei parallelisiert werden. Dadurch entfällt im PRAM-Modell der lineare Term und wir erhalten eine Laufzeit von $\mathcal{O}(k \log n)$. Probleme, die in monadischer Prädikatenlogik zweiter Stufe ausgedrückt werden können, umfassen das kSSP, Matching-Probleme oder das Spannbaumproblem, aber auch das Problem des Handlungsreisenden, für das eine doppelt exponentielle Laufzeitverbesserung im Vergleich zu bestehenden Ansätzen auf allgemeinen Graphen erzielt wird.

Danksagung

The support of a lot of folks compensated for my potential lack of perseverance to finish this thesis. Thanks to Petra Mutzel for giving me a place in her team, and for familiarizing me with Algorithm Engineering. She also made me aware of my tendency to underestimate my research. I was able to do research in the DFG Research Training Group (RTG) on discrete optimization of technical systems under uncertainty, undisturbed from teaching, primarily thanks to Peter Buchholz and Petra Mutzel. Thanks to Uwe Schwiegelshohn for mentoring me as part of the RTG and for providing me with helpful feedback and thorough reviews of my reports.

Thanks to Markus Chimani and his team for maintaining the state-of-the-art C++ graph and data structure library OGDF, thereby pushing and upholding Free Software as a means to reproducibility in computational studies. I am grateful to David Eppstein for hosting my stay in Irvine and inspiring research on k-best enumeration in an FPT context, and for all the ways he presents research to various audiences. Thanks to Ivana Ljubić for teaching me early the academic analogy to the saying “If I had more time, I would have written you a shorter letter.” Putting effort into the presentation of a new result can conceal how much effort it took to find that result in the first place. Thanks to Antonio Sedeño-Noda for providing me with his implementation of his kSSP algorithm.

Thanks to all my present and former colleagues for every bit of inspiration through conversations and talks. I am also grateful for every distraction from work during lunch breaks and otherwise, and every other work-live balance preserving measure. Distraction when distraction is due! Thanks to Gundel Jankord for always cheerfully keeping track of all those organizational bits. Thanks to Bernd Zey for caring for my safe voyage from diploma to PhD thesis, including proofreading of them both. I am grateful to Fritz Bökler for interesting discussions about every imaginable topic for more than ten years; for seeing me through the duration of my study and graduation; and, too, for proofreading my theses.

Danke Mama, dass du dafür gesorgt hast, dass ich zurückfinde, immer wenn ich im Begriff war, mich auf dem Weg zur Promotion zu verrennen, und das mindestens seit der sechsten Klasse. Es gab Zeiten, in denen scheinbar nur du wusstest, wo ich hingehöre. Danke Debby, dass du mir während der Promotion immer den Rücken freigehalten und mich auf jede erdenkliche Weise unterstützt hast.

Denis Kurz
Dortmund, December 2017

List of Figures

List of Tables

Part I.

Preliminaries

1. Introduction

In combinatorial optimization, we are usually interested in one optimal solution. Readers of common literature on the subject [31, 80, 84] might be tempted to believe that a solution to a combinatorial optimization problem is either optimal or completely useless. If there is no unique optimal solution, we sometimes make an effort to enumerate all solutions with optimal value. The usefulness of suboptimal solutions, however, is often disregarded although it has been demonstrated repeatedly.

In combinatorial *k-best enumeration*, sometimes called *ranking*, we compute k solutions such that no other solution is cheaper than the most expensive one we found. The problem of finding k shortest simple (or loopless) paths between two given vertices in a weighted, directed graph has received most attention. It was introduced by Clarke, Krikorian and Rausen [30] in 1963. In 1971, Yen [105] proposed an algorithm with $\mathcal{O}(kn(m + n \log n))$ running time, which was refined numerous times. The most recent refinement, the node classification algorithm from Feng [46], is the most efficient kSSP algorithm to date.

A problem as important as kSSP deserves to be approached by more than one way, though. Although the most recent Yen-based algorithm is only three years old, improvements seem to stagnate. A completely new approach could pave the way for a whole new array of practical improvements. We want to pursue new ways to improve *upon* Yen's approach, both theoretically and in practice.

Consider the street map in Figure 1.1. Assume we want to travel from Speyer to Dortmund by car during rush hour. Driving on a German autobahn is quite strenuous, so we want to get over it and prefer fast routes naturally. The fastest route is via Darmstadt, Montabaur and Olpe, which requires 221 minutes. This does not include breaks; mind your attention span when driving a car this long!

However, we might be constrained in one way or another. If we entertain a passion for cathedrals, we might approve an extra 14 minutes to be able to have a break in Cologne. After all, we just came from Speyer, whose cathedral is the largest remaining Romanesque church. If we drive an electrical vehicle with a battered battery that cannot drive for more than 130 minutes straight without recharge, we would opt for St. Elizabeth's Church, Marburg, which was a model for the Cologne Cathedral.

There may be countless other constraints. A bound on the probability of getting stuck in a traffic jam would have us avoid both Cologne and Frankfurt (on the Darmstadt–Marburg edge). To participate in a guided tour in St. Elizabeth's Church, we might want to arrive there between 16:30 and 17:00. If our car is a rental car, we have to pay per kilometer (plus a fixed rate). This could induce a bound on the travel distance if we are on a tight budget.

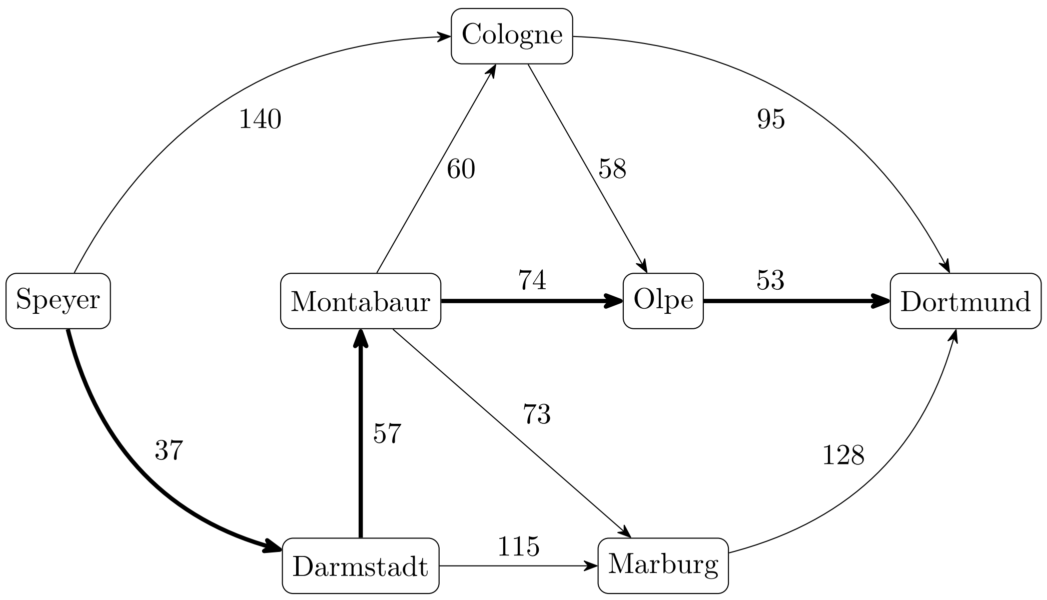


Figure 1.1.: Detailed map of the most important cities in Germany, objectively. Edge labels are travel times in minutes during evening rush hours. Thick edges lie on the unique fastest route from Speyer to Dortmund.

Some of these constraints can easily be incorporated into the process of finding the optimal route, but some are not. For example, by introducing a budget on the travel distance, we end up with the Constrained Shortest Path problem, an NP-hard optimization problem. We can also think of constraints that depend on data that is not yet available, like traffic jam information, roads closed off for maintenance, or the availability of tickets for the guided tour in St. Elizabeth's. The everyday cathedral seeker in an electrical vehicle often faces several constraints at a time. And finally, we cannot teach our navigation system about every possible constraint, and even if we could, communicating them all would probably be cumbersome.

Fortunately, there is an easy way out. Instead of incorporating all our constraints during route optimization, the navigation device computes several good routes with respect to our primary objective, getting from Speyer to Dortmund wasting as little time as possible. Instead of picking one, it would then also present several of those routes to us. It is relatively easy to assess the number of cathedrals, construction sites and recharging units, traveled distance, traffic jam probabilities for a fixed route. We are able to fill in missing data based on experience: Frankfurt area is not jammed when we leave Speyer, but it will definitely be around 16:00. We could decide to enjoy the view from autobahn A5 north of Frankfurt spontaneously, too. Picking a route ourselves also makes us feel less heteronomous and boosts our confidence!

We face the k shortest path problem in a directed, weighted graph (kSP), for a natural number k. Eppstein's algorithm [40] solves this problem on a graph with n vertices and m edges in time $\mathcal{O}(m + n \log n + k)$. Computing the shortest path already requires time $\mathcal{O}(m + n \log n)$ using Dijkstra's algorithm, so computing many good paths only introduces constant overhead per path. If we want the paths to be output in order of increasing length, we need an additional $\mathcal{O}(k \log k)$ time. Modeling our routing problem as kSP may introduce loops, though. Although Figure 1.1 depicts an acyclic graph, we can easily imagine it to contain autobahn junctions. In Germany, these junctions contain loops. Using two of them makes you go in your original direction and usually requires less than a minute. The second shortest path would therefore visit Darmstadt, Montabaur and Olpe, too, as well as two junction loops in addition, but we certainly do not gain anything from this detour.

Instead, we are only interested in loopless, simple paths. Although computing the k shortest simple paths requires $\mathcal{O}(kn(m+n\log n))$ time, and therefore $\mathcal{O}(n(m+n\log n))$ time per path, it may be worth the extra effort if we know in advance that we are only interested in simple ones. If we allow cycles of zero length, the number of non-simple paths can be unbounded. We are then unable to find even small numbers of simple paths with Eppstein's algorithm.

We motivated kSSP directly, but it is also highly relevant as a subproblem of other important problems. It can be used during gap closing for the constrained shortest path problem. It has been applied successfully in bioinformatics [3, 69, 88, 99], especially in biological sequence alignment [17, 20, 83, 97, 98]; to natural language processing [11, 18, 22, 23, 27, 28, 29, 47, 62, 71, 72, 95, 100]; to list decoding [57], minimum quartet inconsistency [56], parsing [63] and vehicle and transportation routing [58, 66, 81, 104]; and many others. See Eppstein's recent comprehensive survey on k-best enumeration [41, 42, 43] for more applications.

The best algorithm for kSSP is still Yen's algorithm [105], proposed in 1971. None of the numerous enhancements [24, 46, 60, 78, 79, 101] for Yen's algorithm were able to lower its running time of $\mathcal{O}(kn(m + n\log n))$. Actually, Vassilevska Williams and Williams suggest [103] that polynomial improvements might not be possible. If the second shortest simple path can be found in time $\mathcal{O}(n^{3-\varepsilon})$, we can also solve the all-pairs shortest path problem in time $\mathcal{O}(n^{3-\varepsilon'})$, for positive real $\varepsilon, \varepsilon'$.

Since we cannot improve upon the status quo in the general case, it is reasonable to consider restrictions on the input. A particularly common restriction regards the treewidth of the input graph. Problems that are easier on trees than they are on general graphs are often also easier on graphs with an arbitrary but fixed bound on the treewidth. Specialized algorithms for bounded treewidth graphs are often based on dynamic programming and leverage the tree decomposition. For example, algorithms with polynomial running time on bounded treewidth graphs have been proposed for independent sets [70], dominating sets [2, 93], q-Coloring [49], and odd cycle traversal [48], the Steiner tree problem [26] and its two-stage stochastic version [74]. Courcelle's theorem [33] states that any graph language that coincides with the set of models for a nullary graph predicate in a variant of monadic second-order logic can be decided in linear time $\mathcal{O}(n)$. An optimization version of Courcelle's theorem was proposed by Courcelle and Mosbah [35] and by Arnborg, Lagergren and Seese [5]. However, bounded treewidth graphs have not been considered in the context of k-best enumeration, yet.

This thesis is intended to close some of the abovementioned gaps. On the practical side, we propose a new algorithm for kSSP that is not based on Yen's algorithm. Its running time matches that of Yen's algorithm. We feel that opportunities for practical improvements of their approach are exhausted. Using a completely new approach often enables new possibilities for improvements.

Our algorithm is based on the concept of sidetracks, introduced by Eppstein for their kSP algorithm. During initialization, a shortest path tree to the target vertex is computed. A sidetrack is an edge that does not occur on this shortest path tree. Any path from the source to the target vertex can now be described as the sequence of its sidetracks. Eppstein's algorithm discovers new paths by progressively enhancing an initially empty sidetrack sequence in several ways. Our account includes first practical improvements and a comprehensive computational study where we demonstrate that our approach is highly competitive.

On the theoretical side, we bridge the gap between bounded-treewidth algorithms and k-best enumeration. We propose an enumeration algorithm with a running time of $\mathcal{O}(n + k \log n)$, i.e., with logarithmic extra time per requested solution. The dynamic programming approach utilizes a special form of tree decompositions called shallow tree decompositions, and persistent arborescences. The algorithm can be used for any problem in counting monadic second-order logic, and thus to kSSP, too. We demonstrate in which ways our approach can be parallelized, depending on whether a tree decomposition is given.

1.1. Overview and Relevant Publications

This thesis is composed of one preliminary part and two main parts. Part I starts with a rather informal introduction into the topic of k-best enumeration for graph problems in Chapter 1 above. We motivate k-best enumeration and show how it can be useful, but also point out its limits. For what is important to both main parts of the thesis, Chapter 2 contains common definitions and notation, and Chapter 3 contains basic concepts and data structures.

Part II is all about the problem of finding k shortest simple paths in a directed, weighted graph between two given vertices (kSSP). We give part-specific definitions and a short review of existing literature and repeat what is known about the problem's complexity in Chapter 4. We review relevant existing approaches for kSSP in Chapter 5 and observe that they are all based on Yen's algorithm from 1971. Here, we will also state some important ideas of Eppstein's algorithm for the problem of finding the k shortest (possibly non-simple) paths in a directed, weighted graph between two given vertices (kSP). In Chapter 6, we propose a new kSSP algorithm based on Eppstein's kSP algorithm. Like existing algorithms, our new algorithm depends on the Dijkstra algorithm that finds shortest paths in a directed graph. However, in contrast to existing algorithms, it is designed to reduce the number of Dijkstra runs instead of the running time per Dijkstra run in practice. We state the basic approach in Section 6.1, and bring its running time on par with state-of-the-art kSSP algorithms in Section 6.2. Further, practical improvements for the new algorithm are presented in Chapter 7. Our base algorithm necessarily encounters non-simple paths. Our first practical improvement in Section 7.1 lets us identify those non-simple paths more efficiently. The second improvement in Section 7.2 aims to speed up the Dijkstra algorithm by applying reduction tests dynamically, drastically decreasing the size of the graph in the process. We conclude this first main part with a computational study in Chapter 8. We compare our algorithm with the two most efficient existing ones on road graphs, Erdős-Rényi random graphs, and grid graphs as well as synthetic network topology graphs, thereby demonstrating that our algorithm is highly relevant in practice. A paper on our sidetrack-based kSSP algorithm has been published [73] in the proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC 2016). The implementation used in the experiments in this paper is called *SN* in this thesis. The edge pruning technique in Section 7.2 that discriminates the latest implementation *SB* from SN has not been published.

In Part III, we propose a framework for graph problems that can be formulated in a certain logic, the counting monadic second order logic with incidence relation (CMS_2). The framework is derived from Courcelle's famous theorem on such problems. Required part-specific notation and definitions, as well as a review of relevant existing research on the topic are introduced in Section 9.1. Since the subject is already heavy on definitions, we first show how to solve a restricted set of problems, specifically those problems in CMS_2 on *undirected* graphs that only contain a single solution set in Chapter 10. To extend the basic framework to directed hypergraphs and problems with multiple solution sets in Chapter 11, we define hypergraph algebras, which then let us generalize the optimization algorithm on hypergraph algebra parse trees by Courcelle and Mosbah [35] to the k-best enumeration setting. A paper on this framework authored by Eppstein and Kurz was accepted for publication in the proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC 2017); a preprint is available [44].

We finish the thesis in Part IV with conclusions in Chapter 12 and a list of open problems in Chapter 13.

2. Definitions and Notation

This chapter will establish notation for concepts that are used in both main parts. More specific concepts are introduced in their corresponding part.

2.1. General Definitions

For two sets M, N, the *disjoint union* of M and N is only defined if M and N are disjoint, i.e., $M \cap N = \emptyset$, and is denoted by $M \sqcup N$. We denote by $\mathbb{N}$ the set of positive integers, and by $\mathbb{N}_0 = \mathbb{N} \sqcup \{0\}$ the set of non-negative integers. For $i \in \mathbb{N}$, $[i]$ is the set $\{1, \dots, i\}$. The set of real numbers is $\mathbb{R}$, and for $x \in \mathbb{R}$, $\mathbb{R}_{\geq x}$ is the set of real numbers greater than or equal to x. The set $\{\mathbf{false}, \mathbf{true}\}$ of Boolean values is denoted by $\mathbb{B}$.

Let M be a set. We denote the *cardinality* of M by $|M|$, and the *power set* of M by 2^M. A *permutation* on M is a bijection $f\colon M \to M$. A function $f\colon [k] \to M$ for some $k \in \mathbb{N}_0$ is a *sequence of length k* or *k-tuple* over M, usually denoted by $(a_1, \dots, a_k)$ where $f(i) = a_i$ for $i \in [k]$. Similarly, a function $f\colon \mathbb{N} \to M$ is an *infinite sequence*. A *sequence* usually refers to a finite sequence in this thesis. For a sequence S of length n and $i \in [n]$, we refer to the i-th element of S by S_i. Similarly, the i-th element of an infinite sequence, $i \in \mathbb{N}$, is denoted by S_i. For $i \in \mathbb{N}_0$, the set M^i contains all sequences over M of length i, and $M^* := \bigcup_{i=0}^{\infty} M^i$.

A *multiset* (M, f) consists of a nonempty set M and a function $f\colon M \to \mathbb{N}_0$. For $a \in M$, $f(a)$ is the *multiplicity* of a. The *cardinality* of a multiset is the sum of all its multiplicities, $|(M, f)| = \sum_{a \in M} f(a)$.

Let $f\colon M \to N$ be a function, and $M' \subseteq M$. If $N = \mathbb{R}$, $f(M')$ denotes the sum of values under f of all elements of M', $f(M') := \sum_{a \in M'} f(a)$. This notation is primarily used to apply weight functions to solutions, both introduced later. For $N \neq \mathbb{R}$, $f(M')$ denotes the image of M' under f, $\{f(a) \mid a \in M'\}$.

Let $k \in \mathbb{N}$ and $f\colon \mathbb{N}^k \to \mathbb{N}$. We define the *Landau symbols* as follows (with each $\mathbf{n}, \mathbf{n}_0 \in \mathbb{N}^k$):

$$\begin{aligned}
\Omega(f(\mathbf{n})) &= \{g\colon \mathbb{N}^k \to \mathbb{N} \mid \exists\, \mathbf{n}_0, c > 0\, \forall\, \mathbf{n} \geq n_0\colon 0 \leq cg(\mathbf{n}) \leq f(\mathbf{n})\},\\
\omega(f(\mathbf{n})) &= \{g\colon \mathbb{N}^k \to \mathbb{N} \mid \forall\, c > 0\, \exists\, \mathbf{n}_0\, \forall\, \mathbf{n} \geq n_0\colon 0 \leq cg(\mathbf{n}) \leq f(\mathbf{n})\},\\
\mathcal{O}(f(\mathbf{n})) &= \{g\colon \mathbb{N}^k \to \mathbb{N} \mid \exists\, \mathbf{n}_0, C > 0\, \forall\, \mathbf{n} \geq n_0\colon 0 \leq f(\mathbf{n}) \leq Cg(\mathbf{n})\},\\
o(f(\mathbf{n})) &= \{g\colon \mathbb{N}^k \to \mathbb{N} \mid \forall\, C > 0\, \exists\, \mathbf{n}_0\, \forall\, \mathbf{n} \geq n_0\colon 0 \leq f(\mathbf{n}) \leq Cg(\mathbf{n})\}, \text{ and}\\
\Theta(f(\mathbf{n})) &= \{g\colon \mathbb{N}^k \to \mathbb{N} \mid \exists\, \mathbf{n}_0, c, C > 0\, \forall\, \mathbf{n} \geq n_0\colon 0 \leq cg(\mathbf{n}) \leq f(\mathbf{n}) \leq Cg(\mathbf{n})\}.
\end{aligned}$$

Let $(M, \leq)$ be a partially ordered set. An element $a \in M$ is a *maximal (minimal) element* of M if there is no $a' \in M$ with $a \leq a'$ $(a' \leq a)$. If $a \leq a'$ $(a' \leq a)$ for every $a' \in M$, a is a *maximum (minimum) element* of M. If M is linearly ordered, then for $N \subseteq M^*$, we denote by $\operatorname{lex\,min} N$ ($\operatorname{lex\,max} N$) the lexicographically minimum (maximum) element of N.

2.1.1. Computation models

We use *parallel random-access machines* (*PRAM*) as described by Akl [1] (mostly called *Single Instruction stream Multiple Data stream (SIMD) computers with Shared Memory (SM)* by the author) to analyze algorithms. We also follow Cormen et al. [31] in some of the following terminology.

A PRAM has a sequence of *processors* $P_1, \ldots, P_N$; the number N of processors may depend on the input size. Each processor has its own *local memory*. Additionally, *shared memory* is common to all processors, and therefore a means to communicate to each other. Some coordination unit aware of the algorithm orchestrates the execution: In every step, it reads one instruction of the algorithm, and pushes this instruction to all processors. It may choose to perform this instruction only on a subset of processors, but it may not tell different processors to perform different instructions in the same step. The instruction is then performed by the processors concurrently.

Both local and shared memory are organized as an infinite sequence of *cells*. Each cell can store a word of $c \log n$ bits, where $c \in \mathbb{R}$ only depends on the algorithm, and n is the number of bits used to represent the input. The input is initially stored in n cells in shared memory.

We further specify how processors are allowed to access the same cell in shared memory simultaneously. Both read (R) and write (W) access may be exclusive (E) or concurrent (C), resulting in the four combinations EREW, CREW, ERCW and CRCW. For example, the CREW model grants exclusive write access to a memory cell to one processor at any time, but all processors may read any part of the memory concurrently at any time. If multiple processors try to write to the same cell at the same time, we assume that only the store operation of the processor with the lowest index succeeds.

A PRAM with only one processor is also called a *random-access machine* (*RAM*). It is not necessary to specify access control for shared memory for RAMs, and we may as well assume that only global memory is used. Algorithms for RAMs are *sequential*; if the number of processors is a superconstant function of the input size, it is *parallel*.

Atomic instructions are arithmetic operations (add, subtract, multiply, divide, remainder, floor, ceiling, exponentiate with base 2), memory instructions for local or shared memory (load, store, copy), or control flow instructions. *Complex instructions* are composed from atomic and complex instructions. When analyzing the time complexity of an algorithm on a PRAM, an atomic instruction accounts for one step. The time required to perform a complex instruction is the sum of step counts for its subinstructions. The *running time* required to perform an algorithm is the sum of times required for all instructions read by the coordination unit before termination.

For results on space complexity, we use *space-restricted Turing machines* [102]. These have an *input tape* that can only be read, and an *output tape* that can only be written to. We require that an algorithm's result is equal to the content of the output tape when the algorithm terminates. Space requirements are measured on an additional *memory tape* with read and write access. We limit this memory tape on one side, index the cell farthest to that side with 1, and require that the head of the Turing machine starts at index 1 on this tape. The space requirement then corresponds to the highest index on the memory tape that was read from or written to.

2.1.2. Graphs

The following notion of directed graphs is based on the definitions of Diestel [36], and generalized to obtain hypergraphs.

A *hypergraph* G consists of a finite set V of *vertices*, a finite set E of *hyperedges*, and a function $\mathit{vert}\colon E \to (V^*)$. A *feature* of a hypergraph is either a vertex or a hyperedge. Although directed hypergraphs are triples (V, E, vert), we will almost universally declare them as pairs (V, E) after this subsection, since vert is only a technical detail. In contexts where the distinction between a hyperedge e and its corresponding vertex sequence $\mathit{vert}(e) = (v_1, \dots, v_r)$ is not important, we identify e with $\mathit{vert}(e)$. A hypergraph implicitly defines functions head : $E \mapsto V$ and tail : $E \mapsto V$ with $\mathrm{tail}(e) = v_1$ and $\mathrm{head}(e) = v_r$ for $\mathit{vert}(e) = (v_1, \dots, v_r)$. A *loop* is a hyperedge e with $\mathrm{tail}(e) = \mathrm{head}(e)$. Hyperedges $e, e' \in E$, $e \neq e'$, are *parallel* if $\mathit{vert}(e) = \mathit{vert}(e')$. G is *loopless* if no hyperedge is a loop, and *simple* if no two edges are parallel, i.e., *vert* is injective.

Let A be an alphabet, $\leq$ a linear order over A and $\tau\colon A \to \mathbb{N}_0$ a mapping that maps every letter in A to an *order*. For $r \in \mathbb{N}_0$, a *labeled r-interface hypergraph* $(V, E, \mathit{vert}, \mathit{lab}, \mathit{src})$ is a hypergraph together with an edge labeling function $\mathit{lab}\colon E \to A$ where for every hyperedge $e \in E$, the length of its vertex sequence $\mathit{vert}(e)$ is equal to the order $\tau(\mathit{lab}(e))$ of its label, and a sequence $\mathit{src}\colon [r] \to V$ of r designated vertices. Hypergraphs can be considered special cases of labeled r-interface hypergraphs with $r = 0$, $A = [\max\{|\mathit{vert}(e)| \mid e \in E\}]$, and $\mathit{lab}(e) = |\mathit{vert}(e)|$. The *order* of an r-interface hypergraph is r.

A *directed graph* G is a hypergraph where the sequence $\mathit{vert}(e)$ has length 2 for every $e \in E$. For a vertex $v \in V$, $\delta^+(v) = (\{v\} \times V) \cap E$ is the set of *outgoing edges* of v and $\delta^-(v) = (V \times \{v\}) \cap E$ is the set of *incoming edges* of v. The *outdegree* $d^+(v)$ of v is the number of outgoing edges $|\delta^+(v)|$, and its *indegree* $d^-(v)$ is the number of incoming edges, $|\delta^-(v)|$. A vertex u is a *predecessor* of v if there is an edge $e \in E$ with $\mathit{vert}(e) = (u, v)$, and a *successor* if there is an edge $e \in E$ with $\mathit{vert}(e) = (v, u)$.

An *undirected graph* G is a pair (V, E), where V is a finite set of vertices, and E is a finite set of (undirected) edges, which are two-element subsets of V. A *feature* of an undirected graph is either a vertex or an undirected edge. Because of the set semantics used here, an undirected edge cannot occur twice in E, and it cannot be a (one-element) loop $\{v, v\}$. Hence, we consider undirected graphs to be simple and loopless by definition. Vertices $u, v \in V$ are neighbors if there is an edge $\{u, v\} \in E$; we denote by $\delta(v)$ the set of neighbors of v. The number of neighbors $|\delta(v)|$ of v is its *degree*.

Let $G = (V, E)$ be an undirected graph and $v \in V$ be a vertex. A vertex v is an *endpoint* of an edge e if $v \in e$. An edge $\{u, v\} \in E$ is *incident* to v. A vertex $u \in V$,

$u \neq v$, is *adjacent to* v if there is an edge that is incident to both u and v, i.e., $\{u, v\} \in E$. Two edges are adjacent to each other if there is a vertex they are both incident to.

Let $\mathcal{G}$ be a class of graphs such that the number of vertices is unbounded. If there is $c \in \mathbb{R}$ such that for every $(V, E) \in \mathcal{G}$ we have $|E| \leq c|V|$, $\mathcal{G}$ is a *class* or *family of sparse graphs*. If there is $c \in \mathbb{R}$ such that $|E| \geq c|V|^2$ for each $(V, E) \in \mathcal{G}$ instead, $\mathcal{G}$ is a *class* or *family of dense graphs*.

A hypergraph $G = (V, E, \mathit{vert})$ is a *supergraph* of a hypergraph $G' = (V', E', \mathit{vert}')$ (and G' is a *subgraph* of G) if $V' \subseteq V$, $E' \subseteq E$, and $\mathit{vert}(e) = \mathit{vert}'(e)$ for each $e \in E'$. Let $W \subseteq V$ be a node subset. The edge subset of E *induced* by W is the set $\{e \in E \mid \forall i \in [|\mathit{vert}(e)|] : \mathit{vert}(e)_i \in W\}$. The hypergraph $G[W] = (W, E[W], \mathit{vert}|_{E[W]})$ is called the induced subgraph of G w.r.t. W. *Removing a vertex subset* $V' \subseteq V$ from G results in the subgraph $G - V' := G[V \setminus V']$. *Removing an edge subset* $E' \subseteq E$ from G results in the subgraph $G - E' := (V, E \setminus E', \mathit{vert}|_{E \setminus E'})$. We conveniently write $G - v$ (instead of $G - \{v\}$) or $G - e$ (instead of $G - \{e\}$) if only a single vertex v or a single vertex e is removed. Supergraphs and (induced) subgraphs are defined for undirected graphs accordingly.

An *edge (vertex) weight function* is a mapping $c\colon E \to M$ ($c\colon V \to M$) from the edge set (node set) of a directed hypergraph (undirected graph) to some ordered set M. The terms *cost function* and *weight function* are used interchangeably. *Uniform weight functions* map every element of the domain to 1.

Edge weighted hypergraphs $G = (V, E, \mathit{vert})$ are also the sole reason for us to define E as separate objects that are mapped to vertex sequences. Alternatively, we could have defined the edge set E of hypergraphs as multisets (V^*, g). Formally, this would eliminate distinct incarnations of the same edge. There might be an edge (u, v) with $g((u, v)) > 1$, but we would not be able to have two such edge with different edge weights. Therefore, edges are objects on their own, so we can have $c(e_1) \neq c(e_2)$ even for $\mathit{vert}(e_1) = \mathit{vert}(e_2)$.

2.1.3. Paths, Cycles and components

Let $s, t \in V$ be vertices of a directed graph $G = (V, E, \mathit{vert})$. For $s \neq t$, a sequence $p = (e_1, \ldots, e_n)$ of edges with $\mathit{vert}(e_i) = (u_i, v_i)$, $u_1 = s$, $v_n = t$ and $v_i = u_{i+1}$ for $i \in [n-1]$ is an *s-t walk* in G. For every $i \in [n]$, the sequence $(e_1, \ldots, e_i)$ is a *prefix* of p and $(e_i, \ldots, e_n)$ is a *suffix* of p. It is an *s-t path* if no edge occurs twice, i.e., $e_i = e_j \Rightarrow i = j$. Note that two distinct edges e_i, e_j may occur on a path even if they connect the same two vertices, i.e., $\mathit{vert}(e_i) = \mathit{vert}(e_j)$. The set of all s-t paths in G is $\mathcal{P}_{s,t} = \mathcal{P}_{s,t}(G)$. The term $s \rightsquigarrow t$ for $s, t \in V$ states that an s-t path exists in G. For $s = t$, such a sequence is called a *cycle* instead. The directed graph G is *acyclic* if there is no cycle in G.

The number of edges in a path or cycle is denoted by $|p|$. The *number of occurrences* of a vertex v on cycle p is the number $|\{i \mid u_i = v\}|$ of edges on p whose tails are v. If p is a path, we add one to this number if $t = v$. If the number of occurrences is at least one, v *occurs* on p. The subgraph of G induced by the set of vertices that do not occur on p is $G - p$. A path or cycle is *simple* if every vertex occurs at most once. The set of all simple s-t paths in G is $\mathcal{P}^s_{s,t} = \mathcal{P}^s_{s,t}(G)$.

The *length* $c(p)$ of a path $p = (e_1, \dots, e_r)$ in a graph $G = (V, E)$ w.r.t. an edge weight function $c\colon E \to \mathbb{R}$ is $c(\{e_1, \dots, e_r\})$. In the absence of an edge weight function, we consider r to be the path's length. An s-t path with minimum length for vertices $s, t \in V$ is a *shortest s-t path.* The *distance* $d_c(s, t)$ between s and t w.r.t. c is the length of a shortest s-t path w.r.t. c, or ∞ if no such path exists. The distance of two vertices of a graph with maximum distance is the graph's *diameter.* In contexts where s, t or c are evident, we will regularly omit their corresponding appositions (e.g., *shortest path* instead of *shortest s-t path w.r.t. c*).

Paths and cycles are defined for undirected graphs accordingly. An undirected cycle p is *trivial* if $|p| = 2$. An undirected graph G is *acyclic* if every simple cycle in G is trivial. Acyclic undirected graphs are also called *forests.*

An undirected graph $G = (V, E)$ is *connected* if $u \rightsquigarrow v$ for every $u, v \in V$. Maximal connected induced subgraphs of G are *connected components* of G. If $u \rightsquigarrow w$ in G for $u, v, w \in V$, but $u \not\rightsquigarrow w$ in $G - v$, v *separates* u from w. If v separates any two vertices of G, i.e., removing v increases the number of connected components, then v is a *split vertex* of G. A connected forest is an undirected *tree.* A subgraph of G that is also a tree is also called a *subtree* of G.

Vertices in a tree are also called *nodes.* A *leaf* is a node with degree 1. A *rooted tree* is a tree with designated *root node*, which is usually not considered a leaf. In a rooted tree where u separates v and r, u is an *ancestor* of v and v is a *descendant* of u. If additionally $\{u, v\} \in E$, u is the *parent* of v and v is a *child* of u. A *subtree rooted at u*, $T(u)$, is the subgraph of a rooted tree that is induced by u and all its descendants.

A directed graph $G = (V, E)$ is *strongly connected* if $u \rightsquigarrow v$ for every $u, v \in V$. A *strongly connected component* is a maximal strongly connected induced subgraph of G. Both connected components for undirected graphs and strongly connected components for directed graphs can be computed in time $\mathcal{O}(|V| + |E|)$, both algorithms being based on depth-first search [31].

Directed trees come in two flavors, both of which are directed acyclic graphs and have a designated root node r. In an *arborescence* or *out-tree*, there is exactly one r-v path in the graph for every $v \in V$. In contrast, in an *in-tree*, there is exactly one v-r path in the graph for every $v \in V$.

The depth of an out- or undirected tree T rooted in node r is the maximum distance from r to any node in T w.r.t. uniform edge weights. For in-trees, it is the maximum distance from any node in T to r w.r.t. uniform edge weights. The depth of an undirected tree without a root is the same as the depth of the same tree rooted in a node that minimizes depth (and therefore approximately half the tree's uniform diameter).

A subgraph $G' = (V', E')$ of a graph $G = (V, E)$ is said to *span* V'. A *subtree* is a subgraph that is also a tree. It *spans* G if $V = V'$. A subtree that spans G is a *spanning tree* of G. Let $r \in V$. A *shortest path tree* (*SP tree*) *from* r in G w.r.t. an edge weight function c is a subtree of G in which distances from r to any vertex match distances in G. An SP tree *to* r in G w.r.t. c is a subtree in which distances from any vertex to r

match those in G. Note that the vertex sets of G and a shortest path tree of G differ if there is no path from some vertex to r (or from r to some vertex).

Let $G = (V, E)$ be an undirected graph. A *tree decomposition* of G is a pair (T, b), where $T = (U, F)$ is an undirected tree and $b\colon U \to (2^V \setminus \{\emptyset\})$ maps every node of T to a vertex subset of G – called *bag* – such that

- b covers V, i.e., $\bigcup b(U) = V$,
- every edge $e \in E$ is subset of at least one bag, i.e., $\forall\, e \in E : \exists\, u \in U : e \subseteq b(u)$, and
- the subgraph of T induced by the set of bags that contain any vertex $v \in V$ is connected, i.e., $\forall\, u_l, u_c, u_r \in U, v \in V : (v \in b(u_l) \land v \in b(u_r) \land u_c$ separates u_l and $u_r) \Rightarrow v \in u_c$.

The *width* of a tree decomposition $(T = (U, F), b)$ is one less than the size of its largest bag, $\max\{|b(u)| \mid u \in U\} - 1$. The *treewidth* $\operatorname{tw}(G)$ of an undirected graph G is the minimum width a tree decomposition of G can have. For any fixed w, it is possible to recognize graphs with treewidth at most w, and to compute a tree decomposition of minimum width, in linear time [12].

In a *rooted tree decomposition* (T, b) of G, we assume T to be rooted. A node u of a rooted tree decomposition *represents* the subgraph of G that is induced by the union of the bags of all descendants of u and the bag of u itself. Thus, the root of T represents G and a leaf u of T represents $G[b(u)]$.

Bodlaender [14] showed that for every fixed $w \in \mathbb{N}$ there is some $c = c(w) \in \mathbb{R}$ such that every undirected graph $G = (V, E)$ with treewidth at most w has a rooted tree decomposition $(T = (U, F), b)$ such that:

- $|U| \leq c|V|$,
- the depth of T is at most $c \log(|V|)$,
- every node of T has at most two child nodes, and
- the width of (T, b) is at most $3w - 2$.

Tree decompositions with these properties are *shallow*. For graphs with bounded treewidth, it can be computed in linear time on a RAM, or in logarithmic time with $\mathcal{O}(|V|^{3\operatorname{tw}(G)+4})$ many processors on a CRCW PRAM [15].

2.2. Combinatorial Optimization

Let $k \in \mathbb{N}$. A *combinatorial optimization problem* according to Nemhauser and Wolsey [84] is the evaluation of the term

$$\max\{c(S) \mid S \in \mathcal{S}\},$$

where $c\colon [k] \to \mathbb{R}$ is a cost function, and $\mathcal{S} \subseteq 2^{[k]}$ is a set of *feasible solution* over $[k]$. Combinatorial optimization problems of this kind are also called *combinatorial maximization problems* in this thesis. In contrast, a *combinatorial minimization problem* is the evaluation of the term

$$\min\{c(S) \mid S \in \mathcal{S}\}$$

with c and $\mathcal{S}$ as above. A subset S of $[k]$ is a *solution*, and *infeasible* if $S \notin \mathcal{S}$. A combinatorial problem is *infeasible* if $\mathcal{S} = \emptyset$, *feasible* if $\mathcal{S} \neq \emptyset$, and *uniquely solvable* if $|\mathcal{S}| = 1$.

The *value* of a solution S is its cost $c(S)$. A solution S is *optimal* for a maximization (minimization) problem if $c(S') \leq c(S)$ ($c(S') \geq c(S)$) for every $S' \in \mathcal{S}$. The value of an optimal solution is the *optimal value*. Note that there may be several optima, which all share the unique optimal value.

The corresponding *search problem* asks for a feasible solution $S \in \mathcal{S}$ that assumes the maximum value under c, i.e.,

$$\arg\max\{c(S) \mid S \in \mathcal{S}\}.$$

Combinatorial optimization problems on graphs with n vertices and m edges can be modeled with a bijective function into $[k]$. For problems asking for an edge subset, we might have $g\colon E \to [m]$; using $g\colon V \to [n]$, we can model problems that ask for vertex subsets. To ease notation, we will abstract away from g, and therefore assume that solutions are vertex or edge subsets. In the same manner, we model problems on multiple sets. For example, instead of defining a mapping from $E \times [3]$ to $[3m]$ for a problem that asks for three edge subsets, we assume that $\mathcal{S}$ consists of triples of edge subsets. *Algorithms* solve *families of combinatorial optimization problems* with common compact *input* format, e.g., an encoding format of a graph. We can transform the input to $[k]$, c and $\mathcal{S}$ explicitly, but this is usually neither required nor desired due to a potentially exponential blowup.

For a language L', subsets of the Cartesian product $L' \times \mathbb{N}$ are *parameterized languages*. A parameterized language L is *fixed-parameter tractable* if there is an algorithm that, for every input (I, k), computes $(I, k) \in L$ in time $f(k) \cdot p(|I|)$, where f is an arbitrary function, p is a polynomial function and $|I|$ is the length of I. FPT is the class of fixed-parameter tractable parameterized languages.

3. Basic Concepts

In this chapter, we introduce some basic concepts. Mostly, they do not contain new definitions or notation, but are still relevant for both main parts.

3.1. Data Structures

We give an overview of the data structures that play an important part in this thesis.

3.1.1. Graph Representations

Graphs as defined in Section 2.1.2 are merely a mathematical concept and have to be encoded in order to be useful for algorithms. Formally, the choice of encoding of the input (and the output, too) is part of a definition of a problem, and can have an impact on the running time of various graph operations, or even the existence of some graph operations. In this section, we will discuss the most common ones.

Let $G = (V, E)$ be a simple directed graph on the linearly ordered sets $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$. The *adjacency matrix* of G is a square matrix $M \in \mathbb{B}^{n \times n}$ with $M_{ij} = \textbf{true} \Leftrightarrow (v_i, v_j) \in E$. While checking for the existence of an edge between two given vertices can be done in constant time, this representation has some downsides. First, the size of adjacency matrices only depends on the number of nodes, leading to a quadratic blowup in size for sparse graphs. Most of the time, we give running times in terms of n and m. For example, we would give the running time of a breadth-first search as $\mathcal{O}(m + n)$. However, n^2 grows faster than $n + m$ for sparse graphs, so the running time of $\mathcal{O}(n + m)$ would actually be sublinear in the length of the encoding length of an adjacency matrix. Another downside is the lack of efficiency for graph operations that are far more common in graph algorithms than the check $(v_i, v_j) \in E$. In particular, we cannot enumerate $\delta^+(v)$ for some vertex v in time $|\delta^+(v)|$.

An *adjacency list* of a vertex v is a linked list of all outgoing edges of v. The *adjacency-list representation* of G consists of an n-length array A and an adjacency list of each vertex. The i-th entry of A is the adjacency list of v_i. This representation mitigates the disadvantages of adjacency matrices mentioned above. We cannot check $(v_i, v_j) \in E$ in constant time any more, but this operation is rarely needed anyway, and it won't be used by any algorithm discussed in this thesis.

Another advantage of adjacency lists are their mutability. We can easily insert new edges in constant time. Deleting an existing edge (v_i, v_j) can also be done in constant time if we already have the required references to modify the adjacency list of v_i accordingly, and takes time $\mathcal{O}(d^+(v))$ if we do not have these references.

If this mutability is not required, we can use a more compact variant of adjacency lists. First, a function $g\colon V \to M$ for an arbitrary set M can be represented by an n-length array such that the i-th entry contains $g(v_i)$. A function on E can be represented accordingly. We now require that E is sorted by the edges' tail vertex in ascending order. A *simple forward star representation* of G is the representation of the head function on E, and a first-edge function $f\colon V \to E$ with $f(v) = \arg\min_{e_i}\{e_i \mid \operatorname{tail}(e_i) = v\}$. Simple forward stars have all the benefits of adjacency-list representations for static graphs without the overhead introduced by $\mathcal{O}(m+n)$ many pointers.

We can efficiently enumerate the outgoing edges of a given vertex and their corresponding heads with a forward star. However, finding incoming edges of a vertex v is not supported directly. We have to enumerate the whole edge set and check for each edge e if v is the head of e. Alternatively, we can use a simple forward star of the graph obtained from G by swapping head and tail of each edge. This representation is the *simple reverse star* of G.

The algorithms discussed in this thesis require efficient access to both the incoming and outgoing edges of vertices. Consider a directed graph $G = (V, E)$ with $V = \{v_1, \dots, v_4\}$ and $E = \{e_1 = (v_1, v_4), e_2 = (v_2, v_3)\}$. Since edges are sorted by their tails, e_1 precedes e_2 in the forward star of G, but e_2 precedes e_1 in the reverse star of G. Now consider an edge function $f\colon E \to M$ for an arbitrary M. As noted before, we store f as an $|E|$-element array where the i-th entry is $f(e_i)$. Therefore, we cannot use edge functions on E directly with edge indices obtained while operating on the reverse star. This problem is circumvented by an additional edge function $\operatorname{trace}\colon [|E|] \to [|E|]$ such that the i-th edge in the reverse star is the same as the $\operatorname{trace}(i)$-th edge in the forward star. The value of f at the i-th edge e'_i in reverse star representation is then $f(\operatorname{trace}(i))$. The combination of a simple forward star, a simple reverse star and the trace function is simply called *forward star* in this thesis.

3.1.2. Priority Queues

We use priority queues in several places in this thesis. Formally, a priority queue with underlying set A is a set $S \subseteq A$ where each *item* is associated with a *key* from a linearly ordered set of keys. We are able to *push* a new item a to a priority queue, resulting in $S' = S \cup \{a\}$. Let $a \in S$ be an item with minimum associated key. Priority queues allow to look up a, which does not modify S, and to *extract* a from S, resulting in $S' = S \setminus \{a\}$.

Some priority queues additionally offer a way to *decrease the key* of an item that is already managed, resulting in a different smaller key being associated with that item. The only algorithm in this thesis, however, that depends on the decrease-key operation is the Dijkstra algorithm for computing shortest paths in a directed, weighted graph, so we do not use this operation directly.

There exist several implementations of priority queues, and most of them are based on *heaps*. Heaps, too, manage sets of items and induce a parent-child relationship upon S. They satisfy the *min-heap property*, i.e., the key of a parent is less than or equal to the key of a child. The operations supported by heaps roughly parallel those of priority queues.

In this thesis, we make use of k-ary heaps, Fibonacci heaps, pairing heaps and interval heaps. The internals of these heaps are beyond our scope. We refer the reader to Cormen et al. [31] for details on k-ary and Fibonacci heaps, to Fredman [51], Iacono [64] and Pettie [87] for pairing heaps and to Sahni [94] for interval heaps. We settle for highlighting the differences between the four heaps.

Fibonacci heaps display the best worst-case running times, with (amortized) constant time for all operations except for minimum-extraction, which requires amortized time $\mathcal{O}(\log n)$ for heaps with n items. The decrease-key operation is asymptotically slower for pairing heaps (amortized time $\mathcal{O}(\log n)$), and just as efficient as Fibonacci heaps for all other operations. Every operation on k-ary heaps requires time $\mathcal{O}(\log n)$ except for the constant-time lookup of the minimum-key item. Interval heaps generalize binary heaps (k-ary heaps with $k = 2$) and therefore inherit their running time behavior.

The k-ary, Fibonacci, and pairing heaps were implemented and compared experimentally in a recent study by Larkin, Sen and Tarjan [75]. Although Fibonacci heaps are superior to the other two in theory, they performed worst in practice. They are primarily used to give lower upper bounds; e.g., Dijkstra's algorithm requires $\mathcal{O}(m + n \log n)$ running time with Fibonacci heaps, but $\mathcal{O}(m \log n)$ time with any of the other three heaps. Pairing and k-ary heaps performed best when used in the Dijkstra algorithm. In tentative experiments, we found that the pairing heap of the Open Graph Drawing Framework [25] is more efficient than the binary heap implementation provided by the GNU C++ standard library at version 4.9.3.

Finally, interval heaps support an operation that the others do not support. In addition to querying and extracting the item with minimum key, we can also query and extract the item with maximum key. If we know that we only extract the minimum-key item at most M times, or that we are only interested in items whose key is below some dynamically updated upper bound (which is both the case in Part II), we can utilize the extract-max to restrict the heap to a reasonable size. Running times are symmetric for the minimum and maximum key item, so querying and extracting both can be done in time $\mathcal{O}(\log n)$.

3.1.3. Persistent Arborescences

Modifying operations of a data structure are usually not reversible. After inserting a new element into a list, tree, or hash map, we forget about its former state and cannot tell which of the elements was inserted last, so we cannot restore a former state. If we keep track of the modifications performed on a data structure, we are usually not able to act on old states as efficiently as on the current state: we first have to restore the old state, which is often inefficient. This is true for both modifying and reading operations. Data structures with this destructive nature are *ephemeral*.

In some situations, however, we need some means to access older states of a data structure efficiently. This is where *persistent* data structures come into play. We distinguish *update* operations that modify the data structure from *querying* operations that only extract information from the data structure without modifying it. All or some update operations cause the creation of a new *version* of the data structure which we

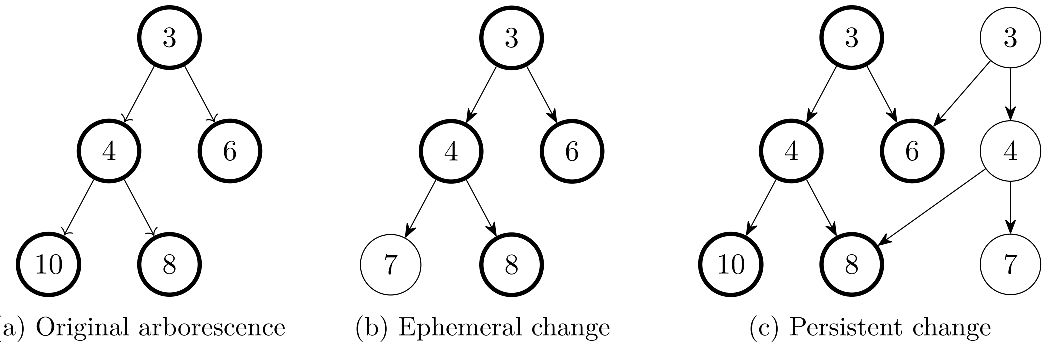


Figure 3.1.: Changing the value stored in one node in an arborescence both in an ephemeral way where the old state is lost, and in a persistent way. Nodes in their original state are thick.

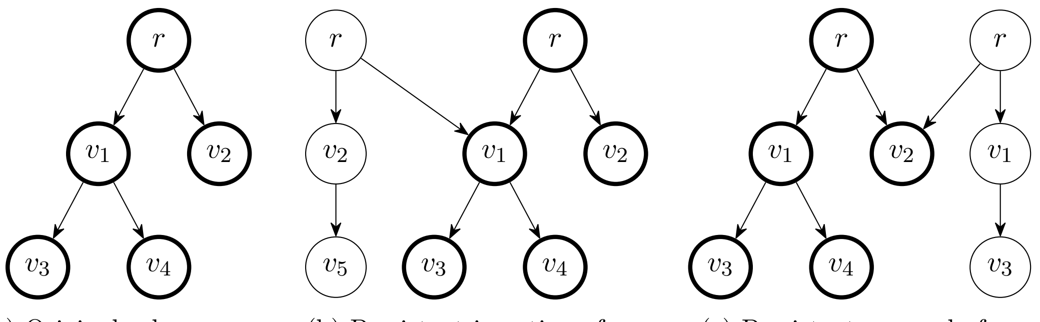
can then operate upon. Every operation of a persistent data structure gets an additional parameter that references the target version. Updating one version does not affect other existing versions in any way. It is therefore possible to derive several new versions from an existing one by using different update operations on the same version. Old versions can be queried, too, without other versions interfering.

We consider ephemeral and persistent variants of a binary arborescence. Let W be the set of nodes that are affected by a change. A new version is created by cloning the nodes in W as well as their ancestors. Pointers among the clones are added accordingly. Additionally, the clones get pointers to those child nodes of the corresponding original node that have not been affected. The change can now be performed solely on the clones. Note that no pointer of the original arborescence is altered in the process.

One such data structure by Driscoll et al. [38] used in several places in this thesis is the *persistent arborescence*. The persistent arborescence is a pointer-based representation of an arborescence. Every node has a separate pointer to each of its child nodes, but no pointer to its parent. The roots of the arborescences serve as references to versions. When we use data structures derived from persistent arborescences, we describe updates and queries for the ephemeral variant using only the root of the arborescence. Query operations can then be applied to versions without modification. We give a generic way to make ephemeral changes persistent.

An example use of the persistent arborescence can be seen in Figure 3.1. Figure 3.1a shows the original version of the arborescence. Assume that each node stores some natural key as denoted by the number inside the node. We want to change the leftmost node to store 7 instead of 10. The ephemeral version of the data structure destroys the old state, as seen in Figure 3.1b, making it impossible to reestablish the original state. In contrast, creating clones of the affected node and its ancestors does not influence the original data structure at all. The new, thin root node is a reference to the new version, while the black root node remains a reference to the old one.

Persistent arborescences are not restricted to updates that change the value associated to nodes. Node insertion and removal can be made persistent as well. Examples for this



(a) Original arborescence (b) Persistent insertion of v_5 (c) Persistent removal of v_4

Figure 3.2.: Persistently inserting a node into a heap or removing a node from a heap without destroying the old state. Nodes in their original state are thick.

can be seen in Figure 3.2.

Persistent data structures are used during the initialization phase of Eppstein's k shortest path algorithm [40] and in our k-best algorithm for problems in counting monadic second-order logic on graphs with bounded treewidth in Part III. In the first case, we always have $|W| = 1$, while in the latter case, all nodes in W form a path from the root to a leaf node. In both cases, the affected nodes, i.e., the nodes in W and their ancestors, form a path. The technique to make the arborescence persistent is therefore called *path-copying persistence*. Using path-copying persistence, the time to create a new version of the arborescence is only linear in the depth of the deepest affected node, instead of being linear in the number of nodes in the arborescence.

3.2. k-Best Optimization

In *k-best optimization*, we are interested in a sequence of solution values. Consider a combinatorial optimization problem defined by its feasible solutions $\mathcal{S}$ and a cost function c. We obtain its canonical *k-best optimization problem* by augmenting it by $k \in \mathbb{N}$. Instead of merely finding the optimal value among $\mathcal{S}$, we now want to find a sequence of k values of pairwise distinct solutions $(S_1, \dots, S_k)$ of length k. We require that $c(S_i) \geq c(S_{i+1})$ for $i \in [k-1]$, and that $c(S_k) \geq c(S')$ for every $S' \in \mathcal{S} \setminus \{S_1, \dots, S_k\}$. It is equivalent to the evaluation of the term

$$\min\nolimits_{\mathrm{k}} \mathcal{S} = \operatorname{lex\,min}\{(c(S_1), \dots, c(S_k)) \mid S_i \in \mathcal{S} \text{ pairwise distinct}\}.$$

The corresponding search problem asks for a tuple $(S_1, \dots, S_k)$ of pairwise distinct solutions with $\min_{\mathrm{k}} \mathcal{S} = (c(S_1), \dots, c(S_k))$. We assume throughout this thesis that $k \leq |\mathcal{S}|$. Otherwise we would have to replace k in the definitions above with $k' = \min\{k, |\mathcal{S}|\}$.

We can think of k either as fixed as part of a problem description, or as part of the problem instance. We describe the latter as *general* k as opposed to *fixed* or *constant* k. Note that there cannot be a polynomial time algorithm for the canonical k-best version of any combinatorial optimization problem for general k if k grows polynomially in its own encoding length in the input. This is the case for unary encodings of k. However, we would naturally expect binary encoding, in which case k grows exponentially in its encoding length. Since we have to output k solutions of at least one bit, we require $\Omega(k)$ output time, which is exponential in the input length. The problem is not mitigated by considering k' instead because $|\mathcal{S}|$ can grow exponentially in the input length, too.

We will drop the requirement of the k solutions being output in increasing order of their cost in some contexts. This can be useful, e.g., if all solutions are presented to a user equally at the same time, as would be the case in our motivational example in Chapter 1.

3.2.1. Solution Space Partition

Efficient algorithms for finding optimal solutions for graph problems are usually problem specific. In contrast, approaches for generalizations to find the k best solutions are quite universal. Let $\mathcal{S} \subseteq 2^M$ be the set of feasible solutions for some combinatorial optimization problem with universal set M. We denote by $\mathcal{S}(I, O) := \{S \in \mathcal{S} \mid I \subseteq S, S \cap O = \emptyset\}$ the set of feasible solutions $S \in \mathcal{S}$ that include all elements of set $I \subseteq M$ and do not include any element of set $O \subseteq M$. The optimization problem that corresponds to $\mathcal{S}(I, O)$ is a *subproblem* of the original problem. The k best solutions in $\mathcal{S}$ can be found by iteratively refining a partition of $\mathcal{S}$, and computing best or second-best solutions for the individual cells. We maintain a partition $(\mathcal{S}_1, \dots, \mathcal{S}_r)$ of $\mathcal{S}$, initially with $r = 1$ and $\mathcal{S}_1 = \mathcal{S} = \mathcal{S}(\emptyset, \emptyset)$. In each iteration, one of the cells of the partition is selected and refined.

Murty [82] and Lawler [76] proposed a framework that requires an algorithm for finding optimal solutions for subproblems. Let $S = I \sqcup \{a_1, \dots, a_n\}$ be optimal in $\mathcal{S}(I, O)$, and $M \setminus (S \sqcup O) = \{b_1, \dots, b_m\}$. Then $\mathcal{S}(I, O) \setminus \{S\}$ can be further partitioned into these sets

$$\mathcal{S}(I \sqcup \{a_1, \dots, a_{i-1}\}, O \sqcup \{a_i\}) \qquad \text{for } i \in [n], \tag{3.1a}$$

$$\mathcal{S}(S \sqcup \{b_i\}, O \sqcup \{b_1, \dots, b_{i-1}\}) \qquad \text{for } i \in [m]. \tag{3.1b}$$

Each of these sets is a solution set of a subproblem.

The Murty/Lawler framework maintains a partition of $\mathcal{S}$, initially $(\mathcal{S})$, and for each cell an optimal solution for the corresponding subproblem. Those cells whose optimal solution has not been determined to be one of the k best solutions yet are stored in a candidate priority queue. The priority of a cell is the value of its optimal solution. In each iteration, the next cell is extracted from the priority queue. Its optimal solution is added to the output set of solutions. The cell is then further partitioned as described above, and the algorithm for finding optimal solutions for the corresponding subproblems is used to find optimal solutions for the subcells. Finally, the cell is replaced by its subcells in the partition, and the subcells are added to the priority queue. After k iterations, or if the priority queue runs empty, the algorithm stops.

This approach generates at most $|M|$ subproblems in each iteration, and computes optimal solutions for them. For problems where one feasible solution cannot be a subset

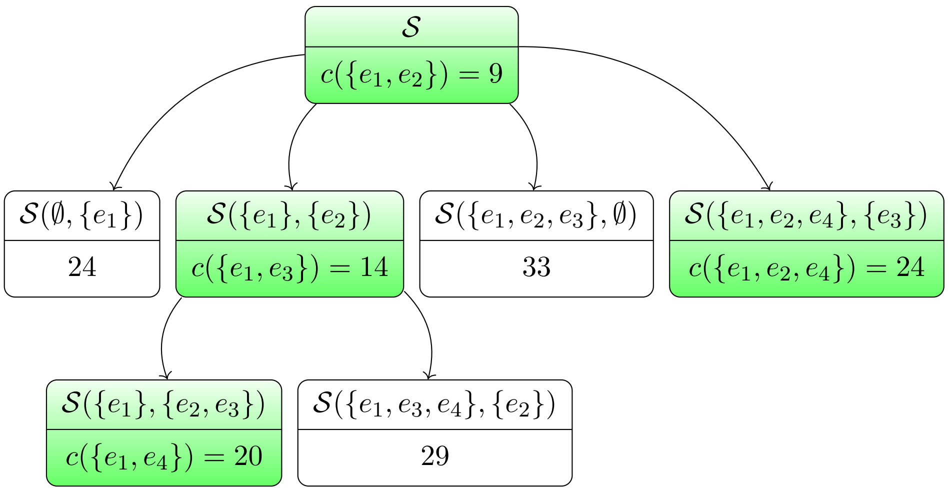

$\mathcal{S}$
$c(\{e_1, e_2\}) = 9$
$\mathcal{S}(\emptyset, \{e_1\})$
24
$\mathcal{S}(\{e_1\}, \{e_2\})$
$c(\{e_1, e_3\}) = 14$
$\mathcal{S}(\{e_1, e_2, e_3\}, \emptyset)$
33
$\mathcal{S}(\{e_1, e_2, e_4\}, \{e_3\})$
$c(\{e_1, e_2, e_4\}) = 24$
$\mathcal{S}(\{e_1\}, \{e_2, e_3\})$
$c(\{e_1, e_4\}) = 20$
$\mathcal{S}(\{e_1, e_3, e_4\}, \{e_2\})$
29

Figure 3.3.: Murty/Lawler heap of subproblems. Rounded rectangles represent subproblems, with the set of feasible solutions in the upper part and the value of an optimal solution in the lower part. Green subproblems contain one of the $k = 4$ best solutions. The tree is a heap w.r.t. the value of the optimal solution. The root problem is a combinatorial toy optimization problem over $M = \{e_1, \ldots, e_4\}$ with feasible solutions $\{S \subseteq M \mid (\{e_2, e_3\} \subseteq S \vee e_4 \notin S) \wedge S \neq \{e_2, e_4\}\}$ and $c(e_1) = 5$, $c(e_2) = 4$, $c(e_3) = 9$, $c(e_4) = 15$.

of another, like the k shortest simple path problem, the solution sets in (3.1b) are always empty and can therefore be omitted. The number of subproblems to be solved in an iteration is then equal to the size of the solution selected in that iteration. This algorithm was proposed independently for the k shortest simple path problem by Yen [105].

An example heap of subproblems for the Murty/Lawler approach can be seen in Figure 3.3. Subproblems whose optimal solutions are among the $k = 4$ best solutions are highlighted in green.

A framework that relies on an algorithm for finding the two best solutions of a subproblem was proposed by Gabow [54] for the k best spanning tree problem, and then generalized for combinatorial optimization problems by Hamacher and Queyranne [59]. We keep the partition of $\mathcal{S}$ as well as the priority queue with subproblems from the last approach. In contrast, however, we use the value of the second-best solution instead of the optimal one as the priority. Let $S(I, O)$ be a subproblem extracted from that priority queue, and let S_1, S_2 be the optimal and second-best solution in $S(I, O)$, respectively. Further, let $x \in S_1 \oplus S_2$ be a *pivot* element in the symmetric difference of S_1 and S_2. We can now replace $\mathcal{S}(I, O)$ with one subproblem $\mathcal{S}(I \sqcup \{x\}, O)$ where all feasible solutions contain the pivot, and one subproblem $\mathcal{S}(I, O \sqcup \{x\})$ where each feasible solution does not contain the pivot. We compute the two best solutions for both these subproblems, and push them to the priority queue of candidates unless they are uniquely solvable.

Initially, the partition of $\mathcal{S}$ is again $(\mathcal{S})$, and we compute the two best solutions for $\mathcal{S}$. The output set is initialized with the optimal solution we just found; the candidate

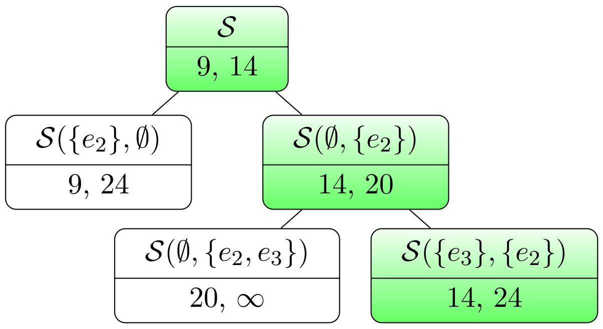

$\mathcal{S}$
9, 14
$\mathcal{S}(\{e_2\},\emptyset)$
9, 24
$\mathcal{S}(\emptyset,\{e_2\})$
14, 20
$\mathcal{S}(\emptyset,\{e_2,e_3\})$
20, ∞
$\mathcal{S}(\{e_3\},\{e_2\})$
14, 24

Figure 3.4.: Hamacher/Queyranne heap of subproblems. Rounded rectangles represent subproblems, with the set of feasible solutions in the upper part and the value of an optimal solution and a second-best solution in the lower part. Green subproblems contain one of the $k = 4$ best solutions. The lower white subproblem is uniquely solvable, so the value of the second-best solution is assumed to be ∞. The tree is a heap w.r.t. the value of the optimal solution and w.r.t. the value of the second-best solution. The root problem is a combinatorial toy optimization problem over $M = \{e_1, \ldots, e_4\}$ with feasible solutions $\{S \in 2^M - \{e_2, e_4\} \mid \{e_2, e_3\} \subseteq S \vee e_4 \notin S\}$ and $c(e_1) = 5$, $c(e_2) = 4$, $c(e_3) = 9$, $c(e_4) = 15$.

priority queue is initialized with the original problem. We then extract and process $k-1$ subproblems from the candidate priority queue as described above. For each extracted subproblem, we add the second-best solution to the output set. If no candidates are left, the algorithm terminates prematurely.

In this second approach, we generate two subproblems in each of the $k-1$ iterations, resulting in $2k-1$ executions of the two-best algorithm for subproblems. The number of subproblems is therefore much smaller than in the first approach. We need to be able to compute the second-best solution in addition to the optimal one. However, this is not a stronger requirement, since we can simply use the first framework to compute the second-best solution, relying solely on an algorithm that solves subproblems to optimality. However, finding the second-best solution might be slower than finding the optimal one. An example for this is the k shortest simple path problem, as described in Section 4.2.

The Hamacher/Queyranne heap for the problem from Figure 3.3 can be seen in Figure 3.4. Both figures contain exactly those subproblems that need to be solved in order to be sure that the four best solutions have been found. We have to solve seven subproblems using the Murty/Lawler approach, and five subproblems using the Hamacher/Queyranne approach. Keep in mind, though, that solving a subproblem to optimality as in the Murty/Lawler approach might be easier than finding the two best solution values as required by the Hamacher/Queyranne approach.

Finally, we consider the tree of subproblems induced by the two above approaches. The root of the tree is the original problem whose solution set is $\mathcal{S}$. The children of each node are the subproblems generated by the frameworks. Leaves of the tree are uniquely solvable subproblems. The tree induced by the first approach is a $|M|$-ary min-heap of subproblems w.r.t. the optimal solution value of a subproblem. In the second approach, the tree is a binary min-heap of subproblems w.r.t. the value of the second-best solution.

Above, the selection of the k best solutions from these heaps is described in terms of a priority queue, but this is only one best-first search. In addition to generating and solving the subproblems, we require time $\mathcal{O}(k \log k)$ just to maintain the priority queue.

Frederickson [50] proposed an optimal best-first search algorithm for heaps. It finds the k smallest values in an d-ary min-heap in time $\mathcal{O}(kd)$. This results in time $\mathcal{O}(|M| \cdot k)$ and $\mathcal{O}(k)$ time for the first and second approach, respectively, giving the binary-tree approach another advantage. We refer to Frederickson's best-first search on heaps as *Frederickson's heap selection algorithm.*

In contrast to the priority-queue based best-first search, Frederickson's heap selection algorithm cannot guarantee any order on the output values. Best-first search on binary heaps in time $\mathcal{O}(k)$ would result in a linear time comparison based sorting algorithm, but sorting of n elements based only on comparisons is in $\Omega(n \log n)$.

For a more detailed overview of k-best enumeration on some graph problems, as well as frameworks that are not based on solution-space partitioning, see the recent survey from Eppstein [41, 42, 43].

Part II.

K Shortest Simple Paths

4. Introduction

This chapter contains definitions that are required for our sidetrack-based k shortest simple path algorithm and a review of existing literature on the subject.

4.1. Definitions

Let $G = (V, E)$ be a directed graph with *source* vertex $s \in V$, *target* vertex $t \in V$, and edge cost function $c\colon E \to \mathbb{R}_{\geq 0}$. The *s-t shortest path problem* (also *single pair shortest path problem* or *SPSP*) is a combinatorial minimization problem. The set of feasible solutions is the set $\mathcal{P}^s_{s,t}$ of all simple s-t paths. By definition of a combinatorial minimization problem, we want to compute the length of a shortest s-t path in G. For the rest of this part, we assume G, V, E, c, s, and t to be fixed, and use the abbreviated form $\mathcal{P} = \mathcal{P}^s_{s,t}$. We denote the number of vertices $|V|$ by n and the number of edges $|E|$ by m.

One might choose to define SPSP via the set of all s-t walks instead. The s-t walk with minimal length is always a simple path. Note, however, that we do not necessarily obtain a combinatorial problem this way. It is possible for a finite graph to have an infinite number of walks. An example for this can be seen in Figure 4.1: The shortest path via v_1 can be extended by an arbitrary number of v_1-v_2-v_3 cycles. In contrast, the edges of a simple path are always a subset of E, which can easily be modeled in combinatorial problems.

The difference between the two notions is even more important when we transition to k-best enumeration. Since there is an infinite number of s-t walks in the example graph in Figure 4.1, k-best enumeration would yield k distinct paths for any k. On the other hand, $\mathcal{P}$ only contains the paths $((s, v_1), (v_1, t))$ and $((s, t))$.

We apply the definition of k-best optimization from Section 3.2 to SPSP to obtain the k shortest simple path problem (kSSP). If we relax kSSP and look for walks instead of simple paths, we obtain the k shortest path problem (kSP). In the latter problem, the second walk can already contain a cycle, which can again be seen in Figure 4.1.

Let $T = (U, F)$ be a shortest path tree to t in G. A *sidetrack* w.r.t. T is an edge e that is not in T, i.e., $e \in (E \setminus F)$. A *sidetrack sequence* w.r.t. T is a sequence of sidetracks $(e_1, \dots, e_r)$ such that for any $i \in [r-1]$, the tail of e_{i+1} is reachable from the head of e_i in T. Every sidetrack sequence represents an s-t walk in G. For $r = 0$, it represents the unique s-t path in T. Otherwise, it represents the walk that starts with the path from s to the tail of e_1, followed by e_1. From the head of e_{i-1}, it proceeds along T until it reaches the tail of e_i, followed by e_i. From the tail of e_r, it proceeds along T until it reaches t. A walk represented by a sidetrack sequence might not be a shortest s-t path,

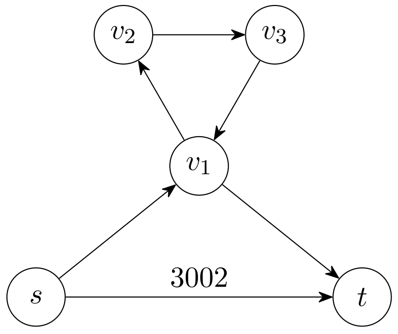


Figure 4.1.: Example input for the s-t shortest path problem. Every unlabeled edge has cost 1. There are two simple s-t paths of lengths 2 and 3002. Further, there is an infinite number of s-t walks, a thousand of which are shorter than the second-shortest simple path.

and its length is never shorter than that of a shortest s-t path.

The *sidetrack cost* of a sidetrack $e = (v, w)$ is the difference between the length of a shortest v-t path on the one side, and the length of the shortest v-t walk that starts with e on the other side. The former is the distance $d_c(v, t)$ from v to t in T. The latter is the cost $c(e)$ of e, plus the distance $d_c(w, t)$ from w to t in T. The length of a walk represented by the sidetrack sequence $(e_1, \dots, e_r)$ is obtained from the length of the walk represented by the sidetrack sequence $(e_1, \dots, e_{r-1})$ by adding the sidetrack cost of e_r. By induction, it is the distance $d_c(s, t)$ from s to t plus the sum of all sidetrack costs in the sequence.

The sidetrack set $D_T(v)$ of a node $v \in V$ is the set of all sidetracks w.r.t. T with tails on the unique v-t path in T. When sidetracks are organized in heaps, we always use sidetrack costs for comparison.

Every algorithm for kSSP or kSP described in this thesis starts with the computation of a shortest path tree T_1 to t. The unique s-t path in T_1 is p_1. The algorithms maintain a set of *candidate s-t paths*, initialized with $\{p_1\}$. Candidates have not yet been determined to be part of the output. However, there is always a candidate that is eligible to be selected as the next-best path. When a candidate path p is selected for output, we *derive* a number of paths from p. The *parent path* of each derived path is p. The parent path of p_1 is undefined. The algorithms terminate when k candidates have been selected, or if no candidate paths are left.

Let $p = (e_1, \dots, e_k)$ be the parent path of a candidate $p' = (e'_1, \dots, e'_l)$, and $i^* = \max\{i \mid e_j = e'_j \text{ for } 1 \leq j < i\}$ be the smallest index where p and p' differ. Then i^* is the *deviation index*, the tail of e'_{i^*} is the *deviation vertex* $\operatorname{dev}(p')$, and e_{i^*} is the *deviation edge* of p'. Removing the deviation edge from p' splits p' into its *prefix path* $\operatorname{pref}(p') = (e'_1, \dots, e'_{i^*-1})$ starting in s, and its *suffix path* $\operatorname{suff}(p') = (e'_{i^*+1}, \dots, e'_l)$ ending in t. An example for these definitions is shown in Figure 4.2. We define the suffix path of p_0 to be p_0 itself. Deviation index, vertex, and edge of p_0, however, are neither defined nor required.

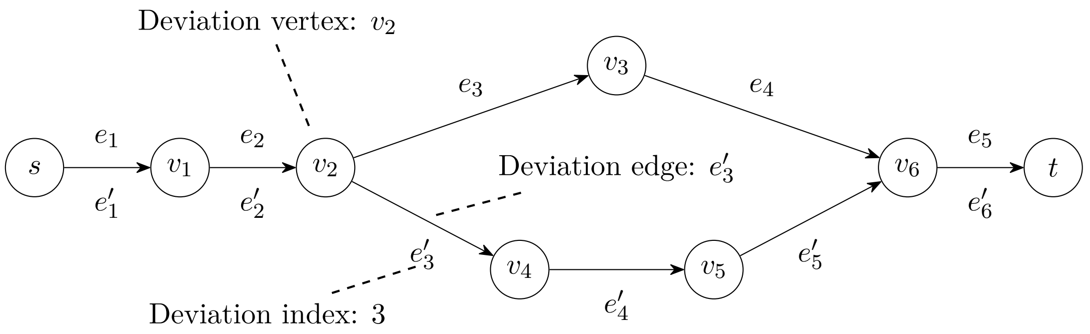


Figure 4.2.: Deviation index, vertex, and edge of path $p' = (e'_1, \dots, e'_{r'})$ derived from path $p = (e_1, \dots, e_r)$.

Two other combinatorial minimization problems relevant for the next sections are the *unrestricted replacement path problem* and the *restricted replacement path problem.* A *unrestricted replacement path* for e_i w.r.t. a path $p = (e_1, \dots, e_r)$, $i \in [r]$, is a shortest s-t path in $G - e_i$. A *restricted replacement path* for e_i w.r.t. p is a shortest s-t path in $G - e_i$ that starts with $(e_1, \dots, e_{i-1})$. Both replacement path problems receive a shortest s-t path $p = (e_1, \dots, e_r)$ in G as input and ask for one respective (unrestricted or restricted) replacement path for each e_i on p.

Each restricted replacement path can, e.g., be found by searching a shortest path from the tail of e_i to t in $G - e_1 - \dots - e_i$, and appending the result to the prefix $(e_1, \dots, e_{i-1})$. This problem is equivalent to searching for a shortest path in $\mathcal{P}(\{e_1, \dots, e_{i-1}\}, \{e_i\})$ for each $i \in [r]$, using the subproblem notation from Section 3.2. Equivalently, the unrestricted replacement path problem is equivalent to finding a shortest path in $\mathcal{P}(\emptyset, \{e_i\})$ for $i \in [r]$.

4.2. Complexity and Related Work

The k shortest simple path problem was introduced in 1963 by Clarke, Krikorian and Rausen [30]. The algorithm by Yen [105], which is based on the restricted replacement path problem, used to have the best asymptotic worst-case running time of $\mathcal{O}(kn(m + n \log n))$ for a long time. A number of other algorithms was derived from Yen's algorithm, all of which share the same worst-case running time. They differ from their original in different practical improvements for the replacement path problem.

Pascoal [85] noticed that the replacement path that deviates from p at node v might be one that uses an edge (v, w) to an unused successor w of v and then follows the path from w to t in T_1. Therefore, they test whether the shortest such path is simple, and fall back to a full shortest path computation if it is not. Although they do not describe in detail how this check is done, it can be done in time $\mathcal{O}(m+n)$ per replacement path instance by partitioning the nodes into blocks, which was first described by Hershberger, Maxel and Suri [60]. Experiments by Pascoal, and by Hershberger, Maxel and Suri suggest that the Yen approach benefits heavily from laziness. Feng [46] uses the shortest path tree T_1 to partition V into red, yellow and green vertices. When looking for a restricted replacement path for e_i, the tails of the edges $e_1, \ldots, e_i$ are red. A vertex v is yellow if the v-t path in T_1 contains at least one red vertex. Every other vertex is green. Instead of looking for a path from each possible deviation vertex to t, they can basically restrict Dijkstra runs to yellow vertices. They demonstrated that yellow vertices often make up a small portion of the graph in practice. Feng does not provide upper bounds on the number of yellow vertices, resulting again in a worst-case running time of $\mathcal{O}(n(m+n\log n))$ for each restricted replacement path instance. Martins, Pascoal and Santos [79] and Martins and Pascoal [78] give other Yen-based accounts on kSSP. Other approaches are proposed by Pollack [89], and by Katoh and Sugimoto [101].

Gotthilf and Lewenstein [55] improved upon Yen's upper bound. They observed that kSSP can be solved by solving $\mathcal{O}(k)$ all pairs shortest path (APSP) instances. Using the APSP algorithm by Pettie [86], they obtain a new upper bound of $\mathcal{O}(kn(m+n\log\log n))$. To the best of our knowledge, this algorithm has not been implemented. Experiments with implementations of Yen-based algorithms suggest that the distances for most pairs of vertices are never required. Pettie's APSP algorithm is not lazy, either, so it remains questionable if the improved theoretical upper bound has any relevance for practical applications.

Another approach by Carlyle and Wood [21] enumerates near-shortest simple paths. For $\varepsilon \in \mathbb{R}_{\geq 0}$, they enumerate all simple s-t paths with a length of at most $(1+\varepsilon)d_c(s,t)$. Using binary search, this algorithm can also be used to solve kSSP in time $\mathcal{O}(kn(m+n\log n)(logn+logC))$, where C is the cost of the most expensive edge. A generalization of kSSP on directed graphs involving an additional capacity edge cost function can be solved in time $\mathcal{O}(knm(n+\log k))$ using an algorithm by Chen [24].

Vassilevska Williams and Williams [103] introduce the concept of *subcubic equivalence*. An algorithm has *truly subcubic* running time if its running time is in $\mathcal{O}(n^{3-\varepsilon})$ for some $\varepsilon > 0$. Two problems are *subcubic equivalent* if either both problems have algorithms with truly subcubic running time, or neither has such an algorithm. They prove subcubic equivalence for a number of problems including APSP and determining the length of a second-shortest path in a directed graph. The latter problem is obviously also subcubic equivalent to kSSP with $k = 2$. No algorithm with truly subcubic running time for the heavily studied APSP is known, nor is any concept that might make finding the $(k+1)$-th shortest simple path in a directed graph any easier than finding the k-th one. Therefore, we do not expect algorithms for kSSP on general directed graphs with running time $\mathcal{O}(kn^{3-\varepsilon})$ any time soon.

The k shortest simple path problem has also been studied on undirected graphs. Katoh, Ibaraki and Mine [67] propose an algorithm that solves kSSP on undirected graphs in time $\mathcal{O}(k(m+n\log n))$. Considering the subcubic reduction to APSP on directed graphs, the undirected kSSP seems to be computationally easier. There are other accounts on kSSP on undirected graphs by Katoh, Ibaraki and Mine [68], Ishii [65] and Christofides, Hadjiconstantinou and Mingozzi [58].

A recent survey of k-best enumeration and applications, including an account on path problems, is due to Eppstein [43].

The fact that the upper time bound for exact kSSP algorithms has not been improved for a long time inspired research on inexact approaches. This line of research so far spans three publications that all use the notion of a *detour* of a path, which is the subpath of a replacement path that diverts from a reference path. It was started by Roditty and Zwick [92] who proposed a Monte Carlo $\mathcal{O}(m\sqrt{n}\log n)$ algorithm for the replacement path problem on directed graphs with small integer weights. They also proposed a framework that solves kSSP by $\mathcal{O}(k)$ computations of second shortest simple paths in appropriate subgraphs, which in turn is solved by their replacement path algorithms. Roditty [91] enhanced this framework to allow for approximate kSSP algorithms when an approximation algorithm for the second shortest path subproblem is used. They also provided a $\frac{3}{2}$-approximation algorithm, leading to a $\frac{3}{2}$-approximation algorithm for kSSP with running time in $\mathcal{O}(k\sqrt{n}(m + n\log n))$. An α-approximation algorithm for kSSP guarantees that the i-th output path is at most α times as long as an actual i-th shortest path. The algorithm for approximate second shortest simple paths distinguishes between two classes of detours, namely short and long ones. This approach was extended by Bernstein [9] from two to $\mathcal{O}(\log n)$ classes of detours, which are handled in increasing order. This way, they were able to obtain an algorithm that gives $(1+\varepsilon)$-approximations in $\mathcal{O}(\varepsilon^{-1}\log^2 n\log(nC/c)(m + n\log n))$ time, where c, C are the minimum and maximum edge cost, respectively, giving the first (approximation) algorithm that breaks the $\mathcal{O}(m\sqrt{n})$ barrier. See Frieder and Roditty [53] for an experimental study of Bernstein's algorithm.

The best known algorithm for the k shortest path problem runs in time $\mathcal{O}(m+n\log n+k\log k)$ and is due to Eppstein [40]. In the initialization phase, the algorithm uses T_1 to build a data structure that contains information about all s-t paths and how they interrelate with each other, in time $\mathcal{O}(m+n)$. The running time for initialization can be reduced from $\mathcal{O}(m+n\log n)$ to $\mathcal{O}(m+n)$ if T_1 can be computed in time $\mathcal{O}(m+n)$ (or is given). In the enumeration phase, a *path graph* is constructed. The path graph is a quaternary min-heap where every path starting in the root correlates to an s-t path in the original graph. We require $\mathcal{O}(k\log k)$ time for the enumeration phase if we want the output paths to be sorted by length in increasing order. If the order is irrelevant, Frederickson's heap selection algorithm [50] can be used to enumerate the paths after the initialization phase in time $\mathcal{O}(k)$.

5. Approaches

Existing algorithm for path enumeration in directed, weighted graphs can basically be partitioned into detour-based algorithms, algorithms based on (restricted or unrestricted) replacement paths, and Eppstein's algorithm. The first approach is exclusively used for approximate shortest simple path enumeration and therefore out of the scope of this thesis. We take a closer look at the other two approaches.

The other two approaches have the set of candidate paths in common. The candidate set is commonly described in terms of a priority queue with efficient access to the element with minimum priority. Every candidate path is pushed to a candidate priority queue. The priority of a path is its length. When the i-th candidate path p_i is extracted from the queue, we derive a set $C(p_i)$ of child paths from it and push each of them to the queue. The operations on the priority queue itself introduce a running time overhead of $\mathcal{O}(kD \log kD)$ if a maximum of D paths is derived from each extracted path.

Priority queues are only the most accessible way to describe the best-first search on the candidate set, though. Instead, we can define a heap of paths as follows. Every set of child paths $C(p)$ of any path p in the heap is itself arranged as a binary min-heap. The parent of the root of this binary heap is p itself. The root of the heap is the shortest path p_1. This way, every path $p' \in C(p)$ has at most two child heaps inside $C(p)$, and the root of the binary heap for $C(p')$. The resulting tree of paths is ternary. Also note that p has to satisfy fewer restrictions than paths derived from p, and can therefore never be longer than any path in $C(p)$. Hence, the tree of paths is also a min-heap. The set $C(p)$ can be arranged in a min-heap in time $\mathcal{O}(|C(p)|)$; this cannot exceed the time to assemble $C(p)$, so the heapification does not introduce an overhead. Frederickson's heap selection algorithm to find the k smallest value on a min-heap of bounded degree only introduces an overhead of $\mathcal{O}(k)$ time.

5.1. Yen-based Algorithms

Algorithms based on replacement paths are exactly those derived from Yen's algorithm. It is the Murty/Lawler framework as described in Section 3.2 adapted to kSSP. Let $p_1 = (e_1, \ldots, e_r)$ be the first path we want to derive replacement paths from. This corresponds to the optimal solution $S = \{e_1, \ldots, e_r\}$ of the combinatorial problem. Let $E \setminus S = \{e'_1, \ldots, e'_m\}$. The Murty/Lawler framework suggests to partition the solution space $\mathcal{P}(\emptyset, \emptyset)$ into sets $\mathcal{P}(I \sqcup \{e_1, \ldots, e_{i-1}\}, \{e_i\})$ for $i \in [r]$, and $\mathcal{P}(S \sqcup \{e'_i\}, \{e'_1, \ldots, e'_{i-1}\})$ for $i \in [m]$. The former sets are exactly the restricted replacement paths: start with a prefix $(e_1, \ldots, e_i)$, but then avoid e_{i+1}. The latter sets consist of solutions that contain a simple s-t path and an additional edge. This contradicts the definition of a simple s-t

path, so the latter sets are all empty. It is therefore sufficient to consider the r restricted replacement path problems.

Now let $p_i = (e_1, \dots, e_r)$, $i > 1$, be the i-th candidate path extracted from the candidate set. Assume that p_i was found to be the shortest path for some subproblem $\mathcal{P}(I, O)$ where I forms a path that starts in s. Further, assume that the tail of any edge in O is either s or the head of some edge in I. The set I then forms a prefix of p_i, and can be written as $I = \{e_1, \dots, e_j\}$ for some $j \in [r]$. Let $E \backslash (S \sqcup O) = \{e'_1, \dots, e'_m\}$. According to the Murty/Lawler framework, we have to derive subproblems $\mathcal{P}(\{e_1, \dots, e_l\}, O \sqcup \{e_{l+1}\})$ for $l \in \{j, \dots, r\}$, and subproblems $\mathcal{P}(\{e_1, \dots, e_r, e'_l\}, O \sqcup \{e'_1, \dots, e'_{l-1}\})$ for $l \in [m]$. The latter subproblems are again infeasible. In the former, we add a new edge e_{l+1} to O, whose tail is the head of e_l. The edges in I also form a path that starts in s again. By induction, our assumptions on the subproblems we encounter always hold. Finally, the feasible subproblems we derive from $\mathcal{P}(I, O)$ can all be solved with a single run of a restricted replacement path algorithm on the graph $G - O$.

Yen-based algorithms come down this:

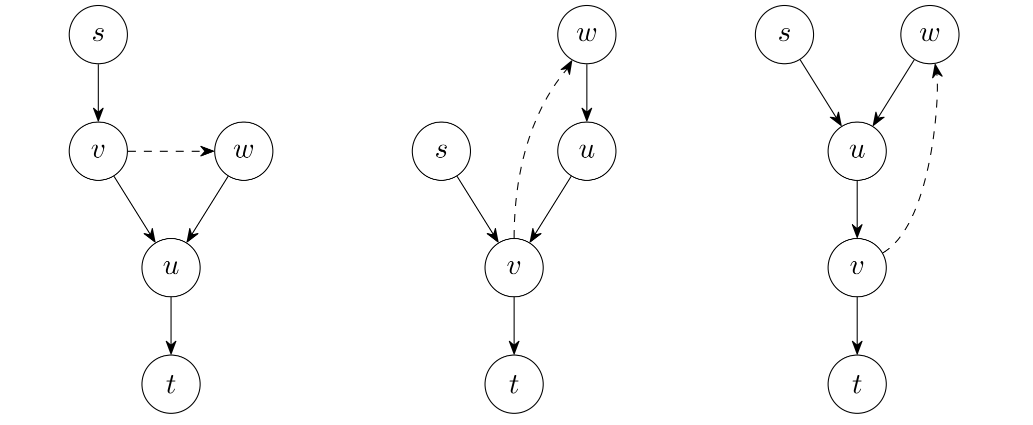
1. Compute an SP tree T_1 to t
2. Push the s-t path in T_1 to Q
3. For $i = 1, \dots, k$: extract p_i from Q and push $C(p_i)$ to Q

In the enumeration phase, we have to solve k restricted replacement path problems to find the sets $C(p_i)$; the computation of child paths of the k-th path extracted from Q can be skipped.

The most trivial approach to solve replacement path problems is already among the fastest known approaches in theory. Let $p = (e_1, \dots, e_r)$ be a shortest s-t path, and let $e_i = (u_i, v_i)$. For each edge e_i on p, compute the shortest u_i-t path p' in $G - u_1 - \cdots - u_{i-1}$. The concatenation of $(e_1, \dots, e_{i-1})$ and p' is a replacement path for e_i. Each shortest path computation requires time $\mathcal{O}(m + n \log n)$ using Dijkstra's algorithm. Path p is simple and can thus comprise at most $n - 1$ edges. Thus, the restricted replacement path problem can be solved in time $\mathcal{O}(n(m + n \log n))$. The resulting kSSP algorithm was proposed by Yen [105].

Now assume that p is the unique s-t path in an SP tree T directed towards t, and let $e = (u_i, v)$ be a sidetrack of p. We obtain an s-t walk by following T from s until we reach the head u_i of e, then e itself, and then T again to reach t from v. This is exactly the walk represented by the sidetrack sequence (e). This walk is a simple path iff the unique v-t path in T does not contain u_i. Note that it is not even a path if the v-t path in T contains u_{i-1}, because it uses the edge e_{i-1} twice. Examples for all three cases can be seen in Figure 5.1. The same observations can be made for sidetracks of other s-t paths.

Let $e = (u_j, v)$ be a sidetrack that represents a simple path. If every other sidetrack has greater sidetrack costs than e, then the path represented by e_j is a restricted replacement path for e_j. Pascoal [85] was the first to observe this, and use it algorithmically. For every vertex u_j on a path p_i we want to compute replacement paths for, we identify



(a) The sidetrack represents a simple path. (b) The sidetrack represents a non-simple path. (c) The sidetrack represents a walk.

Figure 5.1.: Example graphs with their respective SP tree. SP tree edges are solid; sidetracks are dashed. In Figure 5.1a, the walk represented by sidetrack sequence (v, w) visits the vertices s, v, w, u, t in this order, and is therefore a simple path. In Figure 5.1b, the walk visits s, v, w, v, u, t. It does not use any edge twice, so it is a path. The vertex v occurs twice, so it is not simple. In Figure 5.1c, the walk visits s, v, u, w, v, u, t. The edge (v, u) occurs twice, so it is not even a path.

sidetrack e_j with minimum sidetrack costs. If e_j represents a simple path, we output it as the restricted replacement path for (u_j, v_j). Otherwise, we use the Dijkstra algorithm to compute a restricted replacement path, just as in the original Yen algorithm. Experiments by Pascoal suggest practical improvements over Yen.

No non-trivial upper bound is known for the number of Dijkstra runs required by this approach. The trivial bound is the number of edges on a simple s-t path, so the worst-case running time for this restricted replacement path algorithm is still $\mathcal{O}(n(m + n \log n))$, assuming that we can check in $\mathcal{O}(m + n \log n)$ whether e_j represents a simple path. Pascoal's report does not include a detailed description on how this check is done. We can trace the walks represented by each sidetrack, keeping track of the visited vertices and stopping if a vertex occurs twice. However, a path can have $\mathcal{O}(m)$ many sidetracks, so this approach results in a running time of $\mathcal{O}(mn)$. Hershberger and Suri [61] filled in this gap. They proposed to partition V into *blocks*. Vertex v is in block i if the v-t path in T contains u_i, but not u_{i-1}. A sidetrack (u_i, v) represents a simple path iff the block of v is greater than i. The partition can be deduced from the *discovered* (d_v) and *finished* (f_v) numbers after a reverse depth-first search on T starting in t. The block of v is greater than i iff $d_{u_i} < d_v$ and $f_v < f_{u_i}$. The overhead to avoid Dijkstra runs is reduced to $\mathcal{O}(n)$ time for the depth-first search, and $\mathcal{O}(m)$ total time to compare the d and f numbers for each sidetrack. The authors' claim for improvement was again backed by a series of experiments.

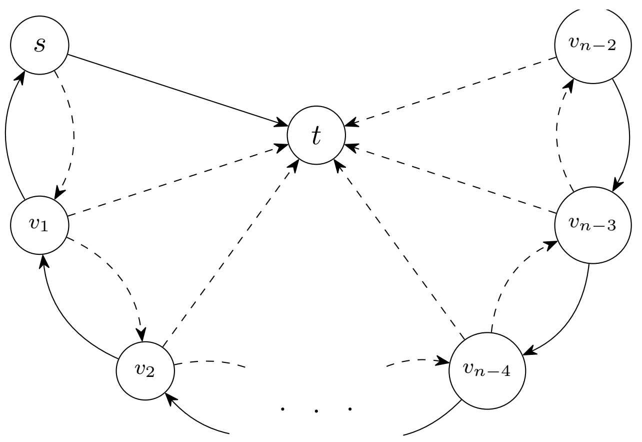


Figure 5.2.: Directed graph with SP tree to t (solid) and its other edges (dashed). We assume unit edge cost. Every sidetrack sequence represents a walk that starts in s and finally reaches t via the edge (s, t).

Feng [46] proposed a vertex-classification based algorithm to speed up replacement path computations in practice. Replacement paths are computed for edges e_i in increasing order of i, starting with e_1. To compute the replacement path for $e_i = (u_i, v_i)$, we partition the vertex set into three sets. Vertices that were used to reach u_i from s are *red*. Vertices that are not red, but whose shortest path to t contains at least one red vertex, are *yellow*. All other vertices are *green*. For every yellow vertex with at least one green successor vertex w, we add a temporary edge (w, t) whose cost is the distance from w to t. We can now run Dijkstra's algorithm on the subgraph of G induced by the yellow edges and t, with u_i as the single source. The Dijkstra algorithm can stop as soon as t is extracted from the priority queue.

While yellow vertices can make up a large part of the graph, the early stopping rule can prune yet again large portions of this subgraph. When iterating from e_i to e_{i+1}, the vertex partition has to be updated. An update procedure also proposed by Feng requires a total time of $\mathcal{O}(m + n)$ summed up over all e_i. Computational studies by Feng suggest that Dijkstra runs are confined to small subgraphs, as their algorithm is yet much faster than other Yen-based algorithms.

Consider sidetrack (u_i, v) such that v is a green node. Since we added an edge (u_i, t) for this case, the Dijkstra run described above would discover a u_i-t path immediately. This path might even be a replacement path for e_i, so we might be inclined to believe that the Feng algorithms detects those easy cases automatically that make the Pascoal approach fast. Since Feng's Dijkstra runs also consider smaller portions of the graph compared to Pascoal's, and their bookkeeping (maintaining the vertex partition) is not slower than that of Hershberger, Maxel and Suri (a depth-first search to identify blocks), this would explain shorter running times directly. However, we have no way of knowing that the u_i-t path is a shortest one as long as there are yellow vertices that are closer to u_i than t.

The practical improvements proposed so far rely on an optimistic assumption. The Hershberger-Maxel-Suri as well as the Feng algorithm both work in linear time per replacement path problem as long as they are not hit by a cheap cycle. The presence of cycles, however, can negatively affect their performances. The graph in Figure 5.2 demonstrates that the optimistic assumption can be violated every time, even in the case of a sparse graph. In the example, s occurs twice on every walk represented by a non-empty sidetrack sequence that does not contain a sidetrack directly to t. Therefore, Yen-based algorithms perform at least one proper shortest path computation per replacement path problem.

5.2. Eppstein's kSP Algorithm

Consider a *relaxed* version of the restricted replacement path problem where replacement paths are not required to be simple. To solve the relaxed version, we first compute a shortest path tree T to t. A replacement path of an edge $e_i = (u_i, v_i)$ of a given path p starts with the prefix of p up to, but not including, e_i. The next edge is a sidetrack e of p with tail u_i with lowest cost. From the head of e, it proceeds along T until it reaches t.

This observation is one of the key ideas of Eppstein's algorithm for the k shortest path problem. The shortest path p_1 is again selected as the one s-t path in T_1, the initial shortest path tree to t. The second shortest path p_2 is the shortest replacement path for some vertex on p_1 w.r.t. the relaxed version above. This can be expressed again in terms of candidate paths. Whenever the next shortest path p is selected from the set of candidate paths, for each relevant sidetrack e of p, we extend p by e and push the result to the set of candidate paths.

We have to make sure, though, that we do not push the same path to the candidate path set twice. Let i be the deviation index of p_2 w.r.t. p_1, and $j \in [i-1]$. The common prefix path of p_1 and p_2 contains the edge $e_j = (u_j, v_j)$, and the s-u_j subpath is the same in both cases, too. Therefore, extending both paths by a sidetrack with tail u_j would result in the same candidate path. This problem also arises with paths that are derived from derived paths. On the other hand, extensions by sidetracks that emanate from the suffix path of a discovered path that proceeds along edges of T are guaranteed to be novel. Hence, when looking for new candidate paths, we only consider these sidetracks.

The paths obtained like this are exactly those represented by valid sidetrack sequences w.r.t. T. In fact, Eppstein used sequences of sidetracks in their description of the algorithm. Instead of the set of candidate paths, we may choose to maintain a set of sidetrack sequences representing those paths. The set would be initialized with the empty sequence. When a sequence of length ℓ is extracted from the set, sequences of length $\ell + 1$ are pushed to it.

Consider a newly discovered candidate path p represented by a sidetrack sequence $(e_1, \ldots, e_r)$ with $e_i = (u_i, v_i)$. Observe that the v_r-t suffix path of p only depends on v_r. The prefix path of p that led to u_r does not affect the suffix in any way. This property enables us to preprocess the input graph after computing its shortest path tree T_1 as follows. For every vertex $v \in V$, $D(v)$ is the set of sidetracks emanating from the unique v-t path in T_1. We organize each $D(v)$ in a binary min-heap w.r.t. sidetrack costs. Since $|D(v)| < m$ for each $v \in V$, all heaps can trivially be constructed in time $\mathcal{O}(mn)$. Using persistent heaps and the fact that $D(w) \subseteq D(v)$ if w is on the unique v-t path in T_1, Eppstein showed that all heaps can actually be constructed in $\mathcal{O}(m+n)$ total time.

With the sidetrack heaps in place, there is no need to push the extension for all sidetracks that emanate from the path represented by $(e_1, \ldots, e_r)$. Instead, we only push $(e_1, \ldots, e_r, e)$, where e is the root of $D(v_r)$ at first, along with a reference to its position in $D(v_r)$. However, this means that $(e_1, \ldots, e_r)$ itself was pushed because $e_r \in D(v_{r-1})$, but for some other sidetracks $e' \in D(v_{r-1})$, $(e_1, \ldots, e_r, e')$ was not pushed. Therefore, we also have to push the child sidetracks of e_r in $D(v_{r-1})$ to the candidate set. The heap property still guarantees that the candidate set always contains a representative of the next shortest path.

Instead of pushing new candidates to a candidate set, we may construct a path heap, where a candidate is a child of the path it was discovered from. This path heap contains subheaps of several (binary) sidetrack heaps, as well as at most one cross heap pointer per path that points to the root of other sidetrack subheaps. The path heap is therefore ternary and its root is the shortest s-t path. We can apply Frederickson's heap selection algorithm to the path heap, which extracts the k shortest paths in $\mathcal{O}(k)$ time. Additionally, we need an initialization phase where T_1 and the sidetrack heaps are computed in time $\mathcal{O}(m + n \log n)$. Thus, the k shortest paths in a weighted, directed graph can be computed in time $\mathcal{O}(m + n \log n + k)$.

6. Algorithm Statement

In this section, we describe a new approach to solve the k shortest simple path problem in weighted, directed graphs. Section 6.1 gives a simplified version of the algorithm. It already contains the most important ideas of our algorithm, but its running time is subpar to existing, Yen-based algorithms. This worst-case running time gap is closed in Section 6.2.

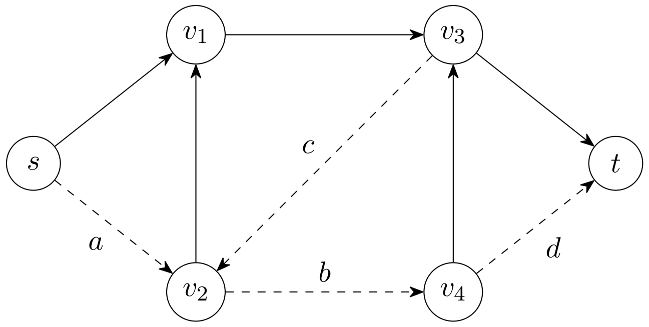
6.1. Basic Algorithm

We generalize Eppstein's representation [40] for paths. Eppstein represents paths as sequences of sidetracks, all w.r.t. the same SP tree. In our representation, every sidetrack in a sidetrack sequence may be associated with a different SP tree. A *rich sidetrack sequence* consists of a sidetrack sequence $(e_1, \ldots, e_r)$ and a mapping T that maps each sidetrack e_i in the sequence to a shortest path tree to t in an induced subgraph of G. The distance from a node v to t in a SP tree $T(e)$ associated with a sidetrack e is denoted by $d_e(v)$. The path represented by a rich sidetrack sequence $(e_1, \ldots, e_r)$ can then be reconstructed as follows. Starting in s, we follow the initial SP tree T_1 until we reach the tail of e_1. After reaching the tail of e_i, we traverse e_i and follow $T(e_i)$ until we reach the tail of e_{i+1}, or, in case $i = r$, until we reach t. Note that Eppstein's representation is the special case where $T(e) = T_1$ for each e in a sidetrack sequence. Also note that both Eppstein's sidetrack sequences and our generalized ones may represent non-simple paths.

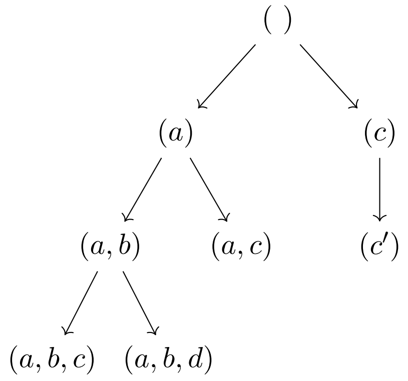
We do not consider sidetracks of a fixed SP tree any more, so we extend the sidetrack set notation $D(v)$. The set $D_T(v)$ for an SP tree T to t in an induced subgraph of $G[V']$ with $v \in V'$ contains every sidetrack w.r.t. T whose tail is reachable from v in T, i.e., whose tail is on the unique v-t path in T.

We refer to the priority queue that represents the candidate set as Q. After the computation of the initial shortest path tree T_1 to t in G, the algorithm proceeds as follows. We push the unique s-t path in T_1, represented by an empty rich sidetrack sequence, to the candidate set. We now process candidates in Q in order of increasing length until k simple paths have been found. A candidate in our algorithm is a rich sidetrack sequence that does not necessarily represent a simple path, so we might have to process more than k candidates.

Let $((e_1, \ldots, e_r), T)$ be a rich sidetrack sequence extracted from the candidate set, and p the path that is represented by this sequence. Although the first path that is pushed to the candidate set is always simple, we will eventually generate non-simple candidates, too. Therefore, we first have to determine whether p is simple in a *pivot step*. This check



(a) Example graph



(b) Sidetrack sequences

Figure 6.1.: Example for the basic algorithm. In Figure 6.1a, solid edges belong to T_1. In Figure 6.1b, every sidetrack except for c' is associated with T_0. $T(c')$ is the SP tree T_1 of the subgraph induced by v_2, v_4 and t, so it only consists of the edges b and d. An arrow from sequence p to sequence p' indicates that p represents the parent path of p'.

can be done by simply walking p and marking every visited node.

We first describe how to handle the case of p being simple. We start by outputting p. Let u be the head of e_r, and $T_r = T(e_r)$. For every sidetrack $e = (v, w) \in D_{T_r}(u)$, we discover a new path p' represented by the sequence $(e_1, \ldots, e_r, e)$. We push p' to the candidate set, represented by the rich sidetrack sequence $((e_1, \ldots, e_r, e), T')$ with $T'(e_i) = T(e_i)$ for $i \in [r]$ and $T(e) = T_r$. By choosing $T(e) = T_r$, we simply reuse the SP tree that is also associated with the last sidetrack in the sequence representing p. We refer to this extension of a rich sidetrack sequence by one sidetrack e that reuses the last SP tree as a *trivial extension* by e. Note that the deviation vertex of p' is v. The length of p' can easily be computed as $c(p) - d_e(v) + c(e) + d_e(w)$, adding the sidetrack cost of e to the length of p. We ignore e if $d_e(w)$ is undefined, i.e., if T_r does not contain a w-t path. Apart from these dead ends, we add one path, the trivial extension, for each sidetrack in $D_{T_r}(u)$ to Q. We also ignore all sidetracks emanating from t.

Example 1. Consider the example in Figure 6.1. The rich sidetrack sequence $((a), T)$ with $T(a) = T_1$ represents a simple path p that visits the vertices s, v_2, v_1, v_3, t in this order. The suffix of this path is its v_2-t subpath, and the sidetracks b, c have tails on this suffix. Therefore, when (a) is extracted from Q, p is output and the rich sidetrack sequences $((a, b), T')$ and $((a, c), T'')$ with $T(a) = T'(a) = T'(b) = T''(a) = T''(c) = T_1$ are pushed to Q.

Now assume we extracted a non-simple path p represented by the rich sidetrack sequence $(e_1, \ldots, e_r)$. We try to extend the concatenation of the prefix path pref(p) of p and e_r to a simple s-t path. Let $e_r = (v, w)$. A valid extension is a v-t path that avoids all vertices of pref(p). We are only interested in shortest extensions. Therefore, we compute a new SP tree T_{new} with distances $d\colon V \to \mathbb{R}_{\geq 0}$, but in $G - \text{pref}(p)$ instead of G to make sure that vertices of the prefix path of p are not used again. If $w \notin T_{\text{new}}$, $\text{pref}(p)$ cannot be extended to a simple s-t path, and we discard p. Otherwise, we push a new rich sidetrack sequence $((e_1, \ldots, e_r), T')$ to the candidate set. We choose $T'(e_i) = T(e_i)$ for $i \in [r-1]$ and $T(e_r) = T_{\text{new}}$. The length of the path represented by this sequence is $c(\text{pref}(p)) + c(e_r) + d_{e_r}(w)$.

Lemma 6.1. *Let $S = ((e_1, \ldots, e_{r-1}), T)$ with $e_{r-1} = (v, w)$ be a rich sidetrack sequence that represents a simple s-t path in G, and let $e_r \in D_{T(e_{r-1})}(v)$ such that the trivial extension of S by e_r is not simple. The rich sidetrack sequence S' resulting from the above repair step of S' results in a simple path.*

Proof. Let p be the path represented by the trivial extension of S by e_r, and p' the path represented by S'. By definition, the prefix paths of p and p' coincide. The path p' is obtained by concatenating the simple prefix path of p, the edge e_r, and the w-t path in T_{new} that, by construction, avoids all nodes of $\text{pref}(p)$. The suffix itself is simple because it is a shortest path in a subgraph of G. Both prefix and suffix of p' are simple and their vertex sets are disjoint by construction, so p' itself is simple. □

Example 2. Consider the example in Figure 6.1 again. The rich sidetrack sequence $((a, c), T)$ with $T(a) = T(c) = T_1$ represents a non-simple path p that visits the vertices s, v_2, v_1, v_3, v_2, v_1, v_3, t in this order. The deviation vertex of p is v_3, its deviation edge is c, and its prefix path is $(a, (v_2, v_1), (v_1, v_3))$. We compute a new SP tree T_{new} in $G - \text{pref}(p)$, which only consists of the edge d. Therefore, T_{new} does not contain a v_2-t path, and p is discarded.

In contrast, consider the rich sidetrack sequence $((c), T)$ with $T(c) = T_1$. It represents almost the same path as the sequence above, but it skips the first visit of v_2. Again, v_3 is the deviation vertex and c the deviation edge. The prefix path comprises the vertices s, v_1 and v_3. After removing them temporarily, a new SP tree T_{new} is computed, consisting only of the edges b and d. The sequence $((c), T')$ with $T'(c) = T_{\text{new}}$ is pushed to Q. This new sequence represents the simple path $((s, v_1), (v_1, v_3), c, b, d)$, where c is the last sidetrack in the extracted sequence, and (b, d) is the unique v_2-t path in T_{new}.

Finally, when $((c), T')$ is extracted, the represented (simple) path is output. The sidetracks emanating from its prefix are (v_2, v_1) and (v_4, v_3). Since $v_1, v_3 \notin T_{\text{new}}$, these sidetracks cannot be extended to simple paths, so no new path is pushed to Q.

Lemma 6.2. *The above algorithm computes the k shortest simple s-t paths of a weighted, directed graph $G = (V, E)$.*

Proof. The algorithm uses the same idea of shortest deviations as existing kSSP algorithms or Eppstein's kSP algorithm. Every non-simple path is repaired immediately upon extraction from the candidate set, which follows from inductively applying Lemma 6.1. We have to show that a non-simple path p is processed before its simple enhancement p', resulting from the suffix repair step in the non-simple case, is actually needed. The set of vertices that are forbidden when the SP tree for p is computed is a proper subset of the vertex set that the SP tree for p' may not use. The suffix of p is therefore not longer than that of p', and p is extracted from the candidate set (and subsequently, p' is pushed) before we need to extract p'. □

This basic form of our algorithm requires too many computations of SP trees:

Lemma 6.3. *The running time of the above algorithm is $\mathcal{O}(km(m + n \log n))$.*

Proof. While processing a non-simple path, at most one new path is pushed to the candidate set, which is always simple according to Lemma 6.1. Thus, the parent of a non-simple path is always simple. We have to process at most k simple paths, each of which requires $\mathcal{O}(n)$ time for traversal, and $\mathcal{O}(m)$ time to process its sidetracks. Every simple path may have $\mathcal{O}(m)$ sidetracks. In the worst case, all of them represent non-simple paths, yielding $\mathcal{O}(km)$ SP tree computations with a total running time of $\mathcal{O}(km(m + n \log n))$. The running time for the non-simple cases clearly dominates.

For every subset of E, there is at most one permutation of this subset that represents a simple s-t path. The maximum number of paths enumerated by the algorithm is therefore $k' := \min\{k, 2^m\}$. We can limit the size of Q efficiently to k' using a double-ended priority queue [94]. We push $\mathcal{O}(k'm)$ paths to Q and extract $\mathcal{O}(k'm)$ paths from it; both operations require $\mathcal{O}(\log k')$ time on interval heaps. The total time spent on processing Q is $\mathcal{O}(k'm \log k') \subset \mathcal{O}(km^2)$.

The pivot step requires $\mathcal{O}(n)$ time for each of the $\mathcal{O}(k'm)$ extracted paths. □

Finally, we turn our attention to the space requirements of the above algorithm.

Lemma 6.4. *The basic sidetrack-based kSSP algorithm requires $\mathcal{O}(kmn)$ space.*

Proof. We need $\mathcal{O}(n)$ space for each SP tree that we compute. Since SP trees are never discarded and we compute one for each non-simple extracted path, the total space for all SP trees is $\mathcal{O}(kmn)$. For each extracted simple path p, we push a path to Q for each edge that has its tail on p. These new paths are represented by an edge and a pointer to some SP tree, and therefore require constant space. We extract up to k simple paths with $\mathcal{O}(m)$ sidetracks each, and therefore require $\mathcal{O}(km)$ space for Q itself. □

Finally, note that the above algorithm does not solve a single replacement path problem. On the one hand, the suffix of the extracted candidate path p might contain a vertex v such that the trivial extension of p by (v, w) for each sidetrack v represents a non-simple path, whereas replacement paths have to be simple. On the other hand, there might be a vertex v such that for multiple sidetracks (v, w), the trivial extension of p by (v, w) represents a simple path. At least one of them (the one with minimum sidetrack cost) could be used as a replacement path. However, we make no attempt to determine which of the sidetracks emanating from v make up the replacement path. If there are both simple and non-simple extensions of p, we do not even know whether a replacement path is among the simple ones. It might be necessary to repair one of the non-simple extensions to obtain a replacement path instead.

6.2. Improved Algorithm

We show how the number of SP tree computations can be reduced from $\mathcal{O}(km)$ to $\mathcal{O}(kn)$ in the worst case. Further, the space requirements are reduced by a factor of n.

So far, we were only able to bound the number of SP tree computations by $\mathcal{O}(m)$ for each extracted simple path. This stems from the fact that there can be $\mathcal{O}(m)$ sidetracks with tails on such a path, each of them requiring a subsequent SP tree computation in the worst case. Consider two sidetrack sequences $(e_1, \dots, e_r, f_1 = (u, v))$, $(e_1, \dots, e_r, f_2 = (u, w))$ that were added when a path p represented by $(e_1, \dots, e_r)$ was processed. Let p_1, p_2 be the paths represented by these sequences, respectively. Assume that both sequences represent non-simple paths, and therefore both require a new SP tree. We assume w.l.o.g. that p_1 is extracted from Q before p_2. When p_1 is extracted from Q, we realize that it contains a cycle. We then have to compute an SP tree T_{new} for the graph $G - \text{pref}(p_1)$. We push $(e_1, \dots, e_r, f_1)$ back to Q, with a slightly adapted SP tree mapping. Accordingly, when p_2 is extracted, we would compute an SP tree to t in $G - \text{pref}(p_2)$. By definition, we have $\text{pref}(p_1) = \text{pref}(p_2)$, and the induced subgraphs we compute SP trees for are the same in both cases. We can therefore skip the SP tree computation when processing p_2, and simply reuse T_{new}.

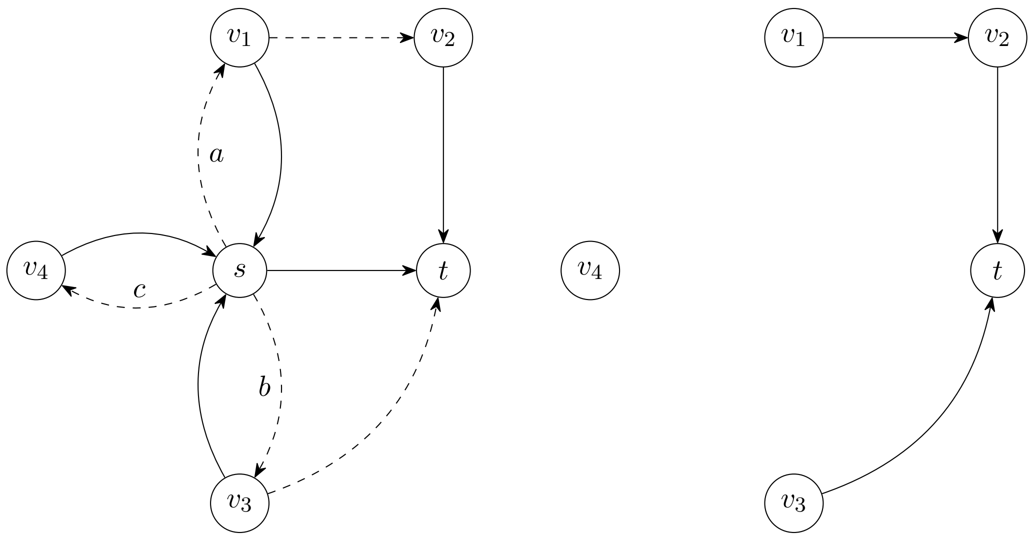
We modify the basic algorithm as follows. Let $S = ((e_1, \dots, e_r), T)$ be a rich sidetrack sequence that represents a non-simple path p, and let $e_r = (v, w)$. When S is extracted from the candidate set and we realize that p is not simple, we first check if an SP tree for v has already been computed. If one has been computed, we simply reuse it. Otherwise, we compute T_{new}, store it along with v in $T(e_r)$ for future reference, and use T_{new}. In the situation above, we would have to compute T_{new} when processing p_1, and simply resort to T_{new} again when processing p_2.

Example 3. Consider the example kSSP input in Figure 6.2. For an SP tree T to t in a subgraph of the input graph G, we denote by T' the SP tree mapping that maps any sidetrack of a rich sidetrack sequence to T. The candidate set is initialized with the empty rich sidetrack sequence, which represents the path $((s, t))$. When it is extracted, we find that the represented path is simple, so we push a trivial extension for each sidetrack a, b, and c. Assume that $((a), T_1')$ is extracted from Q first. We realize that the represented path $(a, (v_1, s), (s, t))$ is not simple. We remove the prefix path of that path from G temporarily to obtain $G - s$, depicted in Figure 6.2b, and obtain the SP tree T_2. The sequence $((a), T_2')$ is pushed to Q and represents the simple path $(a, (v_1, v_2), (v_2, t))$. We also associate T_2 with s for reference.

When $((b), T_1')$ is extracted later on, we find that it is not simple, either, and that an SP tree for s has already been computed. We push $((b), T_2')$ to Q directly, which represents the simple path $(b, (v_3, t))$.

Eventually, $((c), T_1')$ is extracted from Q. Again, we realize that it represents a non-simple path and that we can reuse T_2. There is no v_4-t path in T_2, so we discard the sequence.

We obtain the following result.



(a) Example graph (b) Induced subgraph

Figure 6.2.: Example input for kSSP and an induced subgraph as the sidetrack-based algorithm would encounter it. SP tree edges are solid, sidetracks are dashed. All sequences $((e), T_1')$, $e \in \{a, b, c\}$, represent non-simple path and therefore trigger the repair step. In all cases, a new SP tree in the depicted induced subgraph is required, but the latter two computations can be skipped.

Lemma 6.5. *Excluding the time spent on Q, the algorithm proposed in Section 6.1 in conjunction with SP tree reuse requires $\mathcal{O}(kn(m + n \log n))$ time to process non-simple paths.*

Proof. There are still $\mathcal{O}(km)$ many sequences in Q that represent non-simple paths, but only $\mathcal{O}(kn)$ of them trigger an SP tree computation. Let p be a non-simple path extracted from Q. The initial pivot step still requires time $\mathcal{O}(n)$. We store in Q along with each path a pointer to its parent path, as well as a pointer to the SP tree for $G - p'$ for every prefix path p'. We can then check if an SP tree for some prefix path has already been computed, and access it if it has been computed, both in constant time. □

The total running time of $\mathcal{O}(km^2)$ spent on Q is no longer dominated by SP tree computations. Instead of using a priority queue for the candidate paths, we organize all computed paths in a min-heap in the following way. The shortest path is the root of the min-heap. Whenever a path p' is computed while a path p is processed, we insert p' into the min-heap as a child of p. Figure 6.1b shows an example of such a min-heap. We use Frederickson's heap selection algorithm [50] to extract the km smallest elements from this heap. The heap described above has maximum degree m, again yielding a running time of $\mathcal{O}(km^2)$. Recall that $C(p)$ is the set of paths found during the processing of p. Instead of inserting every $p' \in C(p)$ as a heap child of p, we heapify $C(p)$ to obtain the heap H_p, using the lengths of the paths for keys again. The root of H_p is then inserted into the global min-heap as a child of p. Note that the parent path of every path in H_p is not its heap parent in H_p, but still p itself. Every simple path p in the min-heap now has at most two heap successors with the same parent path as p, and at most one heap successor whose parent is p itself. Every non-simple path has at most one simple path as its heap predecessor. The maximum degree of the global min-heap is therefore bounded by three and Frederickson's heap selection can be applied in time $\mathcal{O}(km)$.

Corollary 6.6. *Let $G = (V, E)$ be a directed graph, $s, t \in V$ two vertices of G, $c\colon E \to \mathcal{R}_{\geq 0}$ an edge weight function, and $k \in \mathbb{N}$. The algorithm proposed in Section 6.1 in conjunction with SP tree reuse and Frederickson's heap selection algorithm computes the k shortest simple s-t paths in G in time $\mathcal{O}(kn(m + n \log n))$.*

The first improvement above reduces the space required by the basic algorithm from $\mathcal{O}(kmn)$ to $\mathcal{O}(kn^2)$. We are not able to reduce the number of SP tree computations to $o(kn)$. The lower bound for kSSP by Vassilevska Williams and Williams [103] even suggests that the number of SP tree computations cannot be further reduced in our approach. However, it is not necessary to permanently store all these SP trees at the same time. Only up to k of them can contain a simple path that eventually gets extracted from Q. We propose to store the computed SP trees in a max priority queue $\mathcal{S}$. The priority of an SP tree T in $\mathcal{S}$ is $\max\{c(p') + c(e) + d_T(w) \mid e = (v, e) \in E\}$, where $p' = (e_1, \ldots, e_r, f)$ is the path T was computed for, and $f = (u, v)$ for some $u \in V$. Whenever $\mathcal{S}$ contains more than k SP trees, at least one of them will not contribute to the k shortest simple paths. By definition of the priority of an SP tree in $\mathcal{S}$, this is always the SP tree with the highest priority. It can be extracted from $\mathcal{S}$ in $\mathcal{O}(\log k)$ time and therefore does not have an impact on the worst-case running time. The space that was used to store the extracted tree can later be used to store new SP trees. The number of SP trees stored at any point in time never exceeds $k + 1$, requiring $\mathcal{O}(kn)$ space. The min heap of candidate paths requires $\mathcal{O}(m)$ entries per simple paths, and therefore $\mathcal{O}(km)$ total space.

Corollary 6.7. *The sidetrack-based algorithm with SP tree reuse, Frederickson's heap selection algorithm and a priority queue for SP trees solves kSSP in time $\mathcal{O}(kn(m + n \log n))$ and space $\mathcal{O}(km)$.*

7. Practical Improvements

In this section, we propose more improvements for the sidetrack-based kSSP algorithm described in Chapter 6. In contrast to the improvements in Section 6.2, the modifications suggested here do not affect the worst-case time or space requirements. Instead, they aim to improve the algorithm's practical performance.

7.1. Faster Reachability Checks

Our first practical improvement lets us to speed up the repeated checks for path simplicity by a factor of n. The overall running time is not altered because the running time for the slower simplicity checks is dominated by the SP tree computations.

Let $p = (e_1, \ldots, e_r)$ be a simple path extracted from Q in the sidetrack-based algorithm, and let $e_i = (v_{i-1}, v_i)$ with $s = v_0$ and $t = v_r$. When p is processed, we push the set $C(p)$ of paths to Q, with $|C(p)| \in \mathcal{O}(m)$. The basic algorithm tests for each $p' \in C(p)$ if p' is simple in time $\mathcal{O}(n)$, leading to a total time of $\mathcal{O}(kmn)$ for these tests. Let T be the SP tree associated with the last sidetrack of p. By removing from T all edges of suff(p), T is decomposed into a set of trees T_i such that T_i is rooted in the i-th vertex of suff(p). The *block* i is the vertex set of T_i.

Lemma 7.1. *A trivial extension of the sidetrack sequence representing p by a sidetrack $e = (v, w)$, with v, w being in blocks i, j, respectively, is simple iff $i < j$.*

Proof. Observe that $v = v_i$, and that the last vertex in block k on the unique u-t path in $T(e)$ for a vertex u in block k is v_k If $i \geq j$, we follow p until we reach v_i, traverse e and follow T to reach v_i again: The w-t path in $T(e)$ exits block j via v_j, and the v_j-t path in $T(e)$ is a suffix path of p and contains v_i. Otherwise, the first vertex on p we hit after deviating from p via e is v_j. Since $i < j$, the v_j-t subpath of p does not contain v_i, so p' is simple. □

Vertices can be partitioned into blocks in time $\mathcal{O}(n)$ by computing DFS numbers for T. We can then collect all sidetracks deviating from p and check for each of them if their heads belong to a smaller block than their tails in $\mathcal{O}(m)$ total time. We store this information along with the corresponding sidetrack sequences in Q. The pivot step is replaced by a constant time lookup. All tests for simplicity then require time $\mathcal{O}(k(m+n))$ instead of $\mathcal{O}(kmn)$.

Example 4. Consider the example graph in Figure 7.1. We partition the vertex set into four blocks. Vertices s, v_1 form block 1; v_2, v_4 make up block 2; v_3, v_5 are block 3; and t constitutes the singleton block 4. Inter-block sidetracks like a always represent

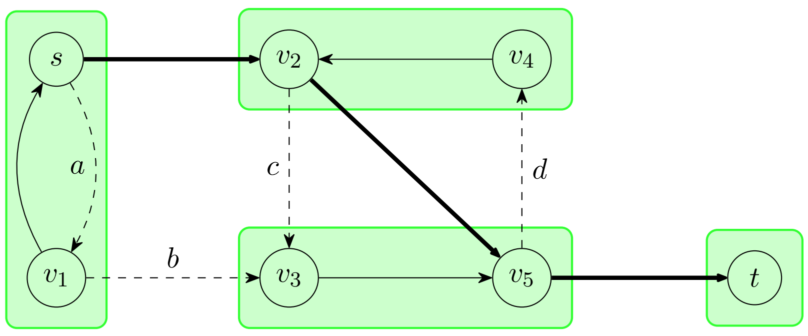


Figure 7.1.: Example block partition of the vertex set. Solid edges denote edges of the SP tree T_1. The thick s-t path is (the suffix path of) the shortest s-t path. When its edges are removed, the SP tree is partitioned into four green blocks.

non-simple paths. Sidetrack c leads from block 2 to block 3, so it represents a simple path. Sidetrack d is the other way around, leading to a smaller block, so it represents a non-simple path.

Now consider the sidetrack sequence $((a), T)$ with $T(a) = T_1$. Its prefix path is the s-t path denoted in thick again, so we obtain the same block partition. Its suffix path emits the sidetrack b. Since it leads from block 1 to block 3, the sequence $((a, b), T')$ with $T(a) = T(b) = T_1$ is simple. Note that extracting $((a), T)$ from Q would have triggered an SP computation, so the sidetrack-based algorithm would never consider the sequence $((a, b), T')$.

7.2. Online Edge Pruning

The following modification helps us narrow down Dijkstra runs to a smaller portion of the input graph. This is most beneficial for instances that are problematic anyway since they require many SP tree computations.

The number k of paths to enumerate is known to the algorithm. This allows for some pruning as soon as k candidates have been found that actually represent simple paths. We propose to maintain two separate candidate sets, one for simple paths and one for non-simple paths. Recall that using the improvement in Section 7.1, candidate simplicity is checked before a path is pushed to the candidate set, so it is straightforward to push simple and non-simple candidates to different sets. When the sidetrack-based kSSP algorithm needs to draw the next candidate, it uses the cheapest candidate in either set, whichever is cheaper. If the cheapest simple candidate is just as cheap as the cheapest non-simple one, we use the simple one. This pivoting approach can save some SP tree computations, since the k-th shortest path might already be among the cheapest ones in the set of simple candidates.

For the set of simple candidates, we use a double-headed priority queue as described by Sahni [94]. This enables us to efficiently both access and remove not only the candidate with minimum priority, but also the one with maximum priority, without increasing the asymptotic running time. Let $\kappa\colon \mathbb{N} \to \mathbb{N}$ be a function such that $\kappa(i)$ simple candidates have been extracted *before* the i-th candidate is extracted. Further, let $\sigma\colon \mathbb{N} \to \mathbb{N}$ be a function such that there are $\sigma(i)$ many candidates in the set of simple candidates *before* the i-th candidate is extracted. Initially, we have $\kappa(i) = 0$ and $\sigma(i) = 1$. The sum $\kappa(i) + \sigma(i)$ is the number of candidates that have been in the set of candidates so far when the i-th candidate is extracted. Thus, as soon as we have $\kappa(i) + \sigma(i) \geq k$, the most expensive simple candidate is an upper bound on the length of the k-th shortest path.

In contrast, we can lower bound the length of s-t paths that use an edge $e = (v, w)$. During initialization, we compute the SP tree T_1 to t with distances $d_t\colon V \to \mathbb{R}_{\geq 0}$ anyway. As soon as an upper bound is found, we compute an additional SP tree T from s with distances $d_s\colon V \to \mathbb{R}_{\geq 0}$. Any path that contains edge e has at least length $d(e) := d_s(v) + d_t(w)$. We propose to sort E in decreasing order of d once. Whenever the upper bound for the length of the k-th shortest path is updated, we remove the most expensive edges w.r.t. d from G until the most expensive remaining one is at most as expensive as the new upper bound.

Using a double-headed priority queue for simple candidates also enables us to limit the size $\ell(i)$ of the simple candidate set to $k - \kappa(i)$ before the i-th candidate is extracted. Whenever the size of the set exceeds $\ell(i)$, we extract the most expensive simple candidate and discard it. With this candidate pruning, it seems natural to use a double-headed priority for non-simple candidates, too. Although we cannot limit the size of the set of non-simple candidates effectively (e.g., to $k - \kappa(i) - \ell(i)$), we can pop and discard the most expensive non-simple candidates until the most expensive one is cheaper than the current upper bound on the cost of the k-th shortest path.

8. Computational Study

To demonstrate the effectiveness of our algorithm, we conducted a series of experiments. We give an overview of the implementations in Section 8.1. In Section 8.2, we summarize the results of our computational study. The graph classes we use match those considered in Feng's experiments [46] and are described along with the results. They include road graphs that are especially relevant in practice.

8.1. Implementations

Feng [46] showed recently that their algorithm is the most efficient one in practice. We therefore compare our sidetrack-based algorithm to Feng's node classification algorithm. For reference, we also include results for the most promising third contender, an algorithm proposed by Sedeño-Noda [96].

Sedeño-Noda kindly provided us with the implementation *KC* (short for the Portuguese term for k shortest path, *K caminho menor*) of their algorithm. Our implementation *NC* of Feng's node-classification algorithm and our implementations *SB* of the sidetrack-based algorithm are both written in C++11. Graphs are given in both forward and reverse star representation, allowing efficient computations of SP trees from s and to t. Express edges as proposed by Feng for their algorithm are not used in NC. We also implemented Feng's algorithm *with* express edges. This implementation, however, was always slower than NC, so we dropped it. Note that Feng does not specify whether express edges were used in their experiments. Neither do they provide evidence that express edges have any advantage to the vanilla version of their algorithm, so the name might be misleading.

Shortest paths (for NC) and SP trees (for SB) are computed using a common implementation of Dijkstra's algorithm. Larkin, Sen and Tarjan demonstrated [75] that pairing heaps perform well when used in the Dijkstra algorithm. Therefore, we use a pairing heap to maintain tentative labels. Our implementation of Dijkstra's algorithm stops as soon as the label of the target node is made permanent if only a single pair shortest path is needed, which is essential for NC. SP trees are computed lazily as follows. A tree is initialized without any edges, and the source node is pushed to a priority queue that is permanently associated to the tree. Whenever a part of the tree (i.e. the distance or predecessor of some node) is queried that has not yet been computed, we simulate the Dijkstra algorithm using the associated priority queue until the queried part is settled.

The queue of candidate paths Q is implemented as an interval heap, a form of double-headed priority queues, which allows us to limit its size efficiently to the number of simple paths that have yet to be output. For SB, we use separate priority queues Q_s and Q_n for simple and non-simple paths, respectively. Whenever a path has to be extracted, SB extracts from Q_n iff the shortest path in Q_n is cheaper than the shortest path in Q_s. For NC, every candidate is a simple path. Hence, splitting Q in two is not necessary for NC, but we still limit its size as described for Q_s in Section 7.2.

We reason that our implementation of Feng's algorithm is at least as efficient as the one used in Feng's experiments. Feng reports that they used a desktop computer with a 3.4GHz Intel Core i7 processor. At the time of their paper submission, there were only four processors matching that description, with the processor we use for our experiments being one of them. Instead of Feng's 12 GB RAM, our machine had 16 GB RAM, but none of the algorithms used close to either of those bounds. In both setups, some Linux distribution was used (Ubuntu GNU/Linux for Feng, Gentoo GNU/Linux for us), so e.g. the schedulers used are comparable, too. Thus, we assume that running times are quite comparable between the two setups. On our machine, KC was consistently slower than what is reported [46] for KC on Feng's computer. For example, we required 10.4 seconds on NY and 15.94 seconds on BAY (described in Section 8.2.1) on average using KC. Feng reported 8.81 and 11.23 seconds for the same instances, respectively. In contrast, our implementation NC of Feng's algorithm consistently gives lower running times than those reported by Feng.

All improvements proposed in Section 6.2 were used in the implementation SB of our sidetrack-based algorithm with a single exception: Frederickson's heap selection algorithm was used neither for NC nor for SB. This results in an additional running time of $\mathcal{O}(km \log k)$ for SB, and $\mathcal{O}(kn \log k)$ for NC. Further, the practical improvements from Chapter 7 were implemented. The overhead introduced by the faster simplicity checks is negligible, so we always include it. The impact of the overhead introduced for the edge and candidate pruning is less clear. It includes splitting one pairing heap into two interval heaps, bearing the risk of breaking cache lines since we operate on one half of an interval heap most of the time. It also requires one computation of an SP tree from s, as well as sorting the edge set once. We found that SB profits from the pruning overall, but we also state running time statistics for a variant *SN* of SB where pruning is omitted. SN is the implementation we used [73] in experiments before.

It is possible to transfer the edge pruning approach to Feng's algorithm. We did not do so for two reasons. First, the whole point of the node-classification effort of Feng's algorithm is to contain Dijkstra runs to small portions of the graph already. Second, the number of simple paths derived from a simple candidate extracted from Q tends to be much slower for Yen-based algorithms. For a path with ℓ edges, we derive at most ℓ new simple paths. Each of them requires a separate Dijkstra run. In contrast, the sidetrack-based approach can yield one simple path for every sidetrack on that path. Every vertex on the path for which a replacement path can be found in Yen-based algorithms, there is at least one sidetrack, particularly the one used by the replacement path. Further, the set of simple candidates is determined in $\mathcal{O}(m + n)$ total time in SB, instead of $\mathcal{O}(m + n \log n)$ per candidate.

The experiments ran on an Intel Core i7-3770 @ 3.40GHz with 16GB of RAM on a

Label	Area	Vertices	Edges
DC	Washington, D.C.	9559	29 818
DE	Detroit	49 109	121 024
VT	Vermont	97 975	215 116
NY	New York	264 346	733 846
BAY	San Francisco Bay Area	321 270	800 172
COL	Colorado	435 666	1 057 066
FLA	Florida	1 070 376	2 712 798

Table 8.1.: Number of vertices and edges of the TIGER graphs used in kSSP experiments. We refer to DC, DE and VT as the *small* road graphs and to NY, BAY, COL and FLA as the *large* road graphs.

GNU/Gentoo Linux with kernel version `4.12.12` and Turbo Boost turned off. Source code was compiled using the GNU C++ compiler `g++-5.2.0` and `-O3` optimization.

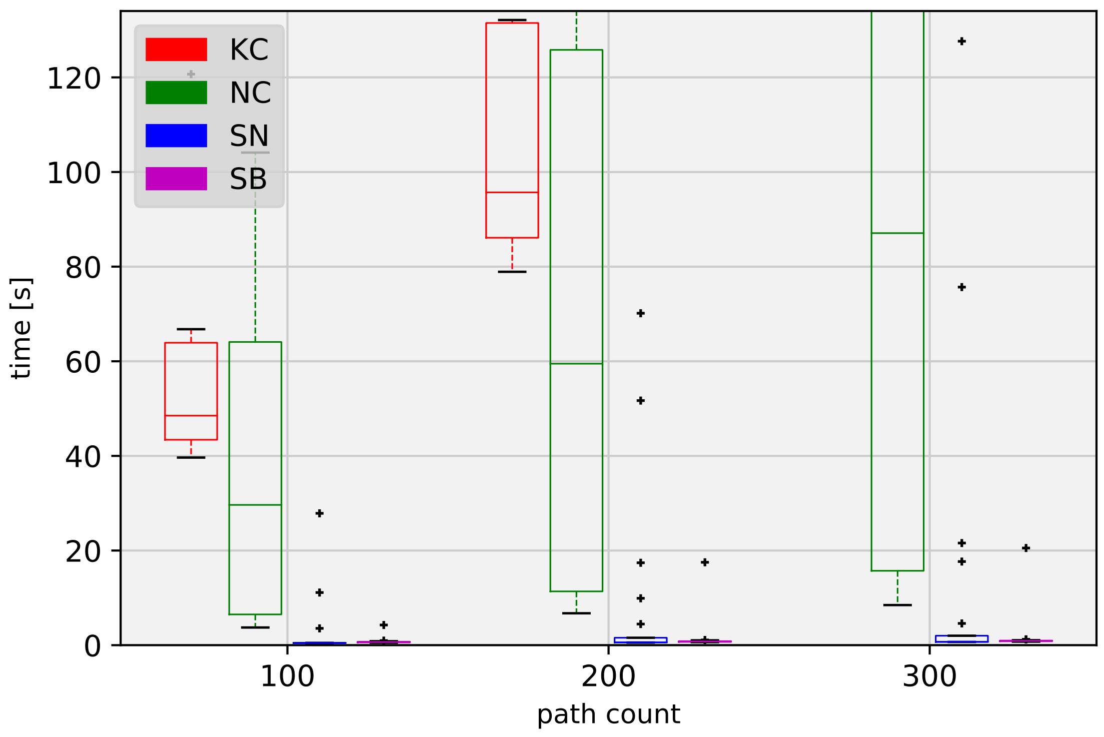
8.2. Experiments

We give statistics of wall clock running times of KC, NC, SN and SB on the set of graph classes that are also used in experiments by Feng [46]. These classes provide an excellent range of properties we would like to control in a meaningful computational study. Routing is a classic motivation for many shortest path problems, so using real-world TIGER road graphs in Section 8.2.1 is most obvious. On the down side, we are unable to modify parameters of real-world graphs. Erdős-Rényi random graphs in Section 8.2.2 and synthetic grid graphs in Section 8.2.3 fill this gap. The former let us control the sizes and densities of the generated graphs. The latter has fixed density, but we can choose the shape of the graph. Synthetic network graphs in Section 8.2.4 try to mimic real-world data, but we may generate a larger number of graphs, each with a chosen size.

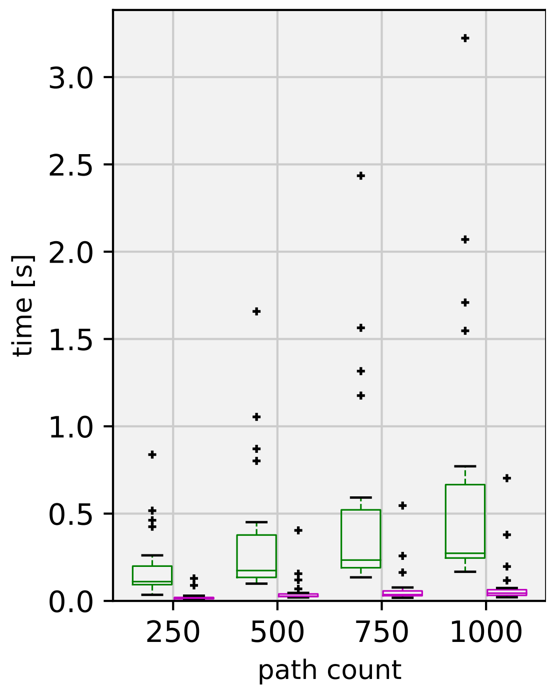
Axes are scaled linearly unless noted otherwise. We choose y-axis limits in boxplot figures such that all boxes of at least two algorithms are in full view. Boxplots show the 25%-quantile $Q_{.25}$ as the upper boundary of the box, the 50%-quantile $Q_{.5}$ (median) as a vertical line inside the box, the 75% quantile $Q_{.75}$ as the lower boundary of the box, as well as the range of points in the 1.5 interquartile range as whiskers and their outliers. If one algorithm is clearly not competitive on a graph class, we only include its boxes in one boxplot for this graph class to improve readability of the remaining figures.

8.2.1. Results on Road Graphs

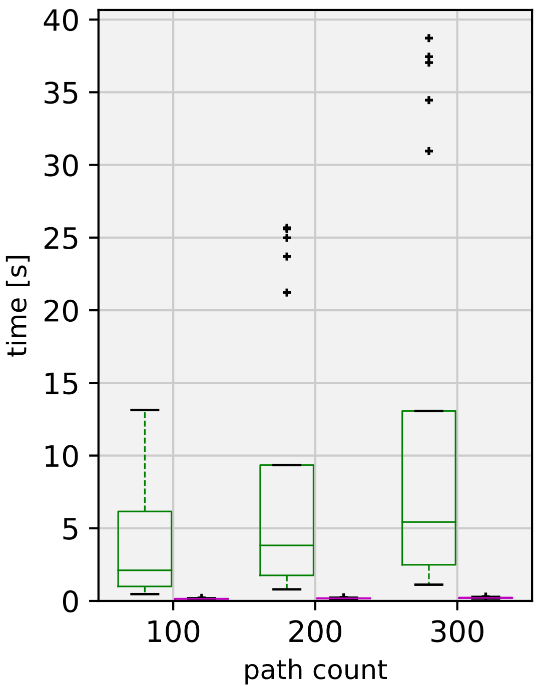
A classic application for shortest path problems are navigation systems that route users in a vehicle through a network of streets. A good resource of real-world street graphs are available as the so-called TIGER graphs, modeling street networks in various areas in the USA. These road graphs were also used in the Ninth DIMACS implementation



(a) Florida



(b) Washington DC



(c) New York

Figure 8.1.: Boxplots of running times in seconds of NC and SB on DC and NY, and of all four kSSP implementations on FLA, depending on the number k of enumerated paths.

		$k = 100$			$k = 200$			$k = 300$		
Area		Med.	$Q_{.9}$	Mean	Med.	$Q_{.9}$	Mean	Med.	$Q_{.9}$	Mean
NY	KC	9.72	11.57	10.40	19.54	23.48	21.04	29.76	35.55	32.07
	NC	2.09	12.38	4.27	3.80	24.30	8.06	5.43	36.18	11.77
	SN	0.18	1.57	0.60	0.26	3.92	1.30	0.43	5.99	1.82
	SB	**0.15**	**0.18**	**0.15**	**0.18**	**0.22**	**0.18**	**0.21**	**0.26**	**0.22**
BAY	KC	12.72	25.90	15.94	25.16	53.00	32.46	38.17	83.36	49.78
	NC	5.32	17.88	8.18	9.49	34.57	15.90	14.14	51.11	23.28
	SN	0.30	4.28	2.51	0.55	9.67	5.88	0.71	14.17	9.19
	SB	**0.19**	**0.27**	**0.22**	**0.23**	**0.67**	**0.38**	**0.27**	**1.12**	**0.59**
COL	KC	17.13	32.99	23.30	34.56	71.66	51.12	56.77	117.12	81.11
	NC	6.83	27.71	11.56	12.04	49.23	21.51	16.73	65.92	30.83
	SN	**0.17**	11.80	2.82	**0.22**	17.43	5.86	**0.29**	26.64	9.13
	SB	0.23	**0.46**	**0.47**	0.29	**0.57**	**0.72**	0.34	**0.94**	**1.15**
FLA	KC	48.51	99.06	57.90	95.69	215.48	120.67	-	-	-
	NC	29.86	70.10	37.21	58.07	132.73	69.96	85.70	193.30	103.03
	SN	**0.47**	4.31	2.52	**0.58**	20.84	8.12	**0.71**	26.99	12.91
	SB	0.67	**0.85**	**0.85**	0.79	**1.03**	**1.63**	0.91	**1.21**	**1.90**

Table 8.2.: Running times for large TIGER road graphs. For each input class, we report median, 90% quantile $Q_{.9}$ and mean running time in seconds.

challenge, which was dedicated to shortest path problems. The graphs are still available today on the DIMACS website [37].

Table 8.1 summarizes the sizes of the road graphs we used in our experiments. For each pair, we enumerated $k \in \{250, 500, 750, 1000\}$ simple paths on the small road graphs, and $k \in \{100, 200, 300\}$ simple paths on the large areas. An *input class* is defined by an area and a number k, resulting in 12 input classes for small and large areas each. For each area, we drew 20 s-t pairs at random, uniformly and independently. For a quick overview of our results on road graphs, see Figure 8.1. This figure also demonstrates that practical running times actually scale linearly with the number of enumerated paths, as the theoretical analysis suggests.

Running times for the large areas are summarized in Table 8.2, along with the median number of polls during runs of the Dijkstra algorithm. With respect to the median running times, KC is clearly dominated by NC on any input class, which in turn is dominated by SN. The relation between the sidetrack-based implementations SN and SB is less clear when comparing the median times, since pruning seems to slightly hurt the performance on larger the instances COL and FLA. However, when taking the 90% quantile into account, the running times of SN show a stronger dispersion than those of SB. The ratio $\frac{Q_{.9}}{Q_{.5}}$ ($Q_{.5}$ being the median running time) is below 2 for most input classes with SB (with a peak of 4.15 for $k = 300$ on BAY), but mostly above 10 with SN. SB also clearly dominates SN when it comes to mean running times.

We compare median running times of NC and SN. SN achieves a minimum speedup of

		$m = 2n$		$m = 4n$		$m = 10n$		$m = 30n$		$m = 50n$	
n		Med.	$Q_{.9}$	Med.	$Q_{.9}$	Med.	$Q_{.9}$	Med.	$Q_{.9}$	Med.	$Q_{.9}$
2000	KC	0.67	0.69	0.85	0.88	1.29	1.32	3.58	4.04	9.77	14.11
	NC	0.99	2.51	0.48	1.25	0.39	1.25	0.47	1.50	2.15	3.84
	SN	0.09	**0.13**	0.08	0.10	0.10	0.15	0.17	0.20	0.23	0.31
	SB	**0.07**	0.14	**0.06**	**0.08**	**0.08**	**0.10**	**0.11**	**0.13**	**0.15**	**0.18**
4000	KC	1.39	1.46	1.79	1.87	2.96	3.06	15.64	16.03	32.88	33.17
	NC	1.01	2.88	0.84	2.71	0.82	1.97	1.33	5.06	2.07	5.49
	SN	0.11	**0.12**	0.09	0.13	0.12	0.19	0.19	0.22	0.26	0.36
	SB	**0.09**	**0.12**	**0.07**	**0.10**	**0.08**	**0.12**	**0.13**	**0.14**	**0.17**	**0.21**
6000	KC	2.19	2.25	2.86	2.90	5.50	5.78	30.13	30.61	53.33	54.51
	NC	3.28	6.44	0.53	1.67	0.71	3.36	2.05	7.92	2.22	9.07
	SN	0.13	0.20	0.11	0.13	0.13	0.21	0.22	0.32	0.30	0.45
	SB	**0.10**	**0.14**	**0.09**	**0.10**	**0.10**	**0.12**	**0.15**	**0.19**	**0.20**	**0.24**
8000	KC	3.04	3.06	4.08	4.14	12.37	12.60	43.36	45.31	73.11	75.00
	NC	1.79	7.84	0.68	2.66	1.92	4.60	3.49	9.67	2.55	9.66
	SN	0.12	0.28	0.10	0.14	0.16	0.21	0.24	0.34	0.32	0.38
	SB	**0.09**	**0.14**	**0.09**	**0.10**	**0.11**	**0.13**	**0.16**	**0.19**	**0.22**	**0.24**
10 000	KC	3.96	3.98	5.48	5.53	15.71	15.84	55.95	57.76	92.81	94.79
	NC	1.86	11.91	1.17	5.44	2.61	9.02	6.31	13.56	9.68	26.23
	SN	0.13	0.18	0.14	0.24	0.17	0.21	0.25	0.30	0.36	0.48
	SB	**0.11**	**0.13**	**0.10**	**0.14**	**0.12**	**0.13**	**0.18**	**0.20**	**0.26**	**0.28**

Table 8.3.: Medians and 90% quantiles of running times in seconds of all four kSSP implementations for $k = 2000$ simple paths on random graphs with varying vertex count and linear density.

around 11 on the smallest of the large graphs, NY, for $k = 100$. This speedup consistently gets bigger as we increase the graph size. It also seems to get bigger as we enumerate more paths, with two minor exceptions: It drops from almost 15 for 200 paths on NY to below 13 for 300 on NY; and there is a minor decrease when enumerating 200 instead of 100 paths on BAY. The latter tendency is more obvious when enabling edge pruning. The more paths we enumerate, the bigger is the speedup of SB in contrast to NC. This comes to little surprise, because the overhead of computing the single SP tree from the source node and sorting the edges once becomes less important. After all, there number of Dijkstra runs that can benefit from the reduction in graph size increases along with k. Although SB is faster than SN most of the time, the maximum speedup of 120 against NC is achieved by SN for 300 paths on FLA.

8.2.2. Results on Random Graphs

We consider random graphs generated by the `sprand` generator provided on the website of the Ninth DIMACS Implementation Challenge [37]. The generator draws at

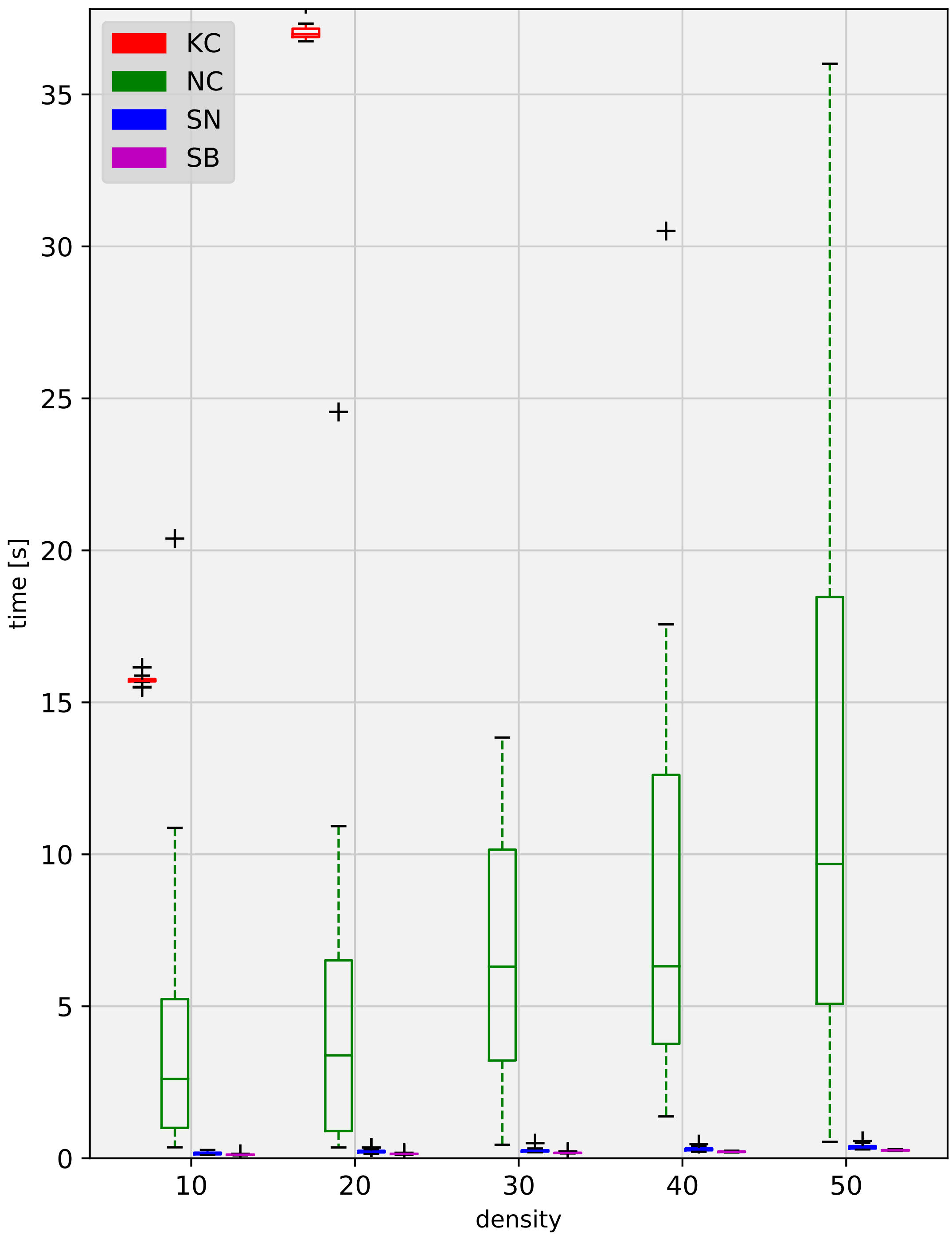


Figure 8.2.: Boxplots for running times in seconds for $k = 2000$ simple paths on random graphs with $n = 10\,000$ vertices, depending on the linear densities $\frac{m}{n}$.

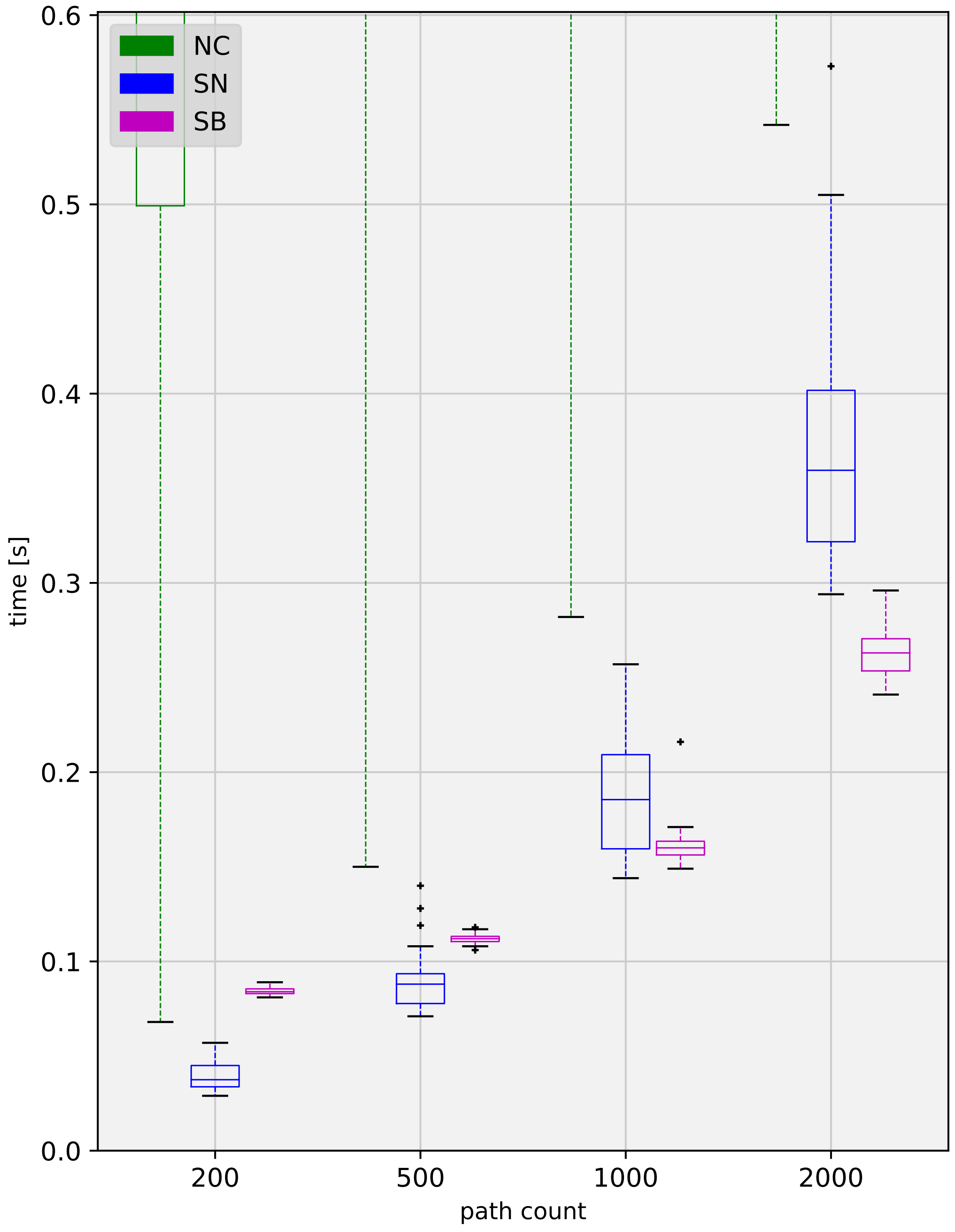


Figure 8.3.: Boxplots for running times in seconds of NC, SN and SB on random graphs with $n = 10\,000$ vertices and linear densities $\frac{m}{n} = 50$, depending on the number k of enumerated paths.

		$m = 4n$		$m = 10n$		$m = 30n$		$m = 50n$	
n		Runs	Polls	Runs	Polls	Runs	Polls	Runs	Polls
	NC	16 272	1.08	14 533	0.60	13 939	0.43	14 510	2.09
2000	SN	46	0.09	65	0.13	38	0.08	44	0.09
	SB	47	**0.06**	66	**0.07**	39	**0.04**	45	**0.05**
	NC	17 292	1.93	14 581	1.45	15 805	1.20	15 605	1.39
4000	SN	25	0.10	20	0.08	21	0.08	29	0.11
	SB	26	**0.05**	21	**0.04**	22	**0.04**	30	**0.05**
	NC	17 499	0.86	16 652	0.70	16 544	1.52	16 444	1.13
6000	SN	23	0.14	19	0.11	24	0.14	21	0.13
	SB	24	**0.06**	20	**0.05**	25	**0.06**	22	**0.06**
	NC	18 300	1.01	17 316	2.40	17 127	2.72	17 034	1.09
8000	SN	16	0.12	17	0.13	17	0.14	17	0.14
	SB	17	**0.06**	18	**0.06**	18	**0.06**	18	**0.06**
	NC	19 074	1.94	17 824	3.53	18 125	5.07	18 187	6.11
10 000	SN	16	0.15	15	0.15	10	0.10	14	0.13
	SB	17	**0.06**	16	**0.06**	11	**0.05**	15	**0.06**

Table 8.4.: Medians and 90% quantiles of the numbers of Dijkstra runs and the numbers of polls during those runs (in millions) of NC, SN and SB for $k = 2000$ simple paths on random graphs with varying vertex count and linear density.

random a fixed amount of edges, possibly resulting in a non-simple graph. Here, an *input class* is defined by the number $n \in \{2000, 4000, 6000, 8000, 10\,000\}$ and a *linear density* $m/n \in \{2, 3, 4, 7, 10, 20, 30, 40, 50\}$. Edge weights are drawn uniformly and independently from $[10\,000]$. For each input class, we enumerate $k \in \{200, 500, 1000, 2000\}$ simple paths.

In Table 8.3, we summarize the median and 90% quantile $Q_{.9}$ of execution times for some densities and $k = 2000$. These results are also shown in Figure 8.3 as boxplots for $n = 10\,000$, $k = 2000$ and varying density, and in Figure 8.2 for $n = 10\,000$, $\frac{m}{n} = 50$ and varying number of enumerated paths. A more detailed view on the two fastest implementations SN and SB in these figures is given in Figure 8.4.

Our results confirm Feng's claim that NC is usually faster than KC on random graphs. NC seems to struggle with low densities of $\frac{m}{n} \leq 2$, and KC is often faster for those. This is especially true for $Q_{.9}$ running times, especially when considering $Q_{.9}$ running times. On the other hand, KC and SB display a more consistent growth. SB is always fastest, with speedup factors ranging from 8 to 20 (outlier: speedup 32.8 for $n = 6000$ and $m = 2n$) for lower densities, and speedups around factor 13 for $m = 50n$ (outlier: speedup 37 for $n = 10\,000$) when compared to NC. SB is faster than SN, but not significantly faster, and the relative gap does not seem to change as we modify graph parameters. Shortest paths in Erdős-Rényi random graphs are often long in comparison to the diameter of the graph, so we cannot benefit from pruning as much as for road graphs.

We now consider the running time dispersion. For SB, 90% of the instances finish within 160% of the corresponding median running times (the fastest 50%) for most input classes. The dispersion of SN is similarly low. For KC, this ratio even stays below 106% for all but two input classes ($n = 2000$, $\frac{m}{n} \geq 30$). We cannot find any correlation between n, m and k on the one side, and the dispersion of running times on the other side, for KC and SB. In contrast, NC regularly requires more than thrice the median running time to answer 90% of the queries. Running times are therefore much harder to predict on random graphs when using NC instead of SB or KC.

Table 8.4 shows the median number of Dijkstra calls. The numbers are relatively stable across the various densities, but the Dijkstra counts for the SB algorithms is orders of magnitudes smaller than the count for the NC algorithm. Note, however, that Dijkstra runs for NC may be confined to a much smaller area than Dijkstra runs for SB. For this reason, we also provide the number of polls, i.e., the total number of vertices that were extracted from Dijkstra's priority queue, for comparability. The ratio of the number of polls of NC and SB ranges from 4.6 to 50, and suggests that saving SP tree computations is much more beneficial than reducing the number of nodes visited to answer single-pair shortest path queries.

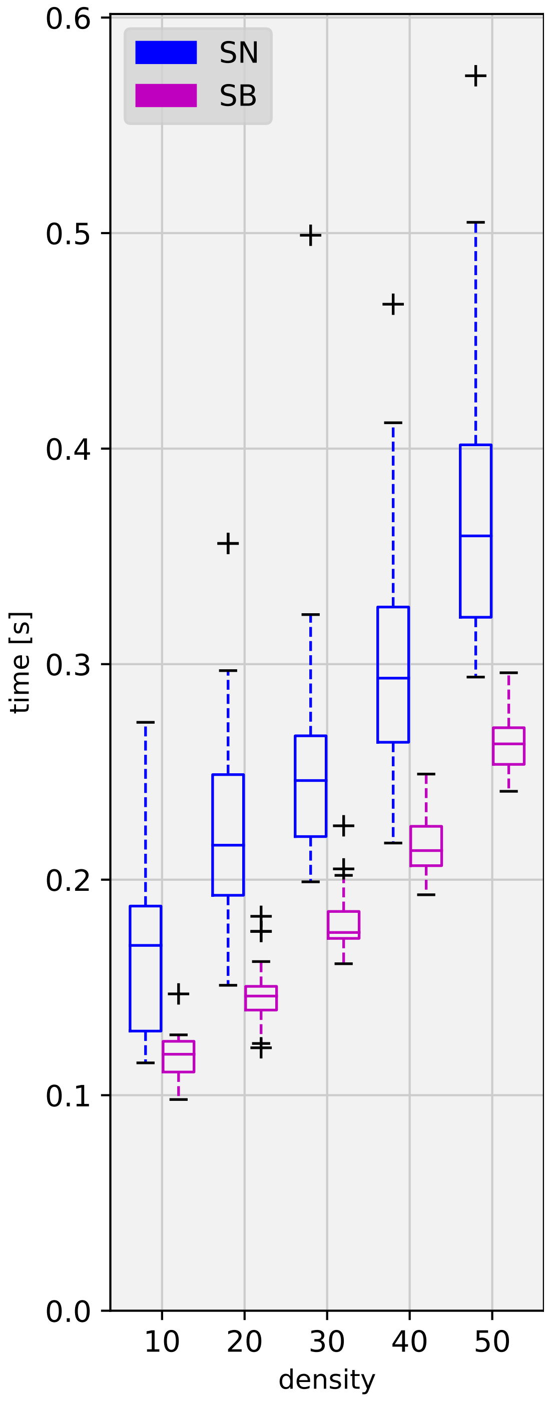
Finally, the number of SP tree computations for SB actually declines as n grows. Recall that, in the worst case, we have to compute one SP tree for each node of each output simple path. Table 8.4 shows results for $k = 2000$ and $n \geq 2000$. Nevertheless, the median number of SP tree computations does not exceed 65. Most simple paths therefore correspond to those well-behaved cases where paths represented by sidetracks in already computed SP tree areas are themselves simple.

8.2.3. Results on Grid Graphs

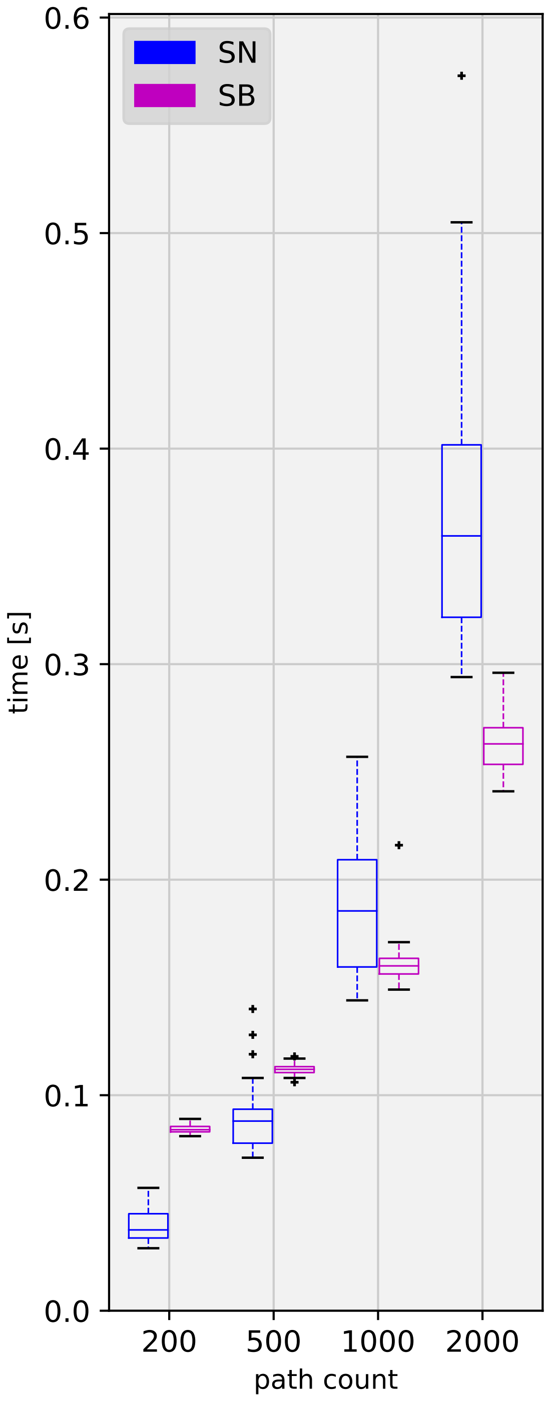
We repeated Feng's experiments on grid graphs generated by the `spgrid` generator provided on the DIMACS website [37]. The grids have side lengths $l, w \in \{50, 100, 200, 400\}$ with $l \leq w$, resulting in 10 different grids. For each grid, we generated 20 weight functions by selecting uniformly from $\{1, \ldots, 10\,000\}$ for each edge, and then enumerated $k \in \{100, 200, 400, 800, 1600, 3200\}$ paths. Source and target vertex lie in opposing corners of the grid.

The results of our experiments on grid graphs are summarized in Figure 8.5. KC is again the slowest algorithm, but NC is not significantly slower than SN any more. In some cases, NC is even faster than SN. Although NC and SB differ on some classes, there does not seem to be any correlation to the shape or size of the grid. For example, NC is slightly faster on 50×100 grids, but slower on 200×400 grids although these two configurations share the same shape, characterized by an aspect ratio of 2. On the other hand, NC loses its advantage over SN when the grid grows from 50×50 to 50×200 (left column of subfigures), but SB loses its advantage over NC as the grid grows from 100×400 to 400×400 (right column). In summary, the comparison between the two remains inconclusive. However, SB is again much faster than SN, and therefore dominates state-of-the-art algorithms on random grids, too.

The shortest path between two adjacent vertices u and v in a grid only consists of the edge (u, v) itself. Consequently, edges of the shortest path tree T_1 used by Feng to

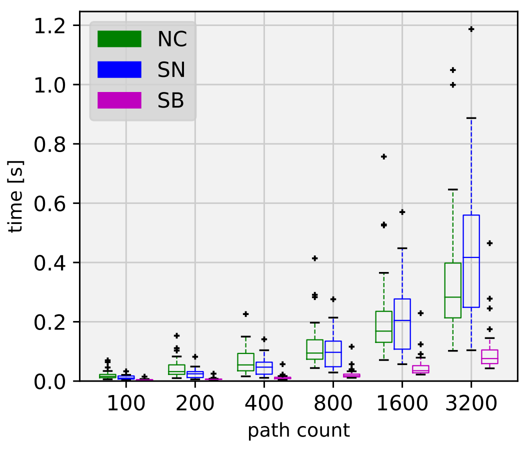


(a) Varying density $\frac{m}{n}$; $k = 2000$ paths

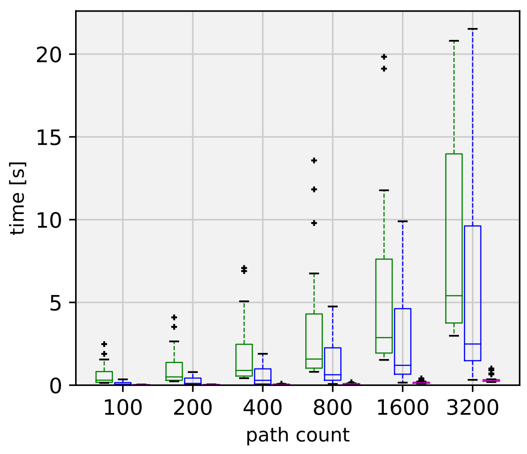


(b) Varying path count k, density $\frac{m}{n} = 50$

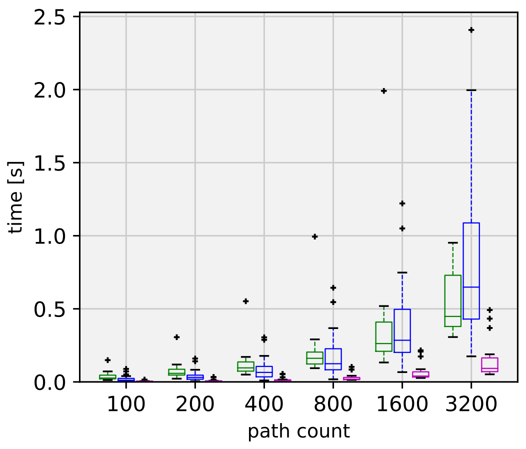
Figure 8.4.: Boxplots for running times in seconds of SN and SB on random graphs with $n = 10\,000$ vertices.



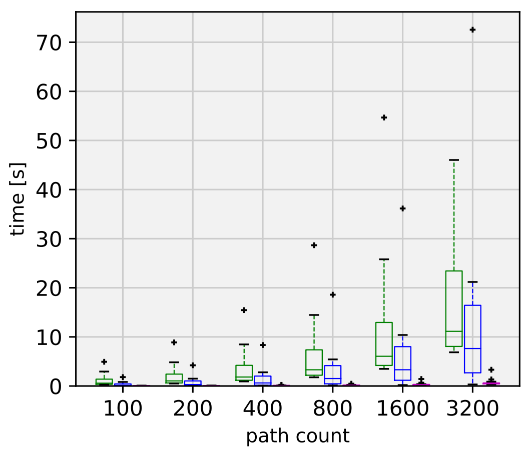
(a) 50 × 50



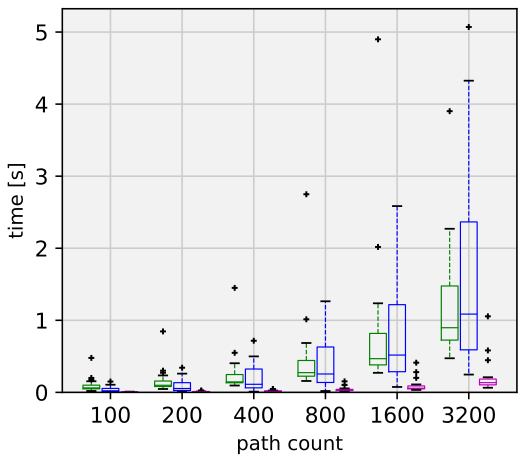
(b) 100 × 400



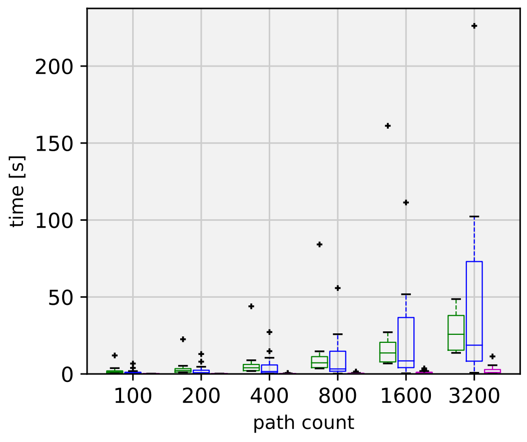
(c) 50 × 100



(d) 200 × 400



(e) 50 × 200



(f) 400 × 400

Figure 8.5.: Boxplots of running times in seconds for grid graphs of various sizes. The logarithmic-scale x-axis corresponds to different numbers k of enumerated paths.

classify vertices are mostly directed toward the target vertex side of the grid. Hence, the probability of a vertex being green is high, and the area to which Dijkstra runs are confined are often empty.

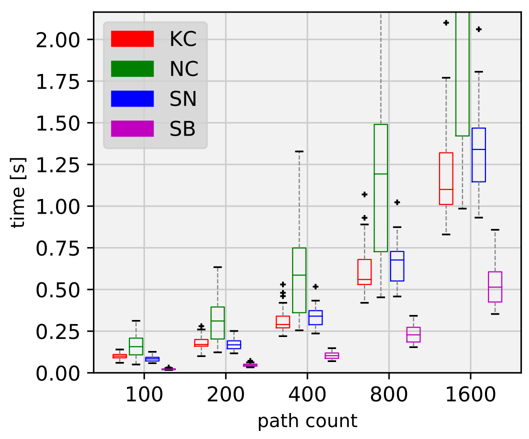
The grid structure is also well suited for the pruning technique that is used in SB, but not in SN, since sidetracks also represent simple paths with high probability. Thus, we can start pruning early, and so more Dijkstra runs benefit from it.

8.2.4. Results on Network Graphs

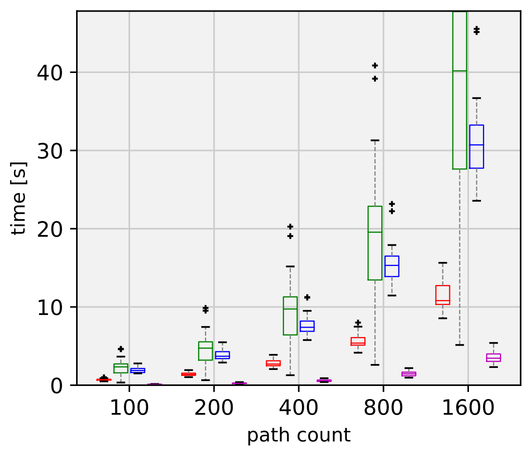
To generate network topology graphs, we use graph generator called BRITE [19] and developed at the Boston University in Boston, Massachusetts. We use the AS Barabaśi-Albert model, which models a topology with symmetric edge relation that was gradually grown as follows. We add vertices $v_1, \ldots, v_n$ in this order. We use v_1 as source vertex and v_n as target vertex. When vertex v_j enters the graph, the probability of an edge (v_i, v_j) (and thus also (v_j, v_i)) being added to the graph is proportional to $d(v_j)/(\sum_{k\in[j-1]} v_k)$. The generator allows to specify the number of vertices a newly added vertex connects to, and scales the probability for adding an edge accordingly. Like Feng, we use graph sizes of $n \in \{1024, 2048, 4096, 8192, 16384\}$, and densities $\frac{m}{n} \in \{4, 12, 20, 28, 36\}$. As for random graphs, an *input class* is a combination of n and $\frac{m}{n}$. Because of some corner cases, the actual density is slightly below the specified one. For example, the first four vertices cannot connect to four other vertices that were added earlier. Edge costs (*bandwidths*) are integers and u.i.d. in $\{10, \ldots, 1024\}$. Vertex placement in the plane and the size of the plane should not have any effect when using the AS Barabaśi-Albert model. For reference, we used random vertex placement, a main plane size of 1000 and an inner plane size of 100. We enumerate up to $k = 1600$ simple paths.

Figure 8.6 shows running times for all four implementations on some input classes. In contrast to the other graph classes, KC is quite competitive on topology graphs. On $(n, \frac{m}{n}) \in \{(4016, 28), (16384, 20), (16384, 28)\}$, it is on par with NC and SN. For very low density, it is even faster than these two. However, this does not mean that topology graphs especially well suited for KC. Instead, the optimistic approach of NC and SN seems to fail, thus spoiling their advantage. Considering running times for other graph classes and similar sizes, NC and SN are much slower on topology graphs. For example, NC enumerates 2000 simple paths on random graphs with 10 000 vertices and density 30 in a median running time of 6.31 seconds. On topology graphs with 16 384 vertices and density approximately 28, NC requires a median running time of more than 50 seconds to enumerate 1600 simple paths. The left column of subfigures suggests that NC gets faster in relation to KC and SN as the graph size increases. Accordingly, the right column suggests the same for rising densities.

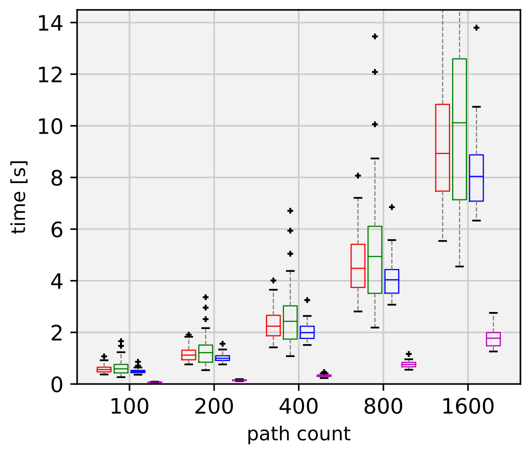
Finally, the sidetrack-based algorithm with pruning, SB, is again the fastest one. For the biggest and densest graphs in Figure 8.6, the median running time speedup when compared to the respective second-fastest implementation is 8 or higher. More importantly, the speedup increases both along with graph size and density.



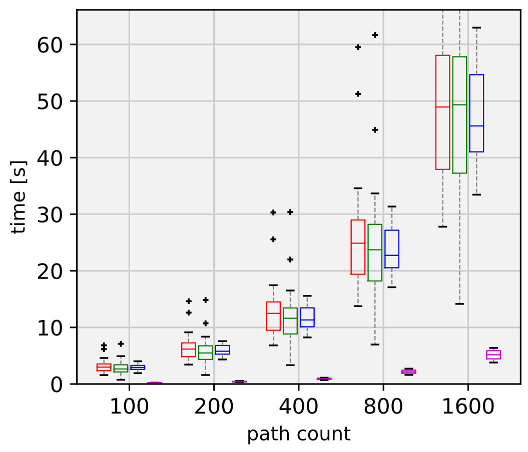
(a) $n = 1024$, $m \approx 28n$



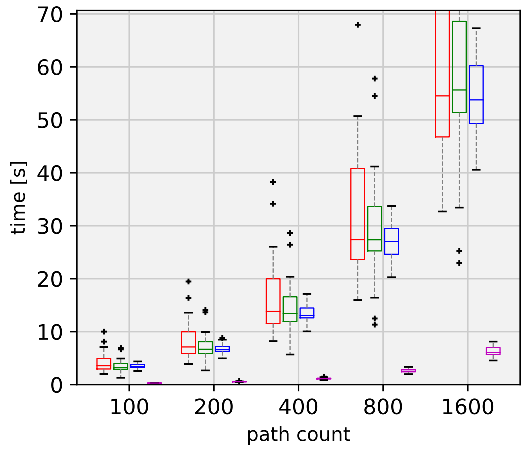
(b) $n = 16\,384$, $m \approx 4n$



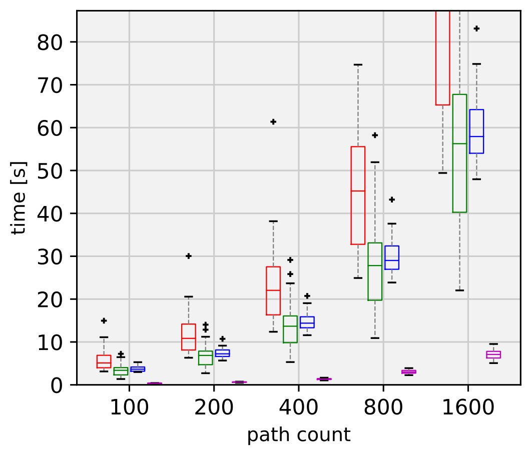
(c) $n = 4096$, $m \approx 28n$



(d) $n = 16\,384$, $m \approx 20n$



(e) $n = 16\,384$, $m \approx 28n$



(f) $n = 16\,384$, $m \approx 36n$

Figure 8.6.: Boxplots of running times in seconds for BRITE network graphs. The logarithmic-scale x-axis corresponds to different numbers k of enumerated paths.

8.2.5. Summary

In our computational study, we compare two state-of-the-art algorithms for kSSP, namely an implementation KC of Sedeño-Noda's algorithm [96] and an implementation NC of Feng's algorithm [46] NC, with the algorithm we propose in this thesis. Besides an implementation SB of our algorithm with all practical improvements, we also consider an implementation SN which omits online edge pruning. We use the same benchmark graph classes that were already used in Feng's computational study, including synthetic Erdős-Rényi random graphs, real-world road graphs from the United States, and others. We can confirm Feng's findings that KC performs much worse than NC in practice, leaving NC as the currently best Yen-based kSSP algorithm practically. However, our algorithm achieves significant speedups when compared to Feng's NC on all graph classes, especially the fully optimized implementation SB. SB was only slower than SN for very small path counts, but the performance hit of each extra path seems to be much harder for SN than for SB.

Part III.

K-Best Enumeration on Tree-decomposable Graphs

9. Introduction

In this part, we propose an algorithm for computing the k best solution values of combinatorial graph problems definable in monadic second-order logic in $\mathcal{O}(|V| + k \log |V|)$ for graphs $G = (V, E)$. Section 9.1 establishes definitions only required in this part. Previous work on the combination of monadic second-order logic and graphs is presented in Section 9.2. Chapter 10 arranges the basic ideas of the k-best algorithm around tree decompositions of undirected graphs. A generalization for directed hypergraphs is given in Chapter 11. We give proof sketches for the undirected case first, and complete proofs for the general case later. The former is a special case of the latter, so correctness follows for the undirected case, too.

9.1. Counting Monadic Second-order Logic

A wide range of graph properties can be expressed in various logic systems. Consider the set of connected undirected graphs. An undirected graph $G = (V, E)$ is connected iff for any pair of vertices s, t, s is in the connected component of t. A formula expressing connectedness might therefore look like this:

$$\forall\, s, t : s \in \mathit{ConnectedComponent}(t).$$

This formula assumes that the universe of discourse, the set the individuals s and t are drawn from, is the set of vertices of G. The above characterization requires access to the function ConnectedComponent that maps vertex t to the set of vertices in its connected component. On the other hand, we are able to describe whether a set of vertices form a connected component in another formula:

$$\mathit{isConnectedComponent}(W) := [\forall\, v, w : (s \in W \wedge t \notin W) \Rightarrow (s, t) \notin \mathbf{E})]\,.$$

In contrast to the first formula, where all variables are universally quantified, this formula has one free variable W in addition. The symbol $\mathbf{E}$ is used as a binary relation symbol in this formula. Assuming standard semantics of second-order logic, $\mathbf{E}$ is a non-logical symbol: we need some context to interpret terms like $(s, t) \in \mathbf{E}$. Assigning context to a formula is called *interpretation*, and is defined more formally below. For now, we simply say that G satisfies the formula $(u, v) \in \mathbf{E}$ if $\{u, v\} \in E$.

Using this formula, we can express connectedness without using the ConnectedComponent function:

$$\mathit{isConnected} := [\forall\, s, t : \forall\, W : (\mathit{isConnectedComponent}(W) \wedge t \in W) \Rightarrow s \in W]\,.$$

In summary, we need some context that specifies the universe of discourse, i.e., the set of vertices of a graph, and a symmetric relation on these vertices, i.e., the edge relation, to be able to describe connected undirected graphs.

Another example is the existence of a Eulerian cycle in G. An Eulerian cycle is a cycle that contains every edge. According to our definition of a cycle, no edge occurs twice, so an Eulerian cycle contains every edge exactly once. It is well known that there is an Eulerian cycle in G iff G is connected and every vertex of G has even degree. The formula

$$\mathit{isConnected} \wedge \forall\, v : \forall\, W : \mathit{isNeighborhood}(v, W) \Rightarrow |W| \text{ is even}$$

therefore characterizes graphs that contain at least one Eulerian cycle, where the formula isNeighborhood(v) expresses that W is the set of all vertices adjacent to v. This neighborhood can easily be expressed in first-order logic, i.e., using a formula that only quantifies individuals (elements of the universe) instead of relations: Every $u \in W$ is adjacent to v, and every $u \notin W$ is not adjacent to v.

To check whether $|W|$ is even, we need another non-logical symbol $\mathbf{Card}_{m,p}$. The atomic formula $\mathbf{Card}_{m,p}(W)$ expresses that the cardinality of W is congruent to p modulo m. For example, sets W of even size are characterized by $\mathbf{Card}_{2,0}(\mathbf{W})$.

We may also consider edges as objects on their own, enabling quantification over edges or relations over edges. An additional symbol **isIncident** expresses the incidence relation between a vertex and an edge. For example, a complete graph can be characterized by $\forall\, s, t : (s, t) \in \mathbf{E}$, where we quantify over V, or by

$$\forall\, s, t : (s \in \mathbf{V} \wedge t \in \mathbf{V}) \Rightarrow (\exists\, e : \mathbf{isIncident}(s, e) \wedge \mathbf{isIncident}(t, e)),$$

quantifying s, t and e over $V \cup E$. Courcelle and Engelfriet [34] demonstrate that there are properties of graphs that can be expressed when edges are discrete objects. They use the graph property of having a perfect matching, which is easily characterized by

$$\exists\, M \subseteq \mathbf{E} : \forall\, s \in \mathbf{V} : \exists\, e \in M : \mathbf{isIncident}(s, e) \wedge \forall\, e' \in M : (\mathbf{isIncident}(s, e') \Rightarrow e = e').$$

Proving that graphs with perfect matchings cannot be expressed without discrete edge objects is more involved; we refer the reader to Courcelle and Engelfriet [34].

For a more formal approach, we use definitions from Libkin [77]. A *signature* σ is a set of constant symbols, set in lower-case bold letters, and a set of relation symbols, set in upper-case bold letters or bold words, each associated with an arity. Let $\{x, y, z, \dots, X, Y, Z, \dots\}$ be a set of variables. A *term* over σ is either a variable or a constant in σ. A *formula in second-order logic* over σ is either an atomic or a composite formula. For terms $t_1, \dots, t_n$ over σ and a relation symbol $\mathbf{P} \in \sigma$ with arity n, $t_1 = t_2$ and $\mathbf{P}(t_1, \dots, t_n)$ are *atomic formulas* over σ. If φ_1, φ_2 are formulas over σ, then their negation $\neg\varphi_1$, conjunction $\varphi_1 \wedge \varphi_2$, and disjunction $\varphi_1 \vee \varphi_2$, as well as $\forall\, x : \varphi_1(x)$, $\exists\, x : \varphi_1(x)$, $\forall_k\, X : \varphi_1(X)$ and $\exists_k\, X : \varphi_1(X)$ are *composite formulas.* The operators $\forall_k\, X$ and $\exists_k\, X$ express universal and existential quantification of k-ary relations over the universe of discourse. A formula in second-order logic is in *monadic second-order logic* if for every quantifier $\forall_k$ or $\exists_k$ we have $k = 1$, i.e., only unary relations are quantified.

Let $h, q, R \in \mathbb{N}$ be fixed, and $\mathbf{s}_1, \ldots, \mathbf{s}_R$ be constant symbols. We denote by $\sigma_1 = \sigma^q_{1,R}$ the signature $\{\mathbf{s}_1, \ldots, \mathbf{s}_R, \mathbf{E}\} \cup \{\mathbf{Card}_{m,p} \mid p \in [q], m \in [p]\}$, where $\mathbf{E}$ is a binary relation symbol and $\mathbf{Card}_{m,p}$ are unary relation symbols. Similarly, $\sigma_2 = \sigma^q_{2,R}$ is the signature $\{\mathbf{s}_1, \ldots, \mathbf{s}_R, \mathbf{isIncident}\} \cup \{\mathbf{Card}_{m,p} \mid p \in [q], m \in [p]\}$, where **isIncident** is a binary relation symbol and $\mathbf{Card}_{m,p}$ are unary relation symbols. Then CMS_i is the set of formulas in second-order logic over σ_i. The set of formulas in second-order logic over signature σ_2 with exactly one free set variable, depth of nested quantification of at most h is denoted by $\Phi_R = \Phi^{h,q}_{2,R}$. Note that for any $\varphi \in \mathrm{CMS}_2$ with exactly one free set variable, there are $h, q, R \in \mathbb{N}$ such that $\varphi \in \Phi^{h,q}_{2,R}$.

A *structure* $\langle A, \sigma, \mathcal{I}\rangle$ consists of a *universe of discourse* A, a signature σ and an interpretation function $\mathcal{I}$. At the minimum, the interpretation function maps every constant in σ to an element in A, and every n-ary relation symbol to an n-ary relation over A. A structure is a *model* for a formula over (a subset of) σ if it *satisfies* φ using standard semantics of second-order logic. Note that in order to be a model for φ, $\mathcal{I}$ also has to map free variables of φ to corresponding elements of A or relations over A. Our aim is to find the k cheapest models for some fixed formula in CMS_2 with respect to a cost function $c\colon A \to \mathcal{R}$ for some $k \in \mathbb{N}$.

Restricting this chapter to graph problems enables us to introduce some simplification. We assume that we want to find optimal solutions for some fixed undirected graph $G = (V, E)$. We restrict the problems to be solved to combinatorial optimization problems where the underlying set is E. A formula to characterize feasible solutions for such problems has exactly one non-logical symbol S that represents the solution. In the rest of this chapter, we only consider structures $\langle V \sqcup E, \sigma_2, \mathcal{I}\rangle$ with

- $\mathcal{I}(\mathbf{s}_i) \in V$,
- $\mathcal{I}(\mathbf{isIncident}) = \{(v, e) \in V \times E \mid v \in e\}$, and
- $\mathcal{I}(\mathbf{Card}_{m,p}) = \{V' \subseteq V \mid |V'| \cong p \mod m\}$.

Instead of giving the full structure every time, we simply write $S \models \varphi$ to mean that the structure above with respect to G and $\mathcal{I}(\mathbf{S}) = S$ for the single free variable $\mathbf{S}$ is a model for φ.

Formally, every quantifier ranges over the whole universe of discourse without distinguishing between vertices and edges. However, the formula $\psi(x) = \exists\, y : \mathbf{isIncident}(y, x)$ expresses that the free variable x is an edge, and its negation characterizes vertices. We will use short versions $\forall\, x \in \mathbf{V} : \varphi(x)$ instead of $\forall\, x : \neg\psi(x) \Rightarrow \varphi(x)$, and $\forall\, x \in \mathbf{E} : \varphi(x)$ instead of $\forall\, x : \psi(x) \Rightarrow \varphi(x)$. The quantifiers $\exists$, $\forall_k$ and $\exists_k$ are used accordingly.

Every $\varphi \in \mathrm{CMS}_2$ corresponds to a family of combinatorial optimization problems; an edge set $S \subseteq E$ is a *feasible solution* for φ on G if $S \models \varphi$. We can now define $\boldsymbol{sat}(\varphi) := \{S \subseteq E \mid S \models \varphi\}$ as the set of all feasible solutions for φ on G. For simplicity, we identify these problems with the corresponding formula and refer to combinatorial optimization problems in CMS_2.

The class CMS_2 spans a wide range of combinatorial graph problems that have been studied extensively. Examples are various matching problems, the traveling salesperson problem, coloring and clique problems and many more.

The constant symbols also allow us to model problems with a fixed number of designated vertices. Of particular interest in this thesis is the k shortest simple path problem, which has a corresponding formula in CMS$_2$, too. We need two designated vertices $s, t \in V$ with $\mathcal{I}(\mathbf{s}_1) = s$ and $\mathcal{I}(\mathbf{s}_2) = t$. First, it is easy to see that the formula for connectedness above can easily be adapted to subgraphs of G induced by some edge subset. Second, we can characterize vertices v with at most d incident edges in a given edge subset P using

$$\begin{aligned}&\mathit{degreeAtMost}_i(\mathbf{v}, \mathbf{P}) =\\ &\qquad\forall\, e_1, \ldots, e_{d+1} : \bigwedge_{i\in[d]} (e_i \in \mathbf{P} \wedge \mathbf{isIncident}(\mathbf{v}, e_i)) \Rightarrow \bigvee_{i,j\in[d+1], i\neq j} e_i = e_j.\end{aligned}$$

Then a subset $P \subseteq E$ that is a model for

$$\begin{aligned}&\mathit{isConnected}(\mathbf{P}) \wedge \mathit{degreeAtMost}_1(\mathbf{s}_1)\\ &\qquad\wedge \exists\, e : (e \in \mathbf{P} \wedge \mathbf{isIncident}(\mathbf{s}_1, e))\\ &\qquad\wedge \forall\, v : (\mathit{degreeAtMost}_2(v) \wedge (\mathit{degreeAtMost}_1(v) \Rightarrow (v = \mathbf{s}_1 \vee v = \mathbf{s}_2))\end{aligned}$$

is a simple path from $s = \mathcal{I}(\mathbf{s}_1)$ to $t = \mathcal{I}(\mathbf{s}_2)$ in G.

A formula ψ with designated vertex symbols $\mathbf{s}_1, \ldots, \mathbf{s}_n$ is symmetric in its designated vertices if for every model $\langle A, \sigma, \mathcal{I}\rangle$ for ψ and every permutation π on $[n]$, the structure $\langle A, \sigma, \mathcal{I}'\rangle$ with $\mathcal{I}'(\mathbf{s}_i) = \mathcal{I}(\mathbf{s}_{\pi(i)})$ also models ψ. Note that the above formula is symmetric in $\mathbf{s}_1$ and $\mathbf{s}_2$: Every path from $s = \mathcal{I}(\mathbf{s}_1)$ to $t = \mathcal{I}(\mathbf{s}_2)$ in an undirected graph is also a path from t to s in the same graph. In contrast, the k shortest simple path problem on directed graphs is not symmetric. Structures for directed graphs are special cases of structures for directed hypergraphs, which are covered in Chapter 11. It is very well possible to model problems where different designated vertices have different roles. For example, every subset $S \subseteq E$ with $s \in S$ and $t \notin S$ is a model for $\mathbf{s}_1 \in \mathbf{S} \wedge \mathbf{s}_2 \notin \mathbf{S}$, using the same interpretation function $\mathcal{I}$ as above.

Let G be an undirected graph and $(T = (U, F), b)$ be a rooted binary tree decomposition. The designated vertices of $G(u)$ for any $u \in U$ are an injective sequence on $b(u)$ of length $n = |b(u)|$. In particular, we have $b(u) = \{s_{u,1}, \ldots, s_{u,n}\}$, designated vertex symbols $\mathbf{s}_1, \ldots, \mathbf{s}_n$ and $\mathcal{I}(\mathbf{s}_i) = s_{u,i}$ for structures modeling $G(u)$. Now assume that u is an inner node with child nodes w_1 and w_2. For each designated vertex $s_{u,m}$ of $G(u)$ and $i \in [2]$, either $s_{u,m}$ is also a designated vertex of $G(w_i)$ or $s_{u,m} \notin b(w_i)$. Let $\varrho_i \colon [|b(u)|] \to ([|b(w_i)|] \cup \{\varepsilon\})$ be the function with and $\varrho_i(m) = \varepsilon$ if the $s_{u,m} \notin b(w_i)$, and $\varrho_i(m) = n$ for $s_{u,m} = s_{w_i,n}$. Further, let $M = \{\{i, j\} \mid \{v_i, v_j\} \in E(u)\}$ be the set of unordered pairs of indices i, j such that the edge $\{v_i, v_j\}$ is introduced by u. The *type* of u is the triple $(M, \varrho_1, \varrho_2)$. Note that for any sets M, N_1, N_2 with $N \neq N'$, $f_1 \colon M \to N_1$ and $f_2 \colon M \to N_2$ are different functions even if $f_1(m) = f_2(m)$ for every $m \in M$. As a consequence, two nodes always have different types if their bag sizes or the bag sizes of their child nodes differ.

Let $R \in \mathbb{N}$. A *CMS$_2$ problem on undirected graphs* is a combinatorial optimization problem whose input is an undirected graph $G = (V, E)$ with designated vertices $s_1, \ldots, s_r \in V$, $r \leq R$, and edge cost function $c\colon E \to \mathbb{R}$, and where the set of feasible solutions is $\boldsymbol{sat}(G, \varphi)$ for some $\varphi \in \Phi_R$. The value of an optimal solution in $\boldsymbol{sat}(G, \varphi)$ with respect to c is $v(G, \varphi)$.

Let G be an undirected graph and $\varphi \in \Phi_R$. Sets of solutions, and $\boldsymbol{sat}(G, \varphi)$ in particular, are sets of n-tuples of sets. Let X, Y be q-tuples of sets, and A, B be sets of n-tuples of sets. Then X and Y *interfere* if any set of X has a nonempty intersection with any set of Y, i.e., there exist $i, j \in [n]$ with $X_i \cap Y_j \neq \emptyset$. By extension, A and B *interfere* if some $X \in A$ interferes with some $Y \in B$. If A and B do not interfere, then $A \uplus B$ denotes the *extended union* of A and B, the set of all element-wise disjoint unions of each $X \in A$ and $Y \in B$, i.e.,

$$A \uplus B := \{(X_1 \sqcup Y_1, \ldots, X_q \sqcup Y_q) \mid X \in A, Y \in B\}.$$

A *rich tree decomposition* of an undirected graph $G = (V, E)$ is a tree decomposition $(T = (U, F), b)$ together with a mapping $\iota\colon E \to U$ such that $e \subseteq b(\iota(e))$ for every $e \in E$. The tree decomposition node $\iota(e)$ *introduces* the edge e. Note that $\iota^{-1}(u)$ for a tree decomposition node u is the set of edges of G that are introduced by u.

Lemma 9.1. *Let $w \in \mathbb{N}$, and $G = (V, E)$ be an undirected graph with treewidth at most w. A tree decomposition $(T = (U, F), b)$ of G with width w can be extended to a rich tree decomposition in time $\mathcal{O}(|V|)$.*

Proof. We compute ι in a depth-first search on T. When we discover node v from node u, we compute the set of new vertices $\Delta_{u,v} = b(v) \setminus b(u)$. An edge $e = \{w_1, w_2\}$ with $w_1 \in \Delta_{u,v}$ and $w_2 \in b(v)$ is subset of $b(v)$, but not subset of $b(u)$. Therefore, v is the lowest common ancestor of all nodes that are eligible to introduce e, and we choose $\iota(e) = u$. Let $E_{u,v}$ be the set of these edges.

Computing $\Delta_{u,v}$ can be done in $\mathcal{O}(w)$. The sets $E_{u,v}$ can be computed by iterating the adjacency list of every vertex in $\Delta_{u,v}$ and testing for the edge $\{w_1, w_2\}$ if w_2 in $b(v)$. Every edge is checked twice, once for each endpoint. The test $w_2 \in b(v)$ can be done in $\mathcal{O}(w)$. Hence, the total running time is $\mathcal{O}(|V|)$ for fixed w. $\square$

Let $(T = (U, F), b)$ be a rich tree decomposition for an undirected graph $G = (V, E)$, $u \in U$ some node of T, $\varphi \in \mathrm{CMS}_2$ and $S \in \boldsymbol{sat}(G, \varphi)$. The set $E(u)$ contains all edges that are introduced by a node in $T(u)$. The *subsolution* $S(u)$ of S is the subset of edges in S that are introduced by a node in $T(u)$, i.e., $S(u) = S \cap E(u)$.

9.2. Related Work

Bertelè and Brioschi [10] were the first to introduce a graph parameter, namely its *dimension*, that was later proven to be equivalent to treewidth by Bodlaender [13]. Research on treewidth was popularized by Robertson and Seymour. Most famously, they proved [90] Wagner's conjecture. A consequence of this is that any bounded-treewidth graph class can be characterized by a finite set of graph minors.

Bodlaender [12] showed that graphs of treewidth at most w can be embedded into a w-tree in linear time, for any fixed $w \in \mathbb{N}$. It is straightforward to deduce a tree decomposition of width w from an embedding in a w-tree. For other graphs, their algorithm finds proof that the treewidth is greater than w. Computing tree decompositions is parallelizable, too. For graphs with n vertices, there is an algorithm by Bodlaender [14] to compute tree decompositions with width w in time $\mathcal{O}(\log n)$ on $\mathcal{O}(n^{3w+4})$ processors in the EREW PRAM model, and an algorithm by Bodlaender and Hagerup [15] with running time $\mathcal{O}((\log n)^2)$ using only $\mathcal{O}(n)$ operations with optimal speedup, and therefore $\mathcal{O}(\frac{n}{(\log n)^2})$ many processors in the CRCW PRAM model. Elberfeld, Jakoby and Tantau [39] showed that tree decompositions can also be computed using only logarithmic memory space.

Many problems, including NP-hard ones, are solvable in linear time on graph classes of bounded treewidth. Notable examples are finding minimum independent or dominating sets [2, 6, 8, 70, 93], Hamiltonian cycles, vertex coloring a graph with a minimum number of colors [6], q-Coloring [49], computing minimum weight matchings [8], odd cycle traversal [48], the Steiner tree problem [26] and the two-stage stochastic Steiner tree problem [74]. In an effort to generalize these results, a line of research mainly by Courcelle studies the expressive power of counting monadic second-order logic. Courcelle's theorem [33] states that graphs of bounded treewidth satisfying a fixed formula $\varphi \in \mathrm{CMS}_2$ with no free variables can be recognized in linear time. The algorithm is usually presented in terms of a bottom-up tree automaton. It computes a state for every node of a rooted binary tree decomposition in a bottom-up manner. To compute the state of an inner node, only the states of its two child nodes have to be considered. The set of possible states is finite and does not depend on the graph size. We can decide whether the graph has the property in question by only looking at the state of the root node. He also conjectured that every graph property that can be recognized by a bottom-up tree automaton can also be characterized by a formula in CMS_2. Only very recently, Bojańczyk and Pilipczuk [16] proposed a proof for this conjecture, establishing equality of the two classes.

Around the beginning of the 1990s, Courcelle and Mosbah [35] and, independently, Arnborg, Lagergren and Seese [5] made the transition from decision problems to optimization problems. Both allow multiple free variables that may represent vertex or edge subsets, and each free variable can have its own cost function. This enables us to model, e.g., graph coloring problems where coloring some vertex v in color 1 is more expensive than coloring it in color 2. Interpreting these algorithms in terms of bottom-up tree automata would lead to infinite sets of possible states. However, they still use dynamic programming, which bottom-up tree automata are special cases of. Courcelle and Mosbah show [35] that for every monadic second-order formula φ that characterizes feasible solutions on graphs with bounded treewidth, there is a set of formulas Φ with $\varphi \in \Phi$ that only has finitely many equivalence classes such that the set of feasible solutions for φ can be decomposed into sets of feasible solutions on appropriate subgraphs, for formulas in Φ. They propose to compute the set of feasible solutions for every formula in Φ at the leaves of the tree decomposition, aggregating it in a way that again allows processing of the inner nodes by only looking at constantly many aggregated values at child nodes. Their approach is also a building block of our generalization to k-best optimization, and is therefore described in more detail in Section 10.1.

To the best of our knowledge, there is no previous research on k-best enumeration on bounded-treewidth graphs at all.

10. Undirected Graphs

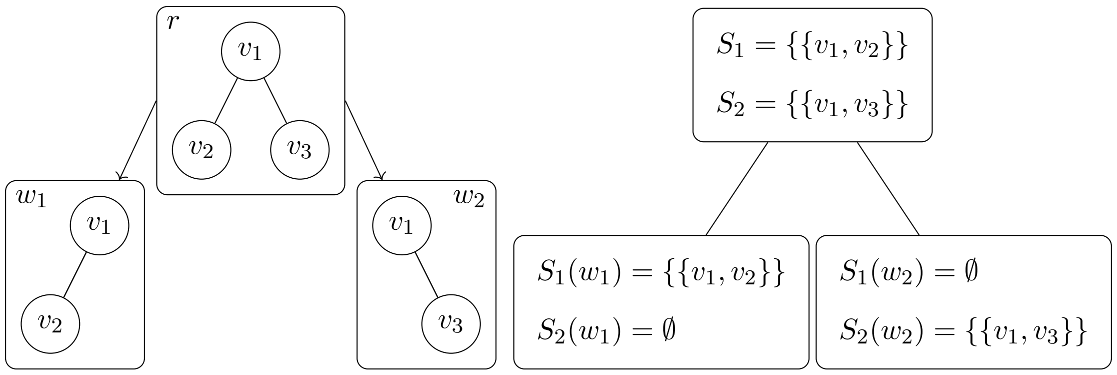
We present a generalization of Courcelle and Mosbah's optimization algorithm on graphs with bounded treewidth to k-best optimization. This generalization is already heavy on notation for the case of edge subset problems on undirected graphs. We present the algorithm for this case first. In Section 10.1, we demonstrate how to enumerate the k best solutions of a CMS_2 optimization problem on graphs with bounded treewidth for a fixed k in time $\mathcal{O}(|V|)$. In Section 10.2, k is considered part of the input, and we demonstrate how to extend the first version for fixed k to find the next-best solution in logarithmic extra time. We explain adjustments required for the general case of directed hypergraphs with multiple vertex and edge subsets in Chapter 11. The algorithm for extracting a solution in case of a fixed k can be applied without modification to the general algorithm. Thus, we can compute the values of the k best solutions in time $\mathcal{O}(|V| + k \log |V|)$, and then extract any of those k solutions in $\mathcal{O}(|V|)$ time each.

For the rest of this section, let $w \in \mathbb{N}$, $G = (V, E)$ be the undirected input graph with edge cost function $c \colon E \to \mathbb{R}$ and treewidth at most w, and $\varphi \in \mathrm{CMS}_2$ be the formula that characterizes feasible solutions of the combinatorial optimization problem we seek to solve. We compute a rich binary tree decomposition $(T = (U, F), b, \iota)$ for G with width w and size $\mathcal{O}(|V|)$ and select a random root $r \in U$. According to Bodlaender [12] and Lemma 9.1, (T, b, ι) can be computed in linear time.

10.1. k-Best Optimization for Fixed k

We are interested in solutions for φ on the whole graph $G = G(r)$. Let w_1, w_2 be the child nodes of r. Any solution $S \in \boldsymbol{sat}(G(r), \varphi)$ can be partitioned into subsolutions $S(w_1)$ and $S(w_2)$, and a set of edges that are introduced at r. Each of these sets can be empty. For some simple problems, as in the case $\boldsymbol{sat}(G(r), \varphi) = 2^E$, we can characterize both sets of subsolutions $\mathcal{S}_i = \{S(w_i) \mid S \in \boldsymbol{sat}(G(r), \varphi)\}$, $i \in [2]$, by using another formula ψ_i with $\boldsymbol{sat}(G(w_i), \psi_i) = \mathcal{S}_i$. We could then combine any solution in $\boldsymbol{sat}(G(w_1), \psi_1)$ with any solution in $\boldsymbol{sat}(G(w_2), \psi_2)$ to obtain a solution in $\boldsymbol{sat}(G(r), \varphi)$. For other problems, however, this is not possible using a single pair (ψ_1, ψ_2) of formulas. An example where this is not possible, the maximum matching problem, can be seen in Figure 10.1. The naive approach would consider the infeasible solution $S_1 \sqcup S_2$ as feasible.

Instead, we can use a sequence of pairs of formulas for these problems. Feferman and Vaught [45] and Courcelle [33] demonstrate how to decompose φ into pairs $(\psi_1^k, \psi_2^k), k \in [l]$ for some $l \in \mathbb{N}$. The decomposition only depends on φ and the type of r. The following lemma is adapted to tree decompositions from Courcelle and Mosbah [35].



(a) Tree decomposition of G (b) Subsolutions for each bag

Figure 10.1.: Example graph for the maximum matching problem. Vertices of the input graph are depicted as circles; nodes of its tree decomposition as rectangles with rounded corners. Both feasible solutions S_1, S_2 with their corresponding subsolutions can be seen in the right figure. The union $S_1(w_1) \sqcup S_2(w_2) = \{\{v_1, v_2\}, \{v_1, v_3\}\}$ is not feasible in $G(r)$.

Lemma 10.1. *Let $\varphi \in \Phi_R$, and let (T, b, ι) be a rich binary tree decomposition rooted in r for an undirected graph G. If r is not a leaf, it has two child nodes w_1, w_2, respectively, and there exist $l \in \mathbb{N}$, $I_k \subseteq \iota^{-1}(r)$ and $\psi_1^k, \psi_2^k \in \Phi_R$ for each $k \in [l]$ such that*

$$\boldsymbol{sat}(G, \varphi) = \bigsqcup \Big\{ \{I_k\} \uplus \boldsymbol{sat}(G(w_1), \psi_1^k) \uplus \boldsymbol{sat}(G(w_2), \psi_2^k) \mid k \in [l] \Big\}, \tag{10.1}$$

where l only depends on the type of r.

Proof. This follows from Lemma 11.5 and Lemma 11.1. □

As indicated, the proof of Lemma 10.1 is deferred to Chapter 11.

In the situation of Lemma 10.1, we call ψ_i^k a *child formula* of φ with respect to the type of r. For $k \in [l]$, we call (ψ_1^k, ψ_2^k) a *pair of child formulas.* Note that Lemma 10.1 can be applied recursively to $G(w_i)$ and ψ_i^k for $i \in [2]$, $k \in [l]$ because of $\psi_i^k \in \Phi_R$.

Corollary 10.2. *In the situation of Lemma 10.1, for every $S \in \boldsymbol{sat}(G, \varphi)$, there is a unique $k \in [l]$ with $S \in \{I_k\} \uplus \boldsymbol{sat}(G(w_1), \psi_1^k) \uplus \boldsymbol{sat}(G(w_2), \psi_2^k)$.*

Proof. This follows directly from the big union operator being a disjoint one. □

The *unique fitting pair of child formulas* of a solution $S \in \boldsymbol{sat}(G, \varphi)$ is the pair (ψ_1^k, ψ_2^k) with the unique k according to Corollary 10.2.

Lemma 10.3. *In the situation of Lemma 10.1, if $e \in S$ for some $S \in \boldsymbol{sat}(G(w_i), \psi_i^k)$, then e is introduced by a node in $T(w_i)$.*

The proof of Lemma 10.3 is deferred to Chapter 11.

Corollary 10.4. *Let $M(u)$ be the union of all feasible solutions for all formulas in Φ_R. For every edge $e \in E$, there is at most one node u of T with $e \in M(u) \setminus (M(v_1) \cup M(v_2))$ for child nodes v_1, v_2 of u, and we have $\iota(e) = u$.*

The original algorithm from Courcelle and Mosbah aggregated each occurring set $\mathcal{S}$ of solutions to a single value, the cost of an optimal solution in $\mathcal{S}$. Since we are only interested in linear cost functions, the cost of the union of two disjoint solutions is just the sum of the costs of the two solutions. The cost of an optimal solution in the disjoint union of solution sets is the minimum of the individual optimal solutions. They obtained the following result for directed hypergraphs; see Chapter 11 for the transition to undirected graphs.

Lemma 10.5. *In the situation of Lemma 10.1, we have*

$$v(G(r), \varphi) = \min \left\{ c(I_k) + v(G(w_1), \psi_1^k) + v(G(w_2), \psi_2^k) \mid k \in [l] \right\}, \tag{10.2}$$

where $v(H, \psi)$ is the value of an optimal solution in $\boldsymbol{sat}(H, \psi)$ for some graph H and some formula ψ.

The basic algorithm of Courcelle and Mosbah, called *CM algorithm* henceforth, is the dynamic programming equivalent to the recursive application of Lemma 10.5 and computes the value of an optimal solution in $\boldsymbol{sat}(G, \varphi)$. For every node $u \in U$, we maintain a dynamic programming table (*DP table*) which maps each $\psi \in \Phi_R$ to the value $v(G(u), \psi)$. Populating the table of u is called *evaluation* of u.

To evaluate a leaf node u, we initialize all optimal solution values with positive infinity. We enumerate every edge subset S of $G(u)$. For every ψ, we check if S is feasible for ψ, and store the value of the cheapest feasible solution seen so far in the table. To evaluate an inner node u, we utilize the child formula relation and apply Lemma 10.5. This requires that all child nodes of u have already been evaluated.

Conceptually, the above algorithm is a depth-first search on T, starting at its root r. Every time we finish a node v, we evaluate it. Since the number of formulas and the number of child formulas per formula are fixed, the overall running time is linear in the size of T.

This algorithm can be modified to find the values of the two best solutions by simply swapping some operators. Instead of reducing a set of feasible solutions at the leaves only to its optimal value, we can just as well reduce it to the value of its two best solutions, resulting in a non-decreasing sequence of two solution values. To evaluate an inner node u, we apply Lemma 10.5 again, but using $\min_2$ instead of min, and $+_2$ instead of $+$. Both $\min_2$ and $+_2$ are binary operators operating on non-decreasing pairs of solution values. The aggregation $\min_2$, applied to (a_1, b_1), (a_2, b_2), returns $(a_1, \min(b_1, a_2))$ if $a_1 < a_2$, and $(a_2, \min(a_1, b_2))$ otherwise. In other words, it picks the two smallest values from a_1, a_2, b_1, b_2 in non-decreasing order. The accumulation $+_2$, applied to the same operands, returns, $(a_1 + a_2, \min(a_1 + b_2, a_2 + b_1))$. This is equivalent to considering every sum of one value from the first operand and one value from the second operand, i.e., $a_1 + a_2$, $a_1 + b_2$, $b_1 + a_2$ and $b_1 + b_2$, and again picking the two smallest in non-decreasing order. Note that we always have $a_1 + a_2 \leq \min(a_1 + b_2, a_2 + b_1) \leq b_1 + b_2$, allowing for the simplification above. Applying these operators takes constant time, so the overall asymptotic running time is not changed by this operator swap.

Lemma 10.6. *Let $w \in \mathbb{N}$, and P be the combinatorial optimization problem characterized by a formula $\varphi \in \Phi_R$ and cost functions c. Given an undirected graph G, the CM algorithm computes the values of the two best solutions satisfying φ on G in linear time when used with the operators min_2 and $+_2$.*

Proof. Correctness follows from Lemma 10.5 and the fact that the operators $\min_2$ and $+_2$ can be evaluated in constant time. Alternatively, it follows from directly from Lemma 11.5 and Lemma 11.1.

□

We refer to the CM algorithm in conjunction with the $\min_2$ and $+_2$ operators as the *2-CM algorithm.*

When aggregating sets of solutions to a sequence of values of bounded length as described above, much information about the solutions themselves is lost. We cannot check if some edge e is part of an optimal solution by only looking at the root node, since we only store the value of the solutions. Further, we cannot check if two equal solution values in a DP table of a node u refer to the same solution, or to two different solutions with the same value.

Example 5. We consider the toy problem of computing minimum weighted matchings of cardinality exactly three, or the minimum three non-adjacent edges problem, on undirected graphs of treewidth at most $w \geq 2$. Figure 10.2 shows an example input graph $G = (V, E)$ with treewidth 2 and edge costs c and a rich binary tree decomposition $(T = (U, F), b, \iota)$ of G with width 2.

The formula $\psi_I^{n,i}$ for $n \in [w]$, $i \in [3]$ and $I \subseteq [3]$ with one free variable $\mathbf{S}$ expresses that a set S is a matching consisting of exactly i edges of a graph with exactly n designated vertices $s_1, \dots, s_n$ such that s_i has an incident edge in S iff $i \in I$, i.e.,

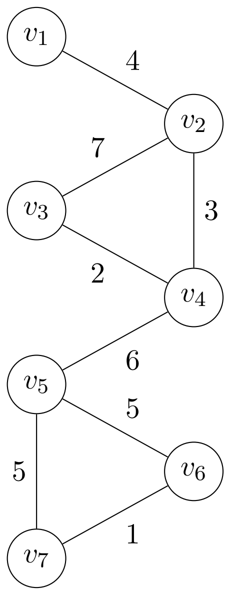
$$\psi_I^{n,i}(\mathbf{S}) = (\forall\, v \in V : \mathit{degreeAtMost}_1(\mathbf{v}, \mathbf{S})) \wedge \bigwedge_{i \in I} (\exists\, e \in \mathbf{S} : \mathbf{isIncident}(\mathbf{s}_i, e)).$$

Then feasible solutions for the three non-adjacent edges problem on a graph with n designated vertices are exactly the structures that model the formula

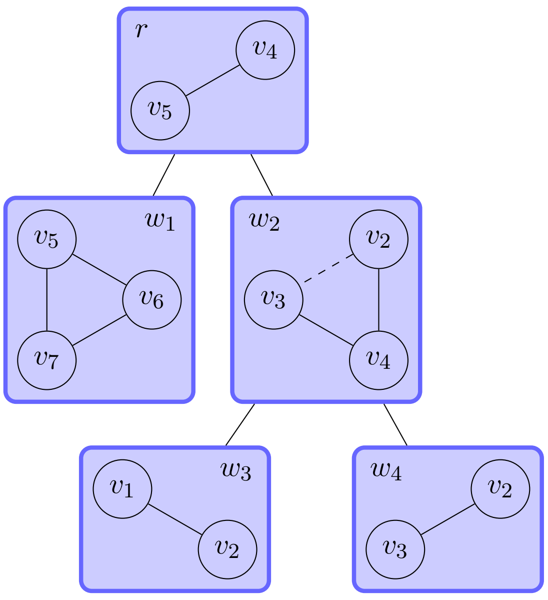
$$\varphi(\mathbf{S}) := \bigvee_{I \subseteq [3]} \psi_I^{n,3}(\mathbf{S})$$

with one free edge set variable $\mathbf{S}$.

Consider a solution S that does not contain edges that are introduced in r, i.e., $S \cap \iota^{-1}(r) = \emptyset$. As a consequence, we have $a = |S(w_1)|$, $b = |S(w_2)|$, $a, b \in [3]$ and $a + b = 3$. In other words, a edges in S were chosen from $G(w_1)$ and b edges were chosen from $G(w_2)$. Both $G(w_1)$ and $G(w_2)$ are graphs with three designated vertices, so there is a pair of formulas $(\psi_I^{3,a}, \psi_J^{3,b})$ with $I, J \subseteq [3]$ and $S \in \boldsymbol{sat}(G(w_1), \psi_I^{3,a}) \uplus \boldsymbol{sat}(G(w_2), \psi_J^{3,b})$.



(a) Input graph G



(b) Rich binary tree decomposition of G

Figure 10.2.: Weighted undirected graph $G = (V, E)$ and a rich binary tree decomposition $(T = (U, F), b, \iota)$. Graphs inside a blue node $u \in U$ are the subgraphs $G[b(u)]$. Every edge $e \in E$ is introduced by the unique node that depicts e as a solid edge.

Now consider a solution S that contains edges that are introduced in r and are therefore not part of the subsolutions $S(w_1)$ and $S(w_2)$, i.e., we have a non-empty intersection $M = S \cap \iota^{-1}(r)$. With a, b as above, we now have $a + b + |M| = 3$. Let $U_i = \{i \in [n] \mid \mathbf{s}_i$ of $G(w_i)$ is not covered by $M\}$ be the indices of designated vertices of the subgraph $G(w_i)$, $i \in [2]$, that are not covered by an edge in M. Let C be the set of vertices in $b(r)$ that are covered by an edge in M. These vertices may not be covered again by edges selected in subsolutions. There is a pair of formulas $(\psi_I^{3,a}, \psi_J^{3,b})$ with $I \in [3] \subseteq U_1$, $J \in [3] \subseteq U_2$ and $S \in \{M\} \uplus \boldsymbol{sat}(G(w_1), \psi_I^{3,a}) \uplus \boldsymbol{sat}(G(w_2), \psi_J^{3,b})$.

For the root node of the example graph, we have

$$\begin{aligned}
&\boldsymbol{sat}(G(r), \varphi) = \\
&\quad \bigsqcup \Big\{ \boldsymbol{sat}(G(w_1), \psi_I^{3,a}) \uplus \boldsymbol{sat}(G(w_2), \psi_J^{3,b}) \uplus \{\emptyset\} \mid I, J \subseteq [3], a + b = 3 \Big\} \\
&\quad \sqcup \bigsqcup \Big\{ \boldsymbol{sat}(G(w_1), \psi_I^{3,a}) \uplus \boldsymbol{sat}(G(w_2), \psi_J^{3,b}) \uplus \mathcal{S}_{v_4 v_5} \mid I, J \subseteq [2], a + b = 2 \Big\} ,
\end{aligned} \tag{10.3}$$

where $\mathcal{S}_{v_4 v_5} = \{\{\{v_4, v_5\}\}\}$ is the set consisting of a single solution consisting only of edge $\{v_4, v_5\}$. This decomposition of φ assumes that the order on the designated vertices are v_4, v_5 for $G(r)$, v_6, v_7, v_5 for $G(w_1)$ and v_2, v_3, v_4 for $G(w_2)$. The child formulas of $\psi_I^{n,i}$ can be found similarly. Thus, the set $\mathcal{F} = \{\varphi\} \sqcup \{\psi_I^{n,i} \mid n \in [w], i \in [3], I \subseteq [3]\}$ is closed under the child formula relation from Lemma 10.1.

Initially, the 2-CM algorithm evaluates the three leaf nodes w_1, w_2 and w_3. For the single edge e in $G(w_i)$, $i \in \{3, 4\}$, the only formula with solution $\{e\}$ is $\psi_{\{1,2\}}^{2,1}$. The only other solution, $\emptyset$, is only feasible for $\psi_\emptyset^{2,0}$. None of the other formulas in $\mathcal{F}$ is feasible. Similarly for $G(w_1)$, there is exactly one formula with the single solution $\{e\}$ for each of the three edges.

The inner node w_2 has to be evaluated next. For each formula ψ in $\mathcal{F}$, we adapt Equation (10.3) to ψ, replacing each $\boldsymbol{sat}(\cdot, \cdot)$ with $v(\cdot, \cdot)$. For example, a solution for $\psi_{[3]}^{3,2}$ either consists of

- three non-adjacent edges introduced by w_2 that cover all three designated vertices of $G(w_3)$ (which is impossible in the example graph, since ι^{-1} forms a triangle), and no edges introduced below w_2, or
- one edge introduced by w_2 that covers the first two designated source vertices v_2 and v_3, one edge introduced by w_3 that covers the third designated source vertex v_4, and no edge introduced in w_4, or
- ...

In our case, the only feasible solution for $\psi_{[3]}^{3,2}$ is $\{\{v_1, v_2\}, \{v_3, v_4\}\}$, which corresponds to the decomposition $\{\{\{v_3, v_4\}\}\} \uplus \boldsymbol{sat}(G(w_3), \psi_{\{1,2\}}^{2,1}) \uplus \boldsymbol{sat}(G(w_4), \psi_\emptyset^{3,0})$. After w_2 is evaluated, the final node r can be evaluated, and the row of φ contains the values of the two best solutions for the minimum three non-adjacent edges problem.

The tables in Figure 10.3 show the values of the best two solution for every formula at every node. If formula ψ is missing in the table of node u, $\boldsymbol{sat}(G(u), \psi)$ is empty and

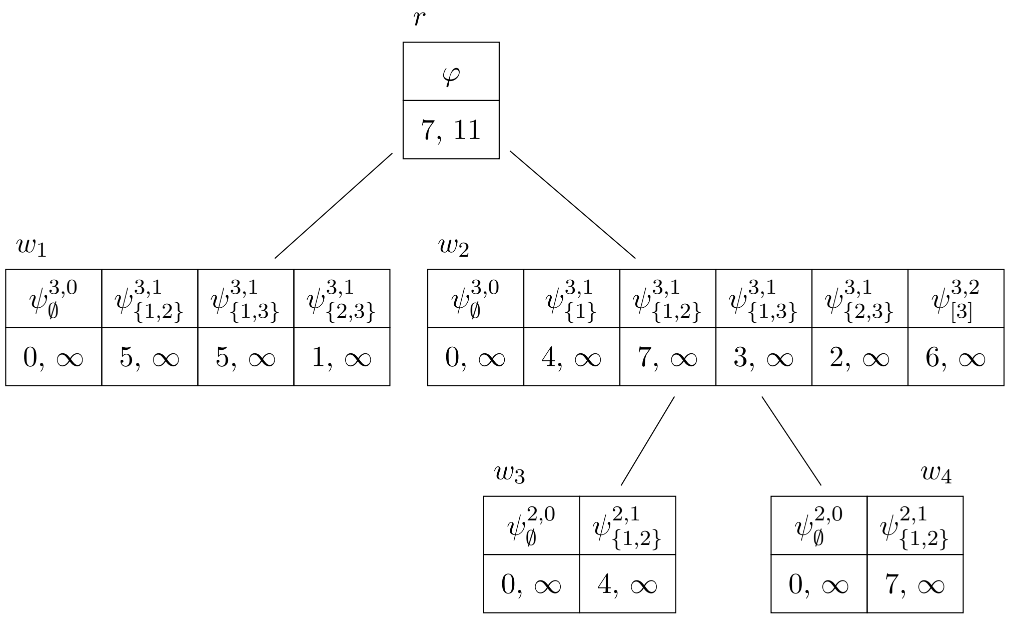


Figure 10.3.: Dynamic programming tables of the 2-CM algorithm for the example input graph and tree decomposition in Figure 10.2.

the table contains (∞, ∞) in the row of ψ. An exception to this is the root node, where only φ is shown.

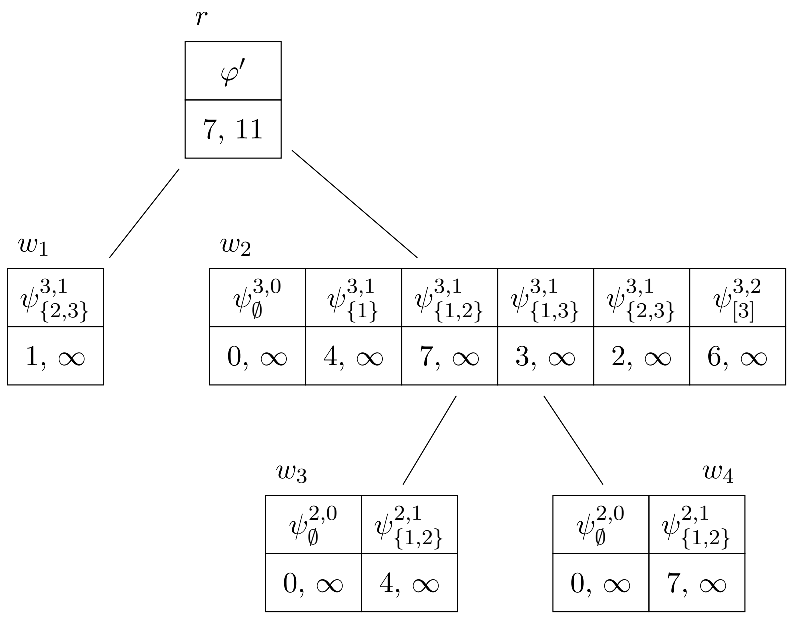
Finally, note that the type of r is different from the type of w_2. This already follows from the fact that the child nodes of r have bag sizes of two and three, while the child nodes of w_2 both have bag size two. Therefore, decomposition of a formula according to Lemma 10.1 can be different for r and w_2.

10.2. k-Best Optimization for General k

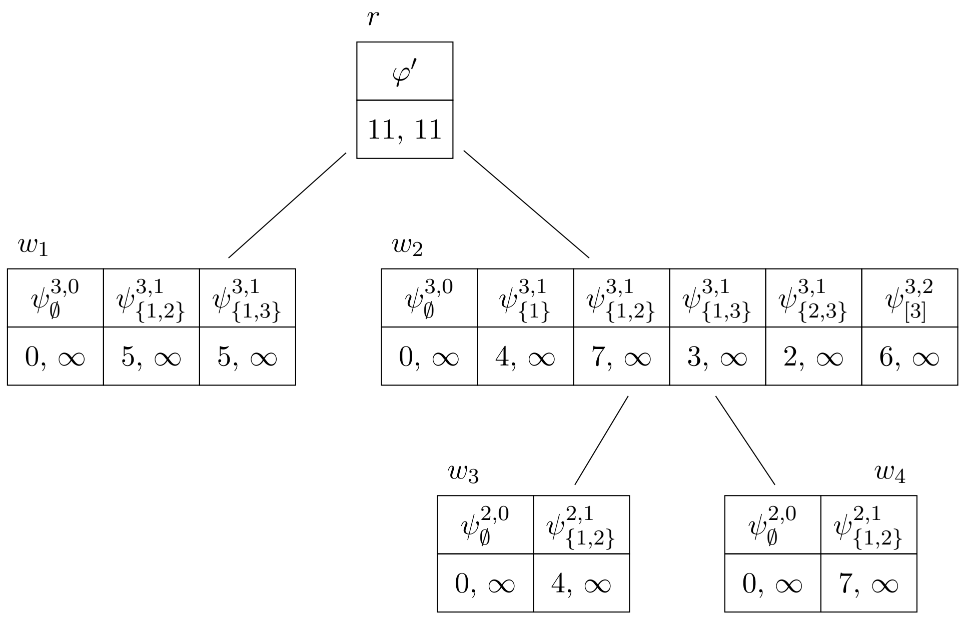
In this section, we extend the algorithm from Section 10.1, where the number k of solution values to enumerate was considered fixed to 2 and therefore independent from the input, to an algorithm that expects k as part of the input.

Naively, we could simply generalize the approach used for $k = 2$. Instead of using the operators $\min_2$ and $+_2$, we would use $\min_k$ and $+_k$. However, evaluating these two general operators requires time $\mathcal{O}(k)$. The resulting algorithm would have a running time of $\mathcal{O}(kn)$. We improve upon this bound and obtain an algorithm with running time $\mathcal{O}(n + kD)$, where D is the depth of the tree decomposition.

The key idea revolves around managing a binary heap of objects that represent subproblems as described in Section 3.2.1. The Hamacher/Queyranne binary subproblem heap approach requires an algorithm that solves the 2-best version of a combinatorial optimization problem, so we will use the algorithm from Section 10.1. In order to use


r
φ'
7, 11
w_1
$\psi^{3,1}_{\{2,3\}}$
1, ∞
w_2
$\psi^{3,0}_{\emptyset}$ | $\psi^{3,1}_{\{1\}}$ | $\psi^{3,1}_{\{1,2\}}$ | $\psi^{3,1}_{\{1,3\}}$ | $\psi^{3,1}_{\{2,3\}}$ | $\psi^{3,2}_{[3]}$
0, ∞ | 4, ∞ | 7, ∞ | 3, ∞ | 2, ∞ | 6, ∞
w_3
$\psi^{2,0}_{\emptyset}$ | $\psi^{2,1}_{\{1,2\}}$
0, ∞ | 4, ∞
w_4
$\psi^{2,0}_{\emptyset}$ | $\psi^{2,1}_{\{1,2\}}$
0, ∞ | 7, ∞

(a) Subproblem $\mathcal{S}(\{\{v_6, v_7\}\}, \emptyset)$ that forces inclusion of the pivot


r
φ'
11, 11
w_1
$\psi^{3,0}_{\emptyset}$ | $\psi^{3,1}_{\{1,2\}}$ | $\psi^{3,1}_{\{1,3\}}$
0, ∞ | 5, ∞ | 5, ∞
w_2
$\psi^{3,0}_{\emptyset}$ | $\psi^{3,1}_{\{1\}}$ | $\psi^{3,1}_{\{1,2\}}$ | $\psi^{3,1}_{\{1,3\}}$ | $\psi^{3,1}_{\{2,3\}}$ | $\psi^{3,2}_{[3]}$
0, ∞ | 4, ∞ | 7, ∞ | 3, ∞ | 2, ∞ | 6, ∞
w_3
$\psi^{2,0}_{\emptyset}$ | $\psi^{2,1}_{\{1,2\}}$
0, ∞ | 4, ∞
w_4
$\psi^{2,0}_{\emptyset}$ | $\psi^{2,1}_{\{1,2\}}$
0, ∞ | 7, ∞

(b) Subproblem $\mathcal{S}(\emptyset, \{\{v_6, v_7\}\})$ that forces exclusion of the pivot

Figure 10.4.: Dynamic programming tables of the 2-CM algorithm for the example input graph and tree decomposition in Figure 10.2 if only solutions are considered that include (above) or exclude (below) the pivot edge $\{v_6, v_7\}$.

Frederickson's heap selection algorithm on this binary heap, we need an efficient way to query the value of the second-best solution for each subproblem. Also, the heap does not exist initially, so we also need to be able to create those parts of the heap that are required by the heap selection algorithm efficiently.

Let us assume for now that we have an oracle that, given a subproblem of the original problem characterized by $\varphi \in \Phi_R$, gives us a pivot edge, i.e., an edge e that is an element of the optimal solution, but not an element of the second-best solution, or vice versa, as well as the node of the tree decomposition that introduces e. Following the Hamacher/Queyranne approach, we use a subproblem $\mathcal{S}(I, O)$ and the pivot e to create two subproblems $\mathcal{S}(I \sqcup \{e\}, O)$ and $\mathcal{O}(I, O \sqcup \{e\})$.

First, observe that the two new subproblems are no longer in CMS_2. We do not have literals for edges in CMS_2, so we cannot express $e \notin \mathcal{S}$. Even if we had literals for edges, the number of these literals would be fixed independently from the size of the graph. Otherwise, the set Φ_R would no longer be finite.

Second, even if we had an algorithm similar to 2-CM that solves subproblems that are not in CMS_2, we would not be allowed to compute the complete DP tables for all the subproblems. If we want to enumerate k values using the Hamacher/Queyranne approach, we have to solve $k-1$ many subproblems. We cannot store DP tables of size $\mathcal{O}(n)$ each, because that would result in an algorithm with running time $\Omega(kn)$.

Example 6. We create the first two subproblems for Example 5. The optimal solution is $\{\{v_1, v_2\}, \{v_3, v_4\}, \{v_6, v_7\}\}$, its value is 7. The value of three second-best solutions is 11; we use the solution $\{\{v_1, v_2\}, \{v_3, v_4\}, \{v_5, v_6\}\}$ for now, and explain later in this section how the algorithm copes with ambiguities. Further, assume that our pivot oracle gave us the pivot $e = \{v_6, v_7\}$ as well as the node w_1 that introduces e.

The DP tables for both subproblems are shown in Figure 10.4. Again, columns of infeasible formulas are not on display. Observe that only tables of nodes on the path from r to w_1 have changed.

The observation in the example is actually a general rule. From Lemma 10.1, we know that solutions at some node u cannot contain an edge e if $e \notin E(u)$, i.e., if e is not introduced by a node in $T(u)$. On the other hand, those nodes u for which $e \in E(u)$ are exactly those on the unique path from r to $\iota(e)$ in T.

We extend the functions $\boldsymbol{sat}$ and v to subproblems that are not in CMS_2: For a subproblem P' with feasible solutions $\mathcal{S}(I, O)$, $\boldsymbol{sat}_{P'}(G(u), \psi)$ is the set of subsolutions $S(u)$ for $S \in \boldsymbol{sat}(G(u), \psi)$ with $I \subseteq S$ and $S \cap O = \emptyset$, and $v_{P'}(G(u), \psi)$ are the values of the two best solutions in $\boldsymbol{sat}_{P'}(G(u), \psi)$.

Let T be the evaluation tree of a subproblem P rooted in r with feasible solutions $\mathcal{S}(I, O)$, and e a pivot for P. We describe how to adapt T to a subproblem P' of P with solution set $\mathcal{S}(I', O')$, with either $I' = I \sqcup \{e\}$ and $O' = O$, or $I' = I$ and $O' = O \sqcup \{e\}$, given e and the nodes $u_1, \ldots, u_l$ of the r-$\iota(e)$ path in T. We update the tables bottom up, i.e., for u_i in descending order, starting with u_l. If u_l is a leaf, we simulate the 2-CM algorithm for leaf nodes by enumerating all subsets of $E(u_l)$, so check for each subset S if S is a feasible solution for any of the formulas in Φ_R. However, we do not consider S

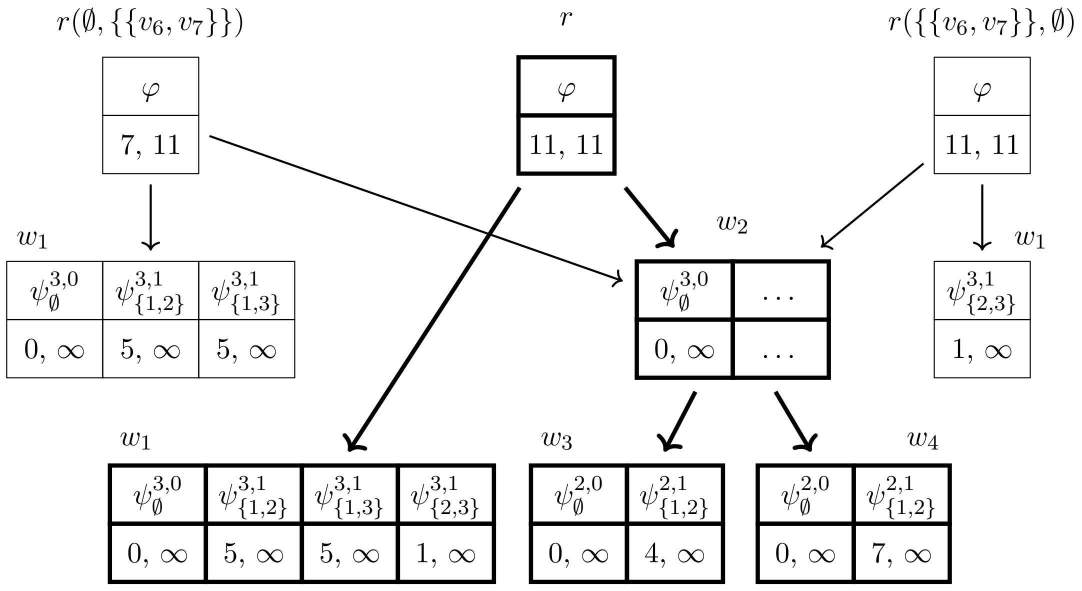


Figure 10.5.: Persistent data structure for Figure 10.2 holding all three versions of the tree of dynamic programming tables from Figure 10.4. From the thick tree of tables, two trees are derived (mixed thin and thin). Note that the original, thick tree is not shown in Figure 10.4.

if $I \not\subseteq S$, since we want to adapt the table to P' where every feasible solution contains e. For each inner node u_i, we simulate the 2-CM algorithm for inner nodes using the subformula relation from Lemma 10.5. This time, we skip every k with $(I' \cap E(u_i)) \not\subseteq I_k$ or with $I_k \cap O' \neq \emptyset$.

Lemma 10.7. *Let T be an evaluation tree of depth d rooted in r for a subproblem with feasible solutions $\mathcal{S}(I, O)$. Given a pivot e for P and the r-$\iota(e)$ path in T, we can adapt T to a subproblem P' whose feasible solutions are either $\mathcal{S}(I \sqcup \{e\}, O)$ or $\mathcal{S}(I, O \sqcup \{e\})$ in time $\mathcal{O}(d)$.*

Proof. We show that the above algorithm finds the values of the two best solutions in $\boldsymbol{sat}_{P'}(G(u_i), \psi)$ for every $\psi \in \Phi_R$.

If u_l is a leaf, we obviously consider exactly those solutions that adhere to the restrictions imposed by P'. Checking if a solution is a model for some ψ is done in the same way as in the 2-CM algorithm.

Let u_i be an inner node. At most one of the child nodes of u_i lies on the r-$\iota(e)$ path, and we compute $\boldsymbol{sat}_{P'}$ correctly for this child node. Let v be a child node of u_i that does not lie on that path. Due to Corollary 10.4, adding edges to I or O that are not in $E(v)$ does not affect the solution sets $\boldsymbol{sat}$ at v. Therefore, we have $\boldsymbol{sat}_P(G(v), \cdot) = \boldsymbol{sat}_{P'}(G(v), \cdot)$, and the DP table of v does not have to be modified. This is also true for every node that is not a child of one of the nodes u_i. It follows that $\boldsymbol{sat}_{P'}$

has already been computed correctly for all child nodes of u_i when we re-evaluate u_i. In other words, for every subsolution S at child node v of u_i we have $(E(v) \cap I) \subseteq S$ and $S \cap O = \emptyset$. The algorithm above only uses those k for the composition of solution sets according to Lemma 10.5 with $(I' \cap E(u_i)) \subseteq I_k$ and $I_k \cap O' = \emptyset$. This is sufficient due to Corollary 10.4. □

Finally, we need a way to compute a pivot e and its introducing node $\iota(e)$. Consider the root node on the right side of Figure 10.5 for subproblem $\mathcal{S}(\{\{v_6, v_7\}\}, \emptyset)$. The column of φ contains the value 11 twice, meaning that the optimal and second-best solution have the same value. We now by constructions that the two solutions behind those values have to be distinct. We also know that the solution with value 11 referenced in the symmetric subproblem $\mathcal{S}(\emptyset, \{\{v_6, v_7\}\})$, as seen on the left side of Figure 10.5, has to be yet another solution, because those two subproblems partitioned the solution space of the parent problem. It is possible, but quite technical to reconstruct the corresponding solutions. Without care, we might want to extract both solutions of the right subproblem, and end up with the same solution twice.

To make things easier, we introduce solution IDs. For every solution value in every DP table, we store some information about the solution for this value. Formally, we compute functions $\pi_u^1 \colon \Phi_R \to [2|\Phi_R|]$ and $\pi_u^1 \colon \Phi_R \to [2|\Phi_R|]$ such that for each $i, j \in [2]$ and $\psi \in \Phi_R$, $\pi_u^i(\psi) = \pi_u^j(\psi)$ iff the i-th solution value for ψ in the DP table of u is meant to reference the same solution as the j-th solution value for ψ in the DP table of u.

We extend the 2-CM algorithm to also compute solution IDs. The computation is straightforward for the leaves: every subset receives its own solution ID. Now consider the evaluation of ψ at an inner node u with child nodes w_1 and w_2. The solution ID of the combination of the i-th best solution value for child formula ψ_1^k at w_1 and the j-th best solution value for child formula ψ_2^k at w_2 is computed as $\pi_{w_1}^i(\psi_1^k) + |\Phi_R| \cdot \pi_{w_2}^i(\psi_2^k)$.

The solution IDs computed so far might exceed the codomain $[2|\Phi_R|]$. In the end, however, only $2|\Phi_R|$ many different solution IDs are used in any table, because this is the number of solution values we store. Hence, we may apply an arbitrary injective compression function $\gamma \colon [2^{|E(u)|}] \to [2|\Phi_R|]$ at leaf nodes, or $\gamma \colon [|\Phi_R|^2] \to [2|\Phi_R|]$ at inner nodes. Storing γ as well as the solution IDs for each DP table only requires $\mathcal{O}(|\Phi_R|)$ extra space per node.

With solution IDs in place, we can find a pivot by following a path starting in the root of an evaluation tree. We only have to make sure that the subsolutions $S_1(u)$ and $S_2(u)$ of the optimal solution S_1 and the second-best solution S_2 differ at the current node u.

We maintain a current node u, $k_1, k_2 \in [2]$, and formulas $\psi_1, \psi_2 \in \Phi_R$. Initially, u is the root of the evaluation tree we want to find a pivot for. For $i \in [2]$, we initialize $k_i = i$, and $\psi_i = \varphi$. We find a pivot recursively as follows.

If u is an inner node, let w_1 and w_2 be the child nodes of u. For $i \in [2]$, we find $k_{i,1}, k_{i,2} \in [2]$ and the unique pair of subformulas $(\psi_{i,1}, \psi_{i,2})$ of ψ_i such that the k_i-th best solution for ψ_i at u is composed of the $k_{i,1}$-th best solution for $\psi_{i,1}$ at w_1 with solution ID $s_{i,1}$, the $k_{i,2}$-th best solution for $\psi_{i,2}$ at w_2 with solution ID $s_{i,2}$, and a subset $I_i \subseteq \iota^{-1}(u)$ of the edges that are introduced by u. We use the solution IDs of the child nodes of u and the compression function of u to find these pairs. If $s_{1,1} \neq s_{1,2}$, we set $u = w_1$, $\psi_1 = \psi_{1,1}$ and $\psi_2 = \psi_{1,2}$, and recurse. Otherwise, if $s_{2,1} \neq s_{2,2}$, we set $u = w_2$, $\psi_1 = \psi_{2,1}$ and $\psi_2 = \psi_{2,2}$, and recurse. Otherwise, i.e., if $s_{1,1} = s_{1,2}$ and $s_{2,1} = s_{2,2}$, we return u and an element in the symmetric difference of I_1 and I_2, and terminate.

If u is a leaf, we do not have to take care of subsolutions. We reconstruct solutions S_1^u and S_2^u with matching solution IDs s_1 and s_2, respectively. We return u and an element of the symmetric difference of S_1^u and S_2^u, and terminate.

Lemma 10.8. *Let G be an undirected graph, $\varphi \in \Phi_R$ be a formula that characterizes feasible solutions of CMS$_2$ problem P on G, and T be an evaluation tree of depth d with solution IDs for the two best solution values of a subproblem P' of P on G. A pivot edge e for P' as well as $\iota(e)$ can be found in $\mathcal{O}(d)$ time.*

Proof. We use the recursive algorithm above and maintain the invariants

1. at the beginning of the j-th iteration, the unique simple path from the root to u has j edges, or the algorithm was already terminated,
2. for $i \in [2]$, there is an optimal solution S_1 and a second-best solution S_2 in $\boldsymbol{sat}_{P'}(G, \varphi)$ such that $S_i(u)$ is the k_i-th best solution for ψ_i at u, and
3. $\pi_{k_1}(\psi_1) \neq \psi_{k_2}(\psi_2)$.

Initially, the first and third invariant hold trivially. The second invariant holds because otherwise the optimal solution would be the same as the second-best solution.

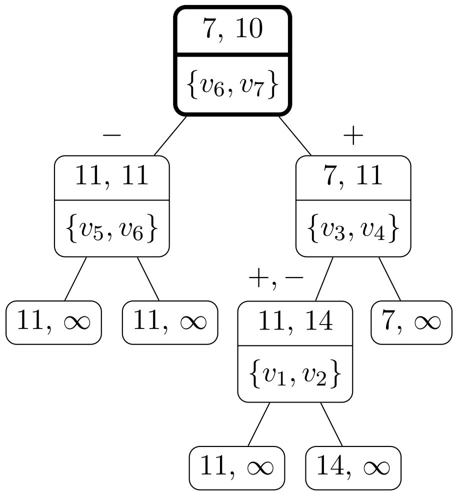
Assume that the invariants hold at the beginning of an iteration. If the algorithm terminates, no variables are modified, leaving the invariants unaffected. The only remaining case is u being an inner node, and there are still subsolutions of $S_1(u)$ and $S_2(u)$ that differ at child node w_1 or w_2 of u. The correct child node is chosen by considering solution IDs, so the second invariant follows for the exact same solutions S_1 and S_2, and the third invariant follows directly. By setting u to one of its child nodes, we increase the depth of u, so the first invariant holds at the beginning of the next iteration.

Let S_1, S_2 be the two solutions of the second invariant. Upon termination, there are no subsolutions of $S_1(u)$ and $S_2(u)$ that differ. Because of the third invariant, $S_1(u)$ and $S_2(u)$ themselves have to differ, so the only difference can lie within the edges introduced by u.

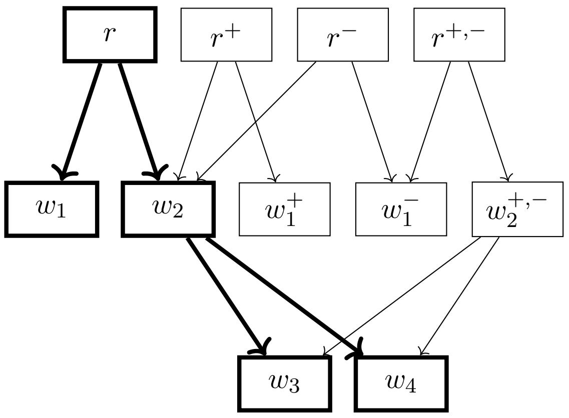
Because of the first invariant, the algorithm terminates after at most d iterations. An iteration basically consists of the enumeration of all pairs of child formulas for ψ_1 and ψ_2, resulting in a total running time of $\mathcal{O}(d)$. □

The above algorithm for finding the pivot can now replace our oracle.

Corollary 10.9. *Let G be an undirected graph, $\varphi \in \Phi_R$ be a formula that characterizes feasible solutions of CMS$_2$ problem P on G, and T be an evaluation tree of depth d with solution IDs for the two best solution values of a subproblem P' of P on G. The evaluation tree T for P can be updated to be an evaluation tree for P' in time $\mathcal{O}(d)$.*



(a) Binary subproblem heap



(b) DP tables in persistent heaps

Figure 10.6.: Subproblem heap and persistent evaluation trees for non-uniquely solvable subproblems for Example 5. Subproblem upper parts: two best solution values; lower parts: selected pivot edge. DP tables with superscripts are created during the DP table tree update for the corresponding subproblem. The evaluation tree of the original problem is thick.

Proof. We can find a pivot e and its introducing node u in time $\mathcal{O}(d)$ (Lemma 10.8), and use this information to update the $\mathcal{O}(d)$ many DP tables bottom-up it time $\mathcal{O}(d)$ (Lemma 10.7). $\square$

The situation for the two child problems of the problem in Example 5 with pivot edge $\{v_6, v_7\}$ is illustrated in Figure 10.5. The complete binary heap of subproblems for one possible selection of pivot edges in Example 5 together with the evaluation trees of all subproblems with at least two feasible solutions, represented as persistent heaps, can be seen in Figure 10.6.

The presence of solution IDs also makes it straightforward to extract a solution from an evaluation tree in linear time.

Lemma 10.10. *Let $G = (V, E)$ be an undirected graph, $\varphi \in \Phi_R$ be a formula that characterizes feasible solutions of problem P on G, and T be an evaluation tree of depth d with solution IDs of a subproblem P' of P on G. Given $\psi \in \Phi_R$, $i \in [2]$ and $u \in U$, the i-th best solution in $\boldsymbol{sat}_{P'}(G(u), \psi)$ can be computed in time $\mathcal{O}(d)$.*

Proof. We perform a depth-first search, starting at (r, ψ). At each inner node u with child nodes w_1, w_2, we recurse using (w_1, ψ_1) and (w_2, ψ_2) where solution IDs match, reconstructing the matching sets I_k according to Lemma 10.1 as we go. At a leaf u, we reconstruct the matching subset of $E(u)$ accordingly. We visit every node once, and only require constant time per node. $\square$

The pivot finding algorithm is a query operation on an arborescence that only requires a handle to the root node. It can easily be modified to record the nodes it visited, i.e., the various states of u. These are exactly those nodes that lie on the path from the root node to the node that introduces the pivot. Combining this modified pivot finding algorithm with the DP table update algorithm, we obtain two update operations on an arborescence that only require a handle to the root node, one that enforces inclusion of a pivot, and one that forces exclusion of the same pivot. Finally, the solution extraction operation, too, only requires the root node of an evaluation tree. Therefore, these four operations are eligible to be used in conjunction with persistent arborescences.

Actually, there are two reasons to exploit persistence. First and foremost, without persistence we can only derive one subproblem P' from any subproblem P. The evaluation tree of P is simply lost if we perform the change ephemerally. For the Hamacher/Queyranne subproblem heap, however, we need to be able to derive two subproblems from each P. We obtain the following.

Theorem 10.11. *Let $w \in \mathbb{N}$ and $\varphi \in CMS_2$ be fixed. Given $k \in \mathbb{N}$ and an undirected graph $G = (V, E)$ with w designated vertices and treewidth at most w, the values of the k best solutions for φ on G can be computed in time $\mathcal{O}(|V| + k \log |V|)$.*

Proof. We compute a tree decomposition (T, b) of G of width w rooted in r in time $\mathcal{O}(|V|)$ according to Bodlaender [12]. We add a new root r' whose bag contains exactly the designated vertices of G, with r being the only child of r'. We have to add the designated vertices to some bags to obtain a valid tree decomposition with width at most $2w$. From this, we compute a shallow, binary, rich tree decomposition (T', b', ι) with width $6w - 2$ using an algorithm from Bodlaender and Hagerup [15] and Lemma 9.1. We compute a persistent evaluation tree using the 2-CM algorithm, and use the Hamacher/Queyranne framework for finding the k best solutions. Computing the first evaluation tree requires $\mathcal{O}(|V|)$ time. Using Frederickson's heap selection algorithm on the subproblem heap, we make sure that $\mathcal{O}(k)$ subproblems are created, each of which requires time $\mathcal{O}(\log |V|)$ using the persistent DP table update algorithm. □

The second reason is the sustained possibility to extract solutions. With handles to every old version of the evaluation tree, we are able to extract any solution whose value we computed.

There is not much room for parallelization in the algorithm above. Evaluating a tree decomposition node only takes constant time. In the update procedure, we need to evaluate the affected nodes bottom-up, and the depth of a binary tree decomposition with optimal width is $\Omega(\log |V|)$. However, the initial evaluation tree can be computed in parallel.

Theorem 10.12. *Let $w \in \mathbb{N}$ and $\varphi \in CMS_2$ be fixed. Given $k \in \mathbb{N}$ and a shallow, binary tree decomposition $(T = (U, F), b)$ with width w for an undirected graph G, the values of the k best solutions for φ on G can be computed in time $\mathcal{O}(k \log |U|)$ in the EREW PRAM model.*

Proof. We allocate one processor $p(u)$ for each node u of T which is responsible for evaluating u. Processor $p(u)$ waits until all processors $p(v)$ of child nodes v of u have finished. The processor of the root node of T therefore waits $\mathcal{O}(\log |G|)$ time. Only processor $p(u)$ writes solutions for node u, and only the parent u' of u reads them, satisfying restrictions of the EREW model. The algorithm to enhance a tree decomposition to a rich one can be parallelized accordingly. □

Finally, using the algorithm of Bodlaender [14] on $\mathcal{O}(|G|^{3w+4})$ processors to compute a shallow tree decomposition, we obtain the following.

Corollary 10.13. *Let $w \in \mathbb{N}$ and $\varphi \in CMS_2$ be fixed. Given $k \in \mathbb{N}$ and an undirected graph G of width w, the values of the k best solutions for φ on G can be computed in time $\mathcal{O}(k \log |U|)$ by $\mathcal{O}(|V|^{3w+4})$ processors in the CRCW PRAM model.*

11. Generalization to Hypergraphs

In this chapter, we prove that the results from the previous section hold for directed hypergraphs, which requires an excursion on hypergraph algebras and semi-homomorphisms. With the basics set, correctness for the generalization of Courcelle and Mosbah's algorithm to finding the two best solution values is trivial. We give the missing proofs for undirected graphs of bounded treewidth with a fixed amount of designated vertices, using the results on hypergraphs.

In this section, we adopt much of the following notation from Courcelle and Mosbah [35]. All hypergraphs in this section are labeled hypergraphs, so we omit the *labeled* qualifiers. After some definitions in Section 11.1, we show that the hypergraph approach is just a more general view on the same problem, and how the terms used here transfer to undirected graphs in Lemma 11.1.

11.1. Definitions

In this section, we define a hypergraph algebra whose parse trees serve as a replacement of tree decompositions, as well as semi-homomorphisms, a generalization of homomorphisms. We redefine some of the terms used in Chapter 10 so they can be applied in a hypergraph context.

11.1.1. Hypergraph Algebra

For hypergraphs, we perform a change in perspective. Given a set of basic hypergraphs and a set of operations to compose larger hypergraphs, we are interested in how a hypergraph could have been created. This can be described formally using a hypergraph algebra. A *hypergraph term* is either a constant symbol of the algebra, or a composed term. Let $(A, \leq, \tau)$ be a ranked alphabet of hyperedge labels, arbitrarily fixed for the reminder of this section. The function $\tau\colon A \to \mathbb{N}$ maps each edge label to its order. An algebra for *hypergraphs over A* contains the following symbols.

The constant symbol $\mathbf{0}$ denotes the empty, 0-interface hypergraph, i.e., it has no vertices or hyperedges. The constant symbol $\mathbf{1}$ denotes the 1-interface hypergraph with exactly one vertex, but no hyperedges. For each $a \in A$, $\mathbf{a}$ denotes the $\tau(a)$-interface hypergraph with node set $\{v_1, \ldots, v_{\tau(a)}\}$, a single hyperedge e with $\mathit{lab}(e) = a$ and $\mathit{vert}(e) = (v_i)_{i \in [\tau(a)]}$, and $\mathit{src} = \mathit{vert}(e)$.

Let term t denote the R-interface hypergraph $G = (V, E, \mathit{vert}, \mathit{lab}, \mathit{src})$ and term t' denote the R'-interface hypergraph $G' = (V', E', \mathit{vert}', \mathit{lab}', \mathit{src}')$. The term $t \oplus_{R,R'} t'$ denotes the $(R + R')$-interface hypergraph defined by

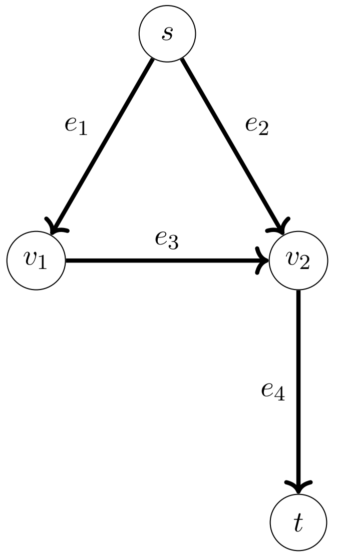
- the vertex set $(V \times \{1\}) \sqcup (V' \times \{2\})$,
- the hyperedge set $(E \times \{1\}) \sqcup (E' \times \{2\})$,
- vertex sequences $vert''$ with $vert''((e,1)) = vert(e)$ for $e \in E$ and $vert''((e,2)) = vert'(e)$ for $e \in E'$,
- labeling function lab'' with $lab''((e,1)) = lab(e)$ for $e \in E$ and $lab''((e,2)) = lab'(e)$ for $e \in E'$, and
- the $R + R'$ designated vertices $src_1, \ldots, src_R, src'_1, \ldots, src'_{R'}$.

This comes down to a disjoint union of two hypergraphs without the restrictions of the vertex or hyperedge sets being disjoint. For each $j \in [R], i \in [j-1]$, $\theta_{i,j,R}(t)$ denotes the R-interface hypergraph obtained from G by removing src_j from the vertex set of G and replacing every occurrence of src_j with src_i in the source sequence of G and in every vertex sequence of a hyperedge of G. This is equivalent to fusing src_j into src_i. For a mapping $\alpha\colon [p] \to [R]$, $\sigma_\alpha(t)$ denotes the p-interface hypergraph obtained from G by replacing its R-element source sequence src with the p-element sequence src', with $src'_i = src_{\alpha(i)}$.

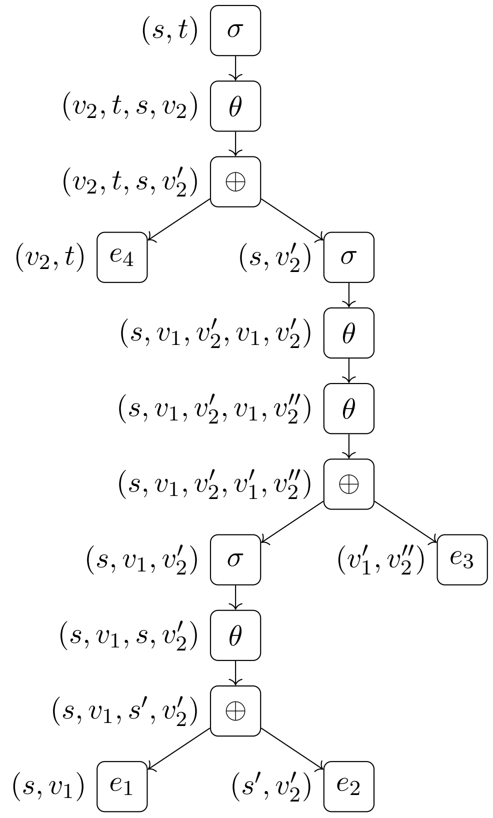
For a hypergraph term t, the hypergraph represented by t is denoted by $v(t)$. A hypergraph G is *generated* by a graph algebra H if there is a term t over H with $v(t) = G$. Every hypergraph over A can be generated by a combination of the constant symbols and hypergraph operators above. However, it consists of an infinite number of hypergraph operators. A family L of hypergraphs over A is *finitely generated* if there is a finite set $O_U = \{\oplus_{R,R'} \mid R, R' \in \mathbb{N}\}$ of disjoint union symbols, a finite set $O_F \subseteq \{\theta_{i,j,R} \mid R \in \mathbb{N}, i \in [R], j \in [R] - i\}$ of fusion symbols and a finite set $O_R = \{\sigma_\alpha \mid p \in \mathbb{N}, \alpha\colon [p] \to [R] \text{ injective}\}$ of relabeling symbols such that every hypergraph in L can be generated by the algebra $\langle \mathbf{0}, \mathbf{1}, \{\mathbf{a} \mid a \in A\}, O_U, O_F, O_R\rangle$.

We say that t is a *subterm* of $t \oplus_{R,R'} t'$, $t' \oplus_{R',R} t$, $\sigma_\alpha(t)$ and $\theta_{i,j,R}(t)$. The subterm relation is transitive, i.e., subterms of subterms are again subterms, and symmetric. The *order* of t is R if $v(t)$ is an R-interface hypergraph. The *depth* of the terms $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{a}$ for $a \in A$ is 1, while the depth of any other term is the maximum depth of any proper subterm plus one. The *width* of t is the maximum order of a subterm of t. The *width* of a hypergraph G is the minimum width of a term t with $v(t) = G$. Every family L of hypergraphs of bounded width over A is finitely generated [4, 7, 32]. We denote by $\mathcal{G}_R = \mathcal{G}_R(A)$ the set of R-interface hypergraphs over A.

Our algorithms work on parse trees of hypergraphs in $\mathcal{G}_R(A)$ with respect to some finite hypergraph algebra H. A *parse tree* $T = (U, F)$ is an arborescence with root node r that represents a term t generated by H. Every node u of T is associated with a symbol in H, called the *type* of u. If $t = t_1 \oplus_{R,R'} t_2$, then r is associated with $\oplus_{R,R'}$ and has two child nodes representing t_1 and t_2. If $t = \sigma_\alpha(t')$ or $t = \theta_{i,j,R}(t')$, then r is associated with σ_α or $\theta_{i,j,R}$, respectively, and has one child node representing t'. If t is a constant symbol, r is associated with that symbol and a leaf of the parse tree. Note that the depth of t coincides with the depth of T. For $u \in U$, $G(u)$ denotes the hypergraph represented by $T(u)$. Figure 11.1 shows an example hypergraph and its parse tree.



(a) Hypergraph G, designated vertices (s, t)



(b) Parse tree of G

Figure 11.1.: A 2-interface (hyper)graph $G \in \mathcal{G}_2$, with designated vertices s and t. The vertices on the left of each parse tree node u are the sequence of designated vertices of $G(u)$. Every edge is labeled with the only label ε with order 2.

We give the missing proofs from Chapter 10 by proving equivalent results on parse trees and then transferring them to undirected graphs. To do this, we give a transformation from a rich binary tree decomposition $(T = (U, F), b, \iota)$ of $G = (V, E)$ with width bounded by w to a hypergraph term over the above hypergraph algebra by replacing every node $u \in U$ with a parse subtree of depth, width and size $\mathcal{O}(w)$ for a hypergraph term of bounded width. We do not require distinct edge labels, so we use the alphabet $A = \{a\}$. We give constructions for terms that *represent* subgraphs or single features of G.

We first construct a hypergraph term t that represents the subgraph of G induced by the bag $b(r)$ of the root r of T that only contains the edges $\iota^{-1}(r)$ introduced by r. For $|\iota^{-1}(r)| = 1$, we have $t = \mathbf{a}$. The two vertices of the graph represented by this term are the two endpoints of the unique edge introduced by r. Otherwise, assume that $n = |\iota^{-1}|$ is divisible by two. We produce two terms t_1, t_2 recursively that contain $\frac{n}{2}$ edges of ι^{-1} each, partitioning ι^{-1} so no edge is produced twice. As the base case, we need to construct subterms for single edges again, which is already described for the case $|\iota^{-1}(r)| = 1$. Now some vertices in $v(t_1)$ represent vertices of G that are also represented in $v(t_2)$. We disjointly merge the two graphs and then fuse those vertices of $v(t_1 \oplus t_2)$ that represent the same vertices of G. Finally, we use the designated vertex relabeling operation to remove duplicate designated vertices. This results in the term $\sigma\theta \cdots \theta(t_1 \oplus t_2)$, with at most $|\iota^{-1}(r)|$ many θ operators.

Let t_{w_1}, t_{w_2} be terms that represent $G(w_1), G(w_2)$ for child nodes w_1, w_2 of r. Then $G = G(r)$ is represented by the term $\sigma\theta \cdots \theta((\sigma\theta \cdots \theta(t_{w_1} \oplus t_{w_2})) \oplus t)$. The inner parentheses represent $G(r)$ without the edges introduced by r and without vertices in $b(r)$ that are neither in $b(w_1)$ nor in $b(w_2)$. Those missing features are covered by t. The σ operator and the θ operators in the inner parentheses make sure that the width of the left operand of the last $\oplus$ is at most w. Those same operators at top-level (on the left side of the term) impose a bound on the width of the whole term.

Finally, for every vertex $v \in b(r)$ that is isolated in $G(r)$, we append $\oplus\mathbf{1}$. The $\mathbf{1}$ term then represents v.

We proceed recursively, and obtain term t_u for every node u of the tree decomposition. By replacing u in the tree decomposition with a parse tree of t_u, we obtain a parse tree for a directed version of G.

Lemma 11.1. *Let $w \in \mathbb{N}$. For every undirected graph $G = (V, E)$ of bounded width w and every binary tree decomposition $(T = (U, F), b)$ of G with width w, there is a hypergraph term t such that*

- *G can be obtained from $v(t)$ by replacing every directed edge with an undirected one on the same two vertices,*
- *the width of t is bounded by $2w$,*
- *the length of t is $\mathcal{O}(|V|)$, and*
- *the depth of t is linearly bounded in the depth of T.*

Proof. The θ operators make sure that several versions of the same vertex of G produced in subterms are fused together. Every edge is produced exactly once in the subterm of the node that introduces it.

The only operators that increase the order of a term are the $\oplus$ operators. However, the operands of $\oplus$ are either $\mathbf{a}$ or start with σ and a series of θ operators, making sure that the width of both subterm is at most $\max\{2, w\}$. The width of the resulting term is clearly bounded by $2w$.

Let $u \in U$ and $n = |\iota^{-1}(u)|$. We show by induction that the length of the subterm produced for the edges introduced by u is bounded by $s(n) := (3 + w)n - w - 2$ if there are no isolated vertices in $G(u)$. If u introduces exactly one edge, the term is $\mathbf{a}$ of length $(3+w)\cdot 1 - w - 2 = 1$. Otherwise, the term is $\sigma\theta \cdots \theta(t_1 \oplus t_2)$. It has one σ operator, one $\oplus$ operators, w many θ operators, and the length of t_1, t_2 is assumed to be bounded by $s(\frac{n}{2})$. Then the length of the overall term is $2 + w + 2s(\frac{n}{2}) = 2 + w + 2((3+w)\frac{n}{2} - w - 2) = (3 + w)n - w - 2$.

A node can only introduce $\frac{w(w-1)}{2} \in \mathcal{O}(w^2)$ many edges. For every $u \in U$, there are at most w vertices that cause an added $\oplus\mathbf{1}$ term. Merging the subterms of the child nodes of u and the subterm for the edges introduced by u adds another $2w + 4$ operators. In summary, the length of t is bounded by $s(w^2) + (2w) + (2w + 4) \in \mathcal{O}(w^2)$. Since w is a constant, the concatenation of all subterms has length $\mathcal{O}(|V|)$.

Finally, the depth of t for node u as above is bounded by its length. Replacing u in T with a parse tree of t therefore does not change its depth asymptotically. □

For fixed w, there are only finitely many subterms distinct t_u that can result from the above transformation. Instead of the definition of the type of a tree decomposition node in Chapter 10, we can define the type of u to be t_u. Both type definitions only depend on the relation between the designated vertices of u and its child nodes, as well as the pairs of designated source vertices for which u introduced an edge, and are therefore equivalent.

Sometimes it is convenient to have exactly two child nodes for each inner node of the parse tree. To achieve this, we can use a slightly modified hypergraph grammar. We can think of σ_α operators as having a second operand that is always the empty hypergraph. Likewise, $\theta_{i,j,R}$ operators can be viewed as having a second operand that is always the one-vertex hypergraph. These modified operators can even be derived from the original ones. Instead of σ_α, we can use $\sigma'_\alpha = \sigma_\alpha \circ \oplus_{0,R}(\mathbf{0})$. The operator $\theta_{i,j,R}$ can be replaced by the composition $\theta'_{i,j,R} = \theta_{i,j,R} \circ \sigma_\alpha \circ \theta_{i,R+1,R+1} \circ \oplus_{1,R}(\mathbf{1})$ with $\sigma\colon [R] \to [R+1]$, $\sigma(i) = i$. We call the resulting algebra the *binary hypergraph algebra*. Parse trees of hypergraphs with respect to the binary hypergraph algebra are called *full parse trees*. The *proper child* of a θ' or σ' node is the one that does not represent the one-vertex or empty hypergraph, respectively.

Let $q \in \mathbb{N}$ be fixed. To express problems on R-interface hypergraphs, $R\mathbb{N}$, over A in formulas, we have to represent hypergraphs in relational structures. In Chapter 10, we used structures over the signature $\{\mathbf{s}_1, \dots, \mathbf{s}_R, \mathbf{isIncident}\} \cup \{\mathbf{Card}_{m,p} \mid m, p \in \mathbb{N}, m > p\}$. The binary incidence relation symbol does not suffice anymore. Instead, we use one incidence relation symbol edg_a of arity $\tau(a)$ for every $a \in A$. Specifically, we use the signature $S^q_{A,R} = \{\mathbf{s}_1, \dots, \mathbf{s}_R\} \cup \{\mathit{edg}_a \mid a \in A\} \cup \{\mathbf{Card}_{m,p} \mid p \in [q], m \in [p]\}$. A hypergraph $G = (V, E, \mathit{vert}, \mathit{lab}, \mathit{src})$ over A can be represented as a relational structure $\langle V \sqcup E, S^q_{A,R}, \mathcal{I}\rangle$ with $\mathcal{I}(\mathbf{s}_i)$ and $\mathcal{I}(\mathbf{Card}_{m,p})$ as above, and $\mathcal{I}(\mathit{edg}_a) = \{(e, v_1, \dots, v_n) \in E \times V^n \mid \mathit{lab}(e) = a \wedge \tau(a) = n \wedge \mathit{vert}(e) = (v_1, \dots, v_n)\}$.

The finite set $\Phi_R = \Phi^{h,q}_{A,R}$ consists of the set of formulas with signature $S^q_{A,R}$ with depth of nested quantification at most $h \in \mathbb{N}$. Let $\psi \in \Phi_R$, and let $\mathbf{X}$ be a free variable of ψ. Further, let $\langle V \sqcup E, S^q_{A,R}, \mathcal{I}\rangle$ be a model for ψ with $X = \mathcal{I}(\mathbf{X})$. We assume that ψ enforces a *type* on $\mathbf{X}$, $\mathit{type}(\mathbf{X}) \subseteq \{V, E\}$, such that $X \subseteq \mathit{type}(\mathbf{X})$ for every such model. In other words, we can deduce from a formula whether a free variable is assumed to contain vertices or hyperedges, and no free variable is intended to represent mixed vertex and hyperedge sets.

We consider a combinatorial optimization problem P on hypergraphs in $\mathcal{G}_R(A)$ whose set of feasible solutions is characterized by a formula $\varphi \in \Phi_R$. In contrast to Chapter 10, we do not restrict φ to one free edge set variable. Instead, φ has a sequence $\mathbf{X}_1, \dots, \mathbf{X}_n$ of free variables. The number and types of the free variables only depends on the problem, but not on the input graph. For each model $\langle V \sqcup E, S^q_{A,R}, \mathcal{I}\rangle$, the tuple $(\mathcal{I}(\mathbf{X}_1), \dots, \mathcal{I}(\mathbf{X}_n))$ is a feasible solution for the problem characterized by φ. We extend the function $\boldsymbol{sat}$, so it maps hypergraph $G \in \mathcal{G}_R(A)$ and formula $\varphi \in \Phi_R$ to the set of feasible solutions for φ on G. Additionally, we denote by $\mathcal{U}(G, \varphi)$ the set of all n-tuples where the i-th element is a subset of $\mathit{type}(\mathbf{X}_i)$. In other words, $\mathcal{U}(G, \varphi)$ is the set of all solutions for P on G, both feasible and infeasible, and we always have $\boldsymbol{sat}(G, \varphi) \subseteq \mathcal{U}(G, \varphi)$. Please note that a solution is a tuple of sets of vertices or hyperedges in this section, while a solution in Chapter 10 was just a set of undirected edges. Two solutions X, X', are distinct if there is $j \in [n]$ with $X_j \neq X'_j$. Therefore, a solution (X_1, X_2) is also distinct from a solution (X_2, X_1) for $X_1 \neq X_2$, even if φ is a formula on two free variables, and symmetric on these free variables. For a parse tree node $u \in U$, $X(u)$ denotes the *subsolution of X at u*, which is obtained from X be removing every graph feature from every set X_i that is not present in $G(u)$. The special solution $\boldsymbol{\emptyset}$ is the n-tuple $(\emptyset, \dots, \emptyset)$.

11.1.2. Semi-Homomorphisms

We use evaluation structures to express values of solution sets. An *evaluation structure* is an algebra $\mathcal{M} = \langle M, \oplus, \otimes, \mathbf{0}, \mathbf{1}\rangle$ such that $\langle M, \oplus, \mathbf{f}\rangle$ and $\langle M, \otimes, \mathbf{1}\rangle$ are monoids, i.e., $\oplus\colon M \times M \to M$ and $\otimes\colon M \times M \to M$ are associative, $m \oplus \mathbf{0} = \mathbf{0} \oplus m = m$ and $m \otimes \mathbf{1} = \mathbf{1} \otimes m = m$ for each $m \in M$. A *semi-homomorphism* w.r.t. structure $\langle \mathcal{U}(G, \varphi), \uplus, \sqcup, \boldsymbol{\emptyset}, \emptyset\rangle$ and $\mathcal{M}$ is a mapping $h\colon \mathcal{U}(G, \varphi) \to M$ that acts like a homomorphism where applicable, i.e., $h(\boldsymbol{\emptyset}) = \mathbf{0}$, $h(\emptyset) = \mathbf{1}$, and for $A, B \in \mathcal{U}(G, \varphi)$, $h(A \uplus B) = h(A) \oplus h(B)$ if A, B do not interfere and $h(A \sqcup B) = h(A) \otimes h(B)$ if $A \cap B = \emptyset$. An *evaluation* v is a function that maps hypergraphs in $\mathcal{G}_R(A)$ to an evaluation structure $\mathcal{M}$. An evaluation v is an *MS-evaluation* if there exists a semi-homomorphism h and a CMS formula $\varphi \in \Phi_R$ such that $v(G) = h(\boldsymbol{sat}(G, \varphi))$ for every hypergraph G.

A *CMS$_2$ problem P on hypergraphs* is a combinatorial optimization problem whose set of feasible solution on each labeled R-interface hypergraph G is $\boldsymbol{sat}(G, \varphi)$ for some $\varphi \in \Phi_R$. Let $\mathbf{X}_1, \dots, \mathbf{X}_n$ be the free variables of φ. The input of P consists of a labeled R-interface hypergraph G, and n cost functions $c_i\colon \mathit{type}(\mathbf{X}_i) \to M$ for an evaluation structure $\mathcal{M}$ as above and a linearly ordered M. The value $c(X)$ of a solution $X \in \mathcal{U}(G, \varphi)$ for P is the sum of the costs of its sets, $c(X) = \sum_{i \in [n]} c_i(X_i)$. Finding the value of an optimal solution for P on G is equivalent to the evaluation $v(G) = \min\{c(X) \mid X \in \boldsymbol{sat}(G, \varphi)\}$. Since c is linear, v can be written as $v(G) = h(\boldsymbol{sat}(G, \varphi))$ with $h(A) = \min\{c(Y) \mid Y \in A\}$, and is therefore an MS-evaluation.

11.2. k-Best Enumeration on Hypergraphs

The basic ideas for k-best enumeration on hypergraphs is the same as for undirected graphs, using parse trees instead of tree decompositions. In this section, we give the required proofs and eventually conclude Chapter 10 by transferring those proofs to the undirected case.

Lemmas 2.4 to 2.6 from Courcelle and Mosbah [35] are vital for our results, so we state them here for completeness.

Lemma 11.2 (Lemma 2.4 from [35])**.** *Let $R_1, R_2 \in \mathbb{N}$ and $R = R_1 + R_2$. For every $\varphi \in \Phi_R$ there is a sequence of pairs of formulas $(\psi_1^1, \psi_2^1), \ldots, (\psi_1^l, \psi_2^2) \in \Phi_{R_1} \times \Phi_{R_2}$ for some $l \in \mathbb{N}$ such that for every R_1-interface hypergraph G_1 and every R_2-interface hypergraph G_2, we have*

$$\boldsymbol{sat}(G_1 \oplus G_2, \varphi) = \bigsqcup_{k \in [l]} \boldsymbol{sat}(G_1, \psi_1^k) \uplus \boldsymbol{sat}(G_2, \psi_2^k).$$

Note that Lemma 11.2 can be applied in full parse trees directly.

Lemma 11.3 (Lemma 2.5 from [35])**.** *For every $\varphi \in \Phi_R$ and $i, j \in [k]$, $i < j$, there is a sequence of formulas $\psi^1, \ldots, \psi^l \in \Phi_R$ for some $l \in \mathbb{N}$ such that for every R-interface hypergraph G, we have*

$$\boldsymbol{sat}(\theta_{i,j,R}(G), \varphi) = \bigsqcup_{k \in [l]} \mathcal{V}_{i,k} \uplus \boldsymbol{sat}(G, \psi^k),$$

where each $\mathcal{V}_{i,k}$ is a set consisting of exactly one solution, each set of which either is empty or only contains the i-th designated vertex of G.

Lemma 11.4 (Lemma 2.6 from [35])**.** *For every $\varphi \in \Phi_R$, $p \in \mathbb{N}$ and $\alpha \colon [R] \to [p]$, there is a formula $\psi \in \Phi_p$ such that for every p-interface hypergraph G, we have*

$$\boldsymbol{sat}(\sigma_\alpha(G), \varphi) = \boldsymbol{sat}(G, \psi).$$

Lemmas 11.3 and 11.4 cannot be applied in full parse trees directly, because we use the derived binary operators θ' and σ' there. However, there are similar decompositions of Ψ_R for our derived operators as well, which follows from Lemma 1.2 from Courcelle and Mosbah [35]. Therefore, we can use the following unified lemma for all operators of our hypergraph algebra.

Lemma 11.5. *Let $R_1, R_2 \in \mathbb{N}$ and $R = R_1 + R_2$. For every $\varphi \in \Phi_R$ and every operator f of the binary hypergraph algebra, there is a sequence of pairs of formulas $(\psi_1^1, \psi_2^1), \ldots, (\psi_1^l, \psi_2^2) \in \Phi_{R_1} \times \Phi_{R_2}$ for some $l \in \mathbb{N}$ such that for every R_1-interface hypergraph G_1 and every R_2-interface hypergraph G_2, we have*

$$\boldsymbol{sat}(f(G_1, G_2), \varphi) = \bigsqcup_{k \in [l]} \boldsymbol{sat}(G_1, \psi_1^k) \uplus \boldsymbol{sat}(G_2, \psi_2^k). \tag{11.1}$$

In the situation of Lemma 11.5, we call ψ_i^k a *child formula* of φ with respect to the corresponding operator. For $k \in [l]$, we call (ψ_1^k, ψ_2^k) a *pair of child formulas.*

Example 7. Courcelle's famous theorem about recognizability of graph properties that are expressible by CMS_2 formulas with no free variables can be expressed as an MS-evaluation. The evaluation structure to use is $\langle \mathcal{B}, \wedge, \vee, \mathbf{true}, \mathbf{false} \rangle$. The function $h \colon \mathcal{U}(G, \varphi) \to \mathcal{B}$ with $h(A) = (|A| > 0)$ applied to $\boldsymbol{sat}(G, \varphi)$ tells whether there is at least one feasible solution for φ. Since φ lacks free variables, $\boldsymbol{sat}(G, \varphi)$ can either be $\emptyset$ or $\{\emptyset\}$, the latter meaning that G has the desired property. Note that $A \uplus B = \emptyset$ iff $A = \emptyset$ or $B = \emptyset$, and therefore $h(A \uplus B) = h(A) \wedge h(B)$. Further, $A \sqcup B = \emptyset$ iff $A = \emptyset$ and $B = \emptyset$, so $h(A \sqcup B) = h(A) \vee h(B)$. Both Feferman and Vaught [45] and Courcelle [33] proved that this special MS-evaluation can be computed bottom-up on parse trees of any finitely generated family of hypergraphs. This result is usually referred to as Courcelle's theorem. So in a sense, Lemma 11.5 is a generalization of Courcelle's theorem.

Example 8. In combinatorial minimization with a linear cost function c, the cost of a solution $c(X \sqcup Y)$ can be computed as $c(X) + c(Y)$ by the definition of linearity. The same holds for solutions that comprise multiple sets, so linearity of the cost function defined for CMS_2 problems follows from the linearity of the separate cost functions. The cost of an optimal solution in a set $A \sqcup B$ is the minimum of a and b, where a is the cost of an optimal solution in A and b is the cost of an optimal solution in B. Therefore, using the MS-evaluation defined via the semi-homomorphism h to $\langle \mathbb{R} \sqcup \{\infty\}, +, \min, 0, \infty \rangle$ with $h(A) = \min\{c(X) \mid X \in A\}$, we can adapt Lemma 10.5 to hypergraphs and then apply it to all CMS_2 problems on hypergraphs. We can apply $+$ and min on two scalars in constant time, so the value of optimal solutions for CMS_2 problems can be computed in time $\mathcal{O}(|t|)$, where t is the relevant hypergraph term.

A semi-homomorphism h to $\langle (\mathbb{R} \sqcup \{\infty\})^k, +_k, \min_k, (0, \ldots, 0), (\infty, \ldots, \infty) \rangle$ for fixed $k \in \mathbb{N}$ is suitable, where $h(A)$ is the lexicographically minimal $(x_1, \ldots, x_k)$ with pairwise distinct $X_1, \ldots, X_k \in A$ and $x_i = c(X_i)$ for $i \in [k]$. Again, we can apply $+_k$ and $\min_k$ on two k-tuples of scalars in constant time. Therefore, the values of the k best solutions of a CMS_2 for any fixed k can be computed in time $\mathcal{O}(|t|)$.

Let T be a full parse tree, and $\mathcal{R} = \langle R, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$ be an evaluation structure. Consider an MS-evaluation defined by the semi-homomorphism h. Subsolutions at leaf nodes consist of empty sets or singletons, so applying h on those solutions can be done in constant time. For inner node u, we may use evaluations of subformulas at child node w_1, w_2 of u; this is precisely the reason for considering semi-homomorphisms after all. Applying h directly to Equation (11.1) yields $h(\boldsymbol{sat}(G(u), \varphi) = \bigotimes_{k \in [l]} h(\boldsymbol{sat}(G(w_1), \psi_1^k)) \oplus h(\boldsymbol{sat}(G(w_2), \psi_2^k))$. Therefore, to evaluate u, we need to apply the $\oplus$ and $\otimes$ operator k times for every formula $\varphi \in \Phi_R$, where k only depends on the type of u and φ, and therefore does not depend on the size of the graph.

Lemma 11.6 (Variant of Proposition 3.1 from [35])**.** *For every MS-evaluation f there is an algorithm that computes $f(v(t))$ for every hypergraph term t of bounded width in time $\mathcal{O}(\eta|t|)$, where η is an upper time bound for computing $a \otimes b$ or $a \oplus b$.*

The operators in Examples 7 and 8, $\wedge$, $\vee$, $+$, min, $+_k$ and $\min_k$ (for fixed k) can all be applied in constant time. In conjunction with Bodlaender's algorithm [12] to compute a tree decomposition graphs with bounded treewidth in linear time, Lemma 9.1 and Lemma 11.1, this completes all proofs of Section 10.1. Also, we did not restrict CMS_2 problems to formulas with only one free set variable. All results above also hold for multiple free variables, not necessarily of the same type. Therefore, we can compute the values of the two best solutions (actually, the k best solutions for fixed k) for any CMS_2 problem on hypergraphs in linear time. This result extends naturally to undirected graphs with bounded treewidth via Lemma 11.1.

The last pending proof is the one for Lemma 10.3. This theorem was used by the pivot finding algorithm. We also used it to show that only DP tables on one path of an evaluation tree have to be updated to instantiate a subproblem.

Lemma 11.7. *For every R-interface hypergraph G, operator f of the binary hypergraph algebra, $\varphi \in \Phi_R$ with pairs of child formulas $(\psi_1^1, \psi_2^1), \ldots, (\psi_1^l, \psi_2^l)$ w.r.t. f and $i, j \in [l]$, the solution sets $\boldsymbol{sat}(G_1, \psi_1^i)$ and $\boldsymbol{sat}(G_2, \psi_2^j)$ do not interfere.*

Proof. For the $\oplus$ operator, G_1 and G_2 have disjoint vertex and hyperedge sets, so no feature can occur both in a solution for G_1 and in a solution for G_2. The same is true for the σ' operator: its non-proper operand is $\mathbf{0}$, and the only possible solution for any formula on $v(\mathbf{0})$ is $\boldsymbol{\emptyset}$. The non-proper operand of θ' is $\mathbf{1}$, and possible solutions for $v(\mathbf{1})$ are exactly the elements of the $\mathcal{V}_{i,k}$ from Lemma 11.3, i.e., tuples consisting of empty sets and 1-element sets containing the i-th designated vertex of the hypergraph. It is not immediately clear from Lemma 11.3 that solutions for the child formulas ψ^i cannot contain the i-th designated vertex, but this follows directly from the construction of child formulas in the proof of Lemma 2.5 in [35]. $\square$

Reusing the terminology from Chapter 10 again, a hyperedge e with $lab(e) = a$ is introduced by the parse tree leaf with type $\mathbf{a}$ that corresponds to e. A vertex v is introduced by the lowest common ancestor of type θ' of all parse tree nodes u for which $G(u)$ contains a vertex that represents u. If there is no such θ' node, there is a unique parse tree leaf of type $\mathbf{1}$ that represents v, or of type $\mathbf{a}$ such that v is represented by a one of the vertices of the produced hyperedge, so v is introduced by that leaf.

Corollary 11.8. *Let T be the parse tree with root r of a term over the binary hypergraph algebra. In terms of Lemma 11.5, if $x \in X$ for a hypergraph feature x and a set X of any solution in $\boldsymbol{sat}(G(w_i), \psi_i^k)$, then x is introduced by a parse tree node in $T(w_i)$.*

Proof. This is a direct consequence of Lemma 11.7. $\square$

The connection between nodes of a tree decomposition introducing an edge e and the parse tree node that introduces a directed counterpart of e constitutes Lemma 10.3.

Corollary 11.9. *If an edge e is introduced by node $u \in U$ of a rich binary tree decomposition $(T = (U, F), b, \iota)$, then it is also introduced by a node in the subtree t_u.*

Proof. We explicitly created $\mathbf{a}$ nodes in t_u for all the edges in $\iota^{-1}(u)$. $\square$

Lemma 11.5 and the concept of semi-homomorphisms allow us to generalize the 2-CM algorithm to parse trees. The idea of solution IDs is easily generalized for solutions with multiple sets, too. The pivot finding algorithm, and the update procedure for evaluation trees only required Corollary 10.4, which we transfered to parse trees in Corollary 11.8. Thus, we can use the Hamacher/Queyranne solution space partition technique in conjunction with persistent evaluation trees again, which establishes the main theorem of this section.

Theorem 11.10. *Let L be a family of finitely generated hypergraphs, and $\varphi \in CMS_2$. Given $k \in \mathbb{N}$ and a parse tree $T = (U, F)$ with bounded width of a hypergraph $G \in L$, the values of the k best solutions satisfying φ on G can be computed in time $\mathcal{O}(|U| + k \log |U|)$.*

Part IV.

Epilogue

12. Conclusion

In this thesis, we consider k-best variants of combinatorial optimization problems on graphs. In particular, the first part studies about the k shortest simple path problem in directed, weighted graphs. The importance of this problem has been established by a variety of research and a great number of algorithms that have already been proposed for it. However, all those algorithms are derived from the first kSSP algorithm by Yen [105] that upper-bounded the problem's complexity by $\mathcal{O}(kn(m + n \log n))$. Improvements were only proposed for the solution of a subproblem, the problem of finding restricted replacement paths, while leaving the algorithm's frame unchanged. In contrast, our algorithm is the first to be based on Eppstein's algorithm for the problem of finding the k shortest paths in a directed, weighted graph, omitting the requirement for the paths to be simple. This problem, kSP, is seemingly less complex, with Eppstein's algorithm [40] having a running time of $\mathcal{O}(m + n \log n + k)$ if the k shortest paths are not required to be output in increasing order of length, and an additional $\mathcal{O}(k \log k)$ otherwise. We describe how to bridge this gap in complexity to achieve a running time equal to that of Yen's algorithm. Furthermore, we propose faster reachability checks and online edge pruning techiques to further speed up the algorithm in practice.

We provide an extensive computational study to compare the proposed algorithm with state-of-the-art Yen-based algorithms, namely Sedeño-Noda's algorithm [96] and Feng's algorithm [46]. We confirm the superiority of Feng's algorithm over Sedeño-Noda's algorithm on all considered graph classes. More importantly, our algorithm further significantly improves on the running times of Feng's algorithm, often by an order of magnitude. This is especially true for the full-improvements implementation of our algorithm for sufficiently large path counts.

In the second part of this thesis, we widen our focus to all combinatorial optimization problems on directed hypergraphs with a fixed set of node and hyperedge labels that can be expressed in monadic second-order logic with counting. More precisely, we consider optimization problems where feasible solutions can be characterized by a formula in this logic. We require that input hypergraphs are generated by a fixed and finite set of production rules. If the algebra only produces hyperedges with exactly two vertices, the resulting hypergraphs are effectively graphs with bounded treewidth.

For such hypergraphs, Courcelle's theorem [33] states that graph properties expressible in counting monadic second-order logic can be recognized in linear time; this was generalized to said combinatorial optimization problems by Courcelle and Mosbah [35] shortly after. We generalize their approach to k-best optimization using Hamacher/Queyranne-style subproblem generation [59]. We introduce a hypergraph algebra with binary operators only, therefore obtaining binary parse trees, without losing expressive power when compared with the hypergraph algebra used by Courcelle and Mosbah. After the initial evaluation of the formula that characterizes feasible solutions on the whole parse tree, we use techniques based on persistent data structures to be able to derive evaluation trees of subproblems. The resulting algorithm matches that of Courcelle and Mosbah in running time $\mathcal{O}(n)$ required to find the initial solution. We compute the value of subsequent solutions in logarithmic time each, resulting in a total running time of $\mathcal{O}(n + \log k)$ for finding the values of the k best solutions. Further, each solution can be extracted from the algorithm's main data structure in linear time in the input size.

13. Open Problems

We provide a list of open problems that we encountered during our research on k-best enumeration, for paths or in general, and that we were not able to answer conclusively.

13.1. Simple Path Enumeration

Our sidetrack-based algorithm for simple path enumeration with edge pruning is already significantly faster than other state-of-the-art algorithms. However, we believe that there is still room for practical improvements.

First and foremost, we do not use the full potential of the candidate heap. Recall that the root of this heap represents the empty sidetrack sequence. Also recall that candidates on a path in the candidate heap starting in the root are represented by sidetrack sequences of non-decreasing length. At any given time, our algorithm only explores a region of the candidate heap where all candidates have the same length. After extracting a candidate represented by $(e_1, \ldots, e_r)$, we discover a candidate $(e_1, \ldots, e_r, e)$ for each sidetrack e whose tail is on the path from the head of e_r to t. Some of these extensions represent simple paths, while others do not.

Our aim is to explore larger parts of the candidate heap early. Right before we explore the abovementioned part of the candidate heap, we perform a depth-first search that lets us determine whether the above trivial extensions are simple or not in constant time. For those trivial extensions that are again simple, we can simply repeat the process. Assume that the trivial extension $C = (e_1, \ldots, e_r, e)$ was determined to be simple. We can perform another depth-first search and in turn discover all trivial extensions of C. If we repeat this process recursively, we explore a large cycle-free part of the candidate heap, spanning various sidetrack sequence length. Since we usually find many new simple candidates, we need a means to prioritize them. We may use another priority queue where candidates' priorities are again their length. Note that candidate discovery has to stop at non-simple candidates, since those require repair including a Dijkstra run. Running Dijkstra's algorithm on the whole graph is significantly slower than a depth-first search on a spanning tree of the graph, which itself is not asymptotically slower than collecting the sidetracks of a path suffix.

The advantage of this greedy candidate heap exploration approach is an early upper bound on the cost of the k-th simple path. As soon as k simple candidates have been found, we obtain our first upper bound, and finding k simple candidates fast is the whole point of this improvement. Recall that this upper bound can be used to prune edges and therefore to speedup Dijkstra runs. It is evident from our experiments that this edge pruning yields a great overall speedup for our sidetrack-based algorithm. As a side effect, we can also limit the size of the non-candidate set earlier.

Question: Does the sidetrack-based kSSP algorithm benefit from an early upper bound found by a more greedy exploration of the candidate heap?

Another potential for improvement is parallelization. The cycle-free area of the candidate heap described above is delimited by several non-simple candidates. Assume that C_1 and C_2 are two such candidates, and the non-simple path represented by C_1 is shorter than the non-simple path represented by C_2. The former is thus extracted earlier from the non-candidate set, and we compute a new SP tree for the corresponding sub problem. However, we do not obtain any information about the subproblem induced by C_2 when acting on C_1. Therefore, we can trivially parallelize the two Dijkstra runs. For example, we could allocate $z-1$ threads, z being the number of processor cores, each of which computes the SP tree for a distinct subproblem. Since Dijkstra runs are offloaded to their own threads, we do not have to wait for the first non-simple candidate to be extracted before we start the first Dijkstra run. Processing a simple candidate is several orders of magnitude faster than a Dijkstra run on the whole graph, so we will not be able to avoid blocking when the first non-simple candidate is extracted. However, subsequent blocking times might very well be evitable.

Parallelization can also be used in conjunction with the cycle-free candidate heap exploration approach above. Several threads can perform depth-first searches to advance the cycle-free area for different simple trivial extensions, thus boosting the whole exploration process.

Question: Does the sidetrack-based kSSP algorithm benefit from parallelization of Dijkstra runs or a parallelized greedy cycle-free candidate heap exploration?

Open questions for the more theoretically inclined reader regard the running time required by a kSSP algorithm *per path.* All existing algorithms for the problem require an extra time of $\mathcal{O}(n(m+n\log n))$ for every additionally requested simple path in the worst case, resulting in $\mathcal{O}(kn(m+n\log n))$ total time for k simple paths. The $\mathcal{O}(m+n\log n)$ part is caused by the Dijktra algorithm.

Most interesting would be a split into a preprocessing phase whose running time only depends (polynomially) on the size of the graph, and a query phase. The k simple paths would be enumerated in the query phase, so its running time has to depend on k.

Question: Is it possible to solve kSSP in time $p(n) + o(kn(m + n\log n))$, where p is a polynomial function?

In Part III, we demonstrated that kSSP can be solved in time $\mathcal{O}(n + k\log n)$ if the treewidth of the input graph is bounded. Tree-decomposable graphs, however, are not the only class of graphs that allow for faster algorithms. Another obvious example are directed acyclic graphs, where shortest paths can be computed in linear time. The running time of any existing kSSP algorithm is thus reduced to $\mathcal{O}(knm)$ on such graphs.

Question: Are there other interesting graph classes that allow for allow for faster shortest path algorithms, or the abovementioned preprocessing-query split, or a more efficient storage of shortest path trees, or an efficient adoption of an SP tree of the graph to an SP tree of a subgraph?

Finally, we turn our attention to opportunities to apply our algorithms in practice. Our experimental study demonstrated that the sidetrack-based approach is to be preferred if kSSP has to be solved directly. However, enumerating good solutions can also appear as a subproblem. Specifically, enumerating paths is one of the two most common approaches for the gap-closing phase of the *constrained shortest path problem* (CSP). In the past, the enumeration approach proved inferior to the alternative label propagation approach known from bicriterial path enumeration. Due to the advent of our efficient kSSP algorithm, we require a reevaluation of this relation.

Question: Should we still prefer label setting algorithms over path enumeration algorithms for the gap-closing phase of state-of-the-art CSP algorithms?

The gap-closing phase is preceded by a strong reduction test phase. A reduction test checks whether a vertex or an edge can possibly be part of an optimal solution. Any graph feature that cannot contribute to optimal solutions can simply be removed from the graph. Strong reduction tests for CSP solve Lagrangian relaxations of subproblems that force graph features to be part of the solution. The remaining graph might exhibit a structure that lead to subpar performance of existing kSSP algorithm, but does not slow down the sidetrack-based algorithm with pruning significantly. We already saw this behaviour on topology graphs in Section 8.2.4.

Question: Do subgraphs resulting from reduction test pruning in CSP show a structure that allows for further optimization of the sidetrack-based kSSP algorithm, or kSSP algorithms in general?

13.2. K-Best Enumeration on Tree-Decomposable Graphs

There is not much room for improvement on the theoretical side of solution enumeration for CMS_2 problems on tree-decomposable graphs. The optimal solution cannot be computed in sublinear time for all these problems. For example, the problem of finding the cheapest edge is an CMS_2 problem, and sabotaging an algorithm for this problem that does not consider every edge is straightforward. The enumeration phase cannot be solved faster, either. A path graph with n vertices has treewidth 1, and enumerating all solutions for the cheapest-edge problem in increasing order cannot be done in $o(n \log n)$ time, since this is equivalent to sorting.

Recall that the size of the set Φ_R of formulas we have to compute solutions for depends on the problem in consideration and on the treewidth. Grohe and Frick [52] proved that this size cannot be bounded by any elementary function in the treewidth and the length of the formula that characterizes feasible solutions unless P = NP. We already saw that the length of a formula that characterizes simple paths between two designated vertices is quite long in Section 9.1. However, there are still problems for which Φ_R is small. The minimum weight three-edge matching toy problem in Example 5 only requires $9w + 1$ subformulas, where w is the upper bound on the treewidth. The similar cheapest-edge problem (equivalently, the minimum weight one-edge matching problem) only requires two subformulas, specifically one for the original problem and one that characterizes empty solutions. Those are still toy problems.

Question: Are there interesting graph problems for which we can derive practically relevant algorithms from our solution enumeration approach?

A. Bibliography

[1] S. G. Akl. *Design and analysis of parallel algorithms.* Prentice Hall, 1989.

[2] J. Alber and R. Niedermeier. “Improved Tree Decomposition Based Algorithms for Domination-like Problems.” In: *5th Latin American Symposium on Theoretical Informatics (LATIN).* Ed. by S. Rajsbaum. *Lecture Notes in Computer Science* 2286. Springer, 2002, pp. 613–628. DOI: 10.1007/3-540-45995-2_52.

[3] M. Arita. “Metabolic reconstruction using shortest paths.” In: *Simulation Practice and Theory* 8.1-2 (2000), pp. 109–125. DOI: 10.1016/S0928-4869(00)00006-9.

[4] S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. “An algebraic theory of graph reduction.” In: *Journal of the ACM* 40.5 (1993), pp. 1134–1164. DOI: 10.1145/174147.169807.

[5] S. Arnborg, J. Lagergren, and D. Seese. “Easy problems for tree-decomposable graphs.” In: *Journal of Algorithms* 12.2 (1991), pp. 308–340. DOI: 10.1016/0196-6774(91)90006-K.

[6] S. Arnborg and A. Proskurowski. “Linear time algorithms for NP-hard problems restricted to partial k-trees.” In: *Discrete Applied Mathematics* 23.1 (1989), pp. 11–24. DOI: 10.1016/0166-218X(89)90031-0.

[7] M. Bauderon and B. Courcelle. “Graph Expressions and Graph Rewritings.” In: *Mathematical Systems Theory* 20.2-3 (1987), pp. 83–127. DOI: 10.1007/BF01692060.

[8] M. W. Bern, E. L. Lawler, and A. L. Wong. “Linear-Time Computation of Optimal Subgraphs of Decomposable Graphs.” In: *Journal of Algorithms* 8.2 (1987), pp. 216–235. DOI: 10.1016/0196-6774(87)90039-3.

[9] A. Bernstein. “A Nearly Optimal Algorithm for Approximating Replacement Paths and k Shortest Simple Paths in General Graphs.” In: *21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* Ed. by M. Charikar. SIAM, 2010, pp. 742–755. DOI: 10.1137/1.9781611973075.61.

[10] U. Bertelè and F. Brioschi. “On Non-serial Dynamic Programming.” In: *Journal of Combinatorial Theory, Series A* 14.2 (1973), pp. 137–148. DOI: 10.1016/0097-3165(73)90016-2.

[11] M. Betz and H. Hild. “Language models for a spelled letter recognizer.” In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE Computer Society, 1995, pp. 856–859. DOI: 10.1109/ICASSP.1995.479829.

[12] H. L. Bodlaender. “A linear-time algorithm for finding tree-decompositions of small treewidth.” In: *SIAM Journal on Computing* 25.6 (1996), pp. 1305–1317. DOI: 10.1137/S0097539793251219.

[13] H. L. Bodlaender. “A Partial k-Arboretum of Graphs with Bounded Treewidth.” In: *Theoretical Computer Science* 209.1-2 (1998), pp. 1–45. DOI: 10.1016/S0304-3975(97)00228-4.

[14] H. L. Bodlaender. “NC-algorithms for graphs with small treewidth.” In: *14th International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*. Ed. by J. van Leeuwen. *Lecture Notes in Computer Science* 344. Springer, 1989, pp. 1–10. DOI: 10.1007/3-540-50728-0_32.

[15] H. L. Bodlaender and T. Hagerup. “Parallel Algorithms with Optimal Speedup for Bounded Treewidth.” In: *SIAM Journal on Computing* 27.6 (1998), pp. 1725–1746. DOI: 10.1137/S0097539795289859.

[16] M. Bojańczyk and M. Pilipczuk. “Definability equals recognizability for graphs of bounded treewidth.” In: *31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. Ed. by M. Grohe, E. Koskinen, and N. Shankar. ACM, 2016, pp. 407–416. DOI: 10.1145/2933575.2934508.

[17] A. Borys. “On fading memory and asymptotic properties of steady state solutions for digital systems.” In: *International Journal of Circuit Theory and Applications* 24.5 (1996), pp. 593–596. DOI: 10.1002/(SICI)1097-007X(199609/10)24:5<593::AID-CTA937>3.0.CO;2-4.

[18] T. Briscoe and J. Carroll. “Generalized Probabilistic LR Parsing of Natural Language (Corpora) with Unification-based Grammars.” In: *Computational Linguistics* 19.1 (1993), pp. 25–59.

[19] *BRITE topology generator*. http://www.cs.bu.edu/brite/. Apr. 21, 2016.

[20] T. H. Byers and M. S. Waterman. “Technical Note - Determining All Optimal and Near-Optimal Solutions when Solving Shortest Path Problems by Dynamic Programming.” In: *Operations Research* 32.6 (1984), pp. 1381–1384. DOI: 10.1287/opre.32.6.1381.

[21] W. M. Carlyle and R. K. Wood. “Near-shortest and K-shortest simple paths.” In: *Networks* 46.2 (2005), pp. 98–109. DOI: 10.1002/net.20077.

[22] E. Charniak and M. Johnson. “Coarse-to-Fine n-Best Parsing and MaxEnt Discriminative Reranking.” In: *43rd Annual Meeting of the Association for Computational Linguistics (ACL)*. Ed. by K. Knight, H. T. Ng, and K. Oflazer. The Association for Computer Linguistics, 2005, pp. 173–180.

[23] J. Chen and F. K. Soong. “An N-best candidates-based discriminative training for speech recognition applications.” In: *IEEE Transactions on Speech and Audio Processing* 2.1 (1994), pp. 206–216. DOI: 10.1109/89.260363.

[24] Y. L. Chen. “Finding the k Quickest Simple Paths in a Network.” In: *Information Processing Letters* 50.2 (1994), pp. 89–92. DOI: 10.1016/0020-0190(94)00008-5.

[25] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel. "The Open Graph Drawing Framework (OGDF)." In: *Handbook on Graph Drawing and Visualization.* Ed. by R. Tamassia. Chapman and Hall/CRC, 2013, pp. 543–569.

[26] M. Chimani, P. Mutzel, and B. Zey. "Improved Steiner Tree Algorithms for Bounded Treewidth." In: *22nd International Workshop on Combinatorial Algorithms (IWOCA).* Ed. by C. S. Iliopoulos and W. F. Smyth. *Lecture Notes in Computer Science* 7056. Springer, 2011, pp. 374–386. DOI: `10.1007/978-3-642-25011-8_30`.

[27] W. Chou, C. H. Lee, and B. H. Juang. "Minimum error rate training based on N-best string models." In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 1993, pp. 652–655. DOI: `10.1109/ICASSP.1993.319394`.

[28] W. Chou, C. Lee, B. Juang, and F. K. Soong. "A Minimum Error Rate Pattern Recognition Approach to Speech Recognition." In: *International Journal of Pattern Recognition and Artificial Intelligence* 8.1 (1994), pp. 5–31. DOI: `10.1142/S0218001494000024`.

[29] W. Chou, T. Matsuoka, B. Juang, and C. Lee. "An algorithm of high resolution and efficient multiple string hypothesization for continuous speech recognition using inter-word models." In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE Computer Society, 1994, pp. 153–156. DOI: `10.1109/ICASSP.1994.389696`.

[30] S. Clarke, A. Krikorian, and J. Rausen. "Computing the N Best Loopless Paths in a Network." In: *Journal of the SIAM* 11.4 (1963), pp. 1096–1102.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[32] B. Courcelle. "Graph Rewriting: An Algebraic and Logic Approach." In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B).* 1990, pp. 193–242.

[33] B. Courcelle. "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs." In: *Information and Computation* 85.1 (1990), pp. 12–75. DOI: `10.1016/0890-5401(90)90043-H`.

[34] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach. Encyclopedia of mathematics and its applications* 138. Cambridge University Press, 2012.

[35] B. Courcelle and M. Mosbah. "Monadic second-order evaluations on tree-decomposable graphs." In: *Theoretical Computer Science* 109.1&2 (1993), pp. 49–82. DOI: `10.1016/0304-3975(93)90064-Z`.

[36] R. Diestel. *Graph Theory, 4th Edition. Graduate texts in mathematics* 173. Springer, Heidelberg, New York, Dordrecht, London, 2012. DOI: `10.1007/978-3-642-14279-6`.

[37] *The Ninth DIMACS Implementation Challenge: 2005-2006.* `http://www.dis.uniroma1.it/challenge9/`. Accessed: 2015-11-12.

[38] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. "Making data structures persistent." In: *J. Comput. Syst. Sci.* 38.1 (1989), pp. 86–124. DOI: `10.1016/0022-0000(89)90034-2`.

[39] M. Elberfeld, A. Jakoby, and T. Tantau. "Logspace Versions of the Theorems of Bodlaender and Courcelle." In: *51st Annual IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, 2010, pp. 143–152. DOI: `10.1109/FOCS.2010.21`.

[40] D. Eppstein. "Finding the k Shortest Paths." In: *SIAM Journal on Computing* 28.2 (1998), pp. 652–673.

[41] D. Eppstein. "K-Best Enumeration." In: *Encyclopedia of Algorithms.* Ed. by M. Kao. Springer, New York, 2015.

[42] D. Eppstein. "K-best enumeration." In: *Bull. EATCS* 115 (2015).

[43] D. Eppstein. "k-best enumeration." In: *CoRR* abs/1412.5075 (2014).

[44] D. Eppstein and D. Kurz. "K-Best Solutions of MSO Problems on Tree-Decomposable Graphs." In: *CoRR* abs/1703.02784 (2017).

[45] S. Feferman and R. L. Vaught. "The first order properties of products of algebraic systems." In: *Fund. Math.* 47 (1959), pp. 57–103.

[46] G. Feng. "Finding k shortest simple paths in directed graphs: A node classification algorithm." In: *Networks* 64.1 (2014), pp. 6–17.

[47] K. Filippova. "Multi-Sentence Compression: Finding Shortest Paths in Word Graphs." In: *23rd International Conference on Computational Linguistics (COLING).* Ed. by C. Huang and D. Jurafsky. Tsinghua University Press, 2010, pp. 322–330.

[48] S. Fiorini, N. Hardy, B. A. Reed, and A. Vetta. "Planar graph bipartization in linear time." In: *Discrete Applied Mathematics* 156.7 (2008), pp. 1175–1180. DOI: `10.1016/j.dam.2007.08.013`.

[49] J. Flum and M. Grohe. *Parameterized Complexity Theory. Texts in Theoretical Computer Science.* Springer, Berlin, Heidelberg, 2006. DOI: `10.1007/3-540-29953-X`.

[50] G. N. Frederickson. "An Optimal Algorithm for Selection in a Min-Heap." In: *Inf. Comput.* 104.2 (1993), pp. 197–214.

[51] M. L. Fredman. "On the Efficiency of Pairing Heaps and Related Data Structures." In: *J. ACM* 46.4 (1999), pp. 473–501. DOI: `10.1145/320211.320214`.

[52] M. Frick and M. Grohe. “The complexity of first-order and monadic second-order logic revisited.” In: *Annals of Pure and Applied Logic* 130.1-3 (2004), pp. 3–31. DOI: 10.1016/j.apal.2004.01.007.

[53] A. Frieder and L. Roditty. “An Experimental Study on Approximating k Shortest Simple Paths.” In: *ACM J. Exp. Algorithmics* 19.1 (2014).

[54] H. N. Gabow. “Two algorithms for generating weighted spanning trees in order.” In: *SIAM Journal on Computing* 6.1 (1977), pp. 139–150. DOI: 10.1137/0206011.

[55] Z. Gotthilf and M. Lewenstein. “Improved algorithms for the k simple shortest paths and the replacement paths problems.” In: *Information Processing Letters* 109.7 (2009), pp. 352–355.

[56] J. Gramm and R. Niedermeier. “Minimum Quartet Inconsistency Is Fixed Parameter Tractable.” In: *12th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Ed. by A. Amir and G. M. Landau. *Lecture Notes in Computer Science* 2089. Springer, 2001, pp. 241–256. DOI: 10.1007/3-540-48194-X_23.

[57] V. Guruswami. “List decoding of error correcting codes.” PhD thesis. Massachusetts Institute of Technology, 2001.

[58] E. Hadjiconstantinou, N. Christofides, and A. Mingozzi. “A new exact algorithm for the vehicle routing problem based on q-paths and k-shortest paths relaxations.” In: *Annals of Operations Research* 61.1 (1995), pp. 21–43. DOI: 10.1007/BF02098280.

[59] H. W. Hamacher and M. Queyranne. “K best solutions to combinatorial optimization problems.” In: *Annals of Operations Research* 4.1 (1985), pp. 123–143. DOI: 10.1007/BF02022039.

[60] J. Hershberger, M. Maxel, and S. Suri. “Finding the k shortest simple paths: A new algorithm and its implementation.” In: *ACM Transactions on Algorithms* 3.4 (2007).

[61] J. Hershberger and S. Suri. “Vickrey Prices and Shortest Paths: What is an Edge Worth?” In: *42nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 2001, pp. 252–259.

[62] T. Hisamitsu and Y. Nitta. “A generalized algorithm for Japanese morphological analysis and a comparative evaluation of some heuristics.” In: *Systems and Computers in Japan* 26.1 (1995), pp. 73–87. DOI: 10.1002/scj.4690260107.

[63] L. Huang and D. Chiang. “Better k-best Parsing.” In: *9th International Workshop on Parsing Technology (IWPT)*. Ed. by H. Bunt, R. Malouf, and A. Lavie. Association for Computational Linguistics, 2005, pp. 53–64.

[64] J. Iacono. “Improved Upper Bounds for Pairing Heaps.” In: *7th Scandinavian Workshop on Algorithm Theory (SWAT)*. Ed. by M. M. Halldórsson. *Lecture Notes in Computer Science* 1851. Springer, 2000, pp. 32–45. DOI: 10.1007/3-540-44985-X_5.

[65] H. Ishii. “A new method finding the K-th best path in a graph.” In: *Journal of the Operations Research Society of Japan* 21.4 (1979).

[66] W. Jin, S. Chen, and H. Jiang. “Finding the K shortest paths in a time-schedule network with constraints on arcs.” In: *Computers & Operations Research* 40.12 (2013), pp. 2975–2982. DOI: `https://doi.org/10.1016/j.cor.2013.07.005`.

[67] N. Katoh, T. Ibaraki, and H. Mine. “An efficient algorithm for K shortest simple paths.” In: *Networks* 12.4 (1982), pp. 411–427.

[68] N. Katoh, T. Ibaraki, and H. Mine. “An $O(Kn^2)$ algorithm for K shortest simple paths in an undirected graph with nonnegative arc length.” In: *Electronics and Communications in Japan* 61.12 (1978), pp. 1–8.

[69] B. P. Kelley, R. Sharan, R. M. Karp, T. Sittler, D. E. Root, B. R. Stockwell, and T. Ideker. “Conserved pathways within bacteria and yeast as revealed by global protein network alignment.” In: *Proceedings of the National Academy of Sciences* 100.20 (2003), pp. 11394–11399. DOI: `10.1073/pnas.1534710100`.

[70] J. M. Kleinberg and É. Tardos. *Algorithm design.* Addison-Wesley, 2006.

[71] K. Knight and J. Graehl. “Machine Transliteration.” In: *Computational Linguistics* 24.4 (1998), pp. 599–612.

[72] K. Knight and V. Hatzivassiloglou. “Two-Level, Many-Path Generation.” In: *33rd Annual Meeting of the Association for Computational Linguistics (ACL).* Ed. by H. Uszkoreit. Morgan Kaufmann Publishers / ACL, 1995, pp. 252–260.

[73] D. Kurz and P. Mutzel. “A Sidetrack-Based Algorithm for Finding the k Shortest Simple Paths in a Directed Graph.” In: *27th International Symposium on Algorithms and Computation (ISAAC).* Ed. by S. Hong. *Leibniz International Proceedings in Informatics (LIPIcs)* 64. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 49:1–49:13. DOI: `10.4230/LIPIcs.ISAAC.2016.49`.

[74] D. Kurz, P. Mutzel, and B. Zey. “Parameterized Algorithms for Stochastic Steiner Tree Problems.” In: *8th International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS).* Ed. by A. Kucera, T. A. Henzinger, J. Nesetril, T. Vojnar, and D. Antos. *Lecture Notes in Computer Science* 7721. Springer, 2012, pp. 143–154. DOI: `10.1007/978-3-642-36046-6_14`.

[75] D. H. Larkin, S. Sen, and R. E. Tarjan. “A Back-to-Basics Empirical Study of Priority Queues.” In: *16th Workshop on Algorithm Engineering and Experiments (ALENEX).* Ed. by C. C. McGeoch and U. Meyer. SIAM, 2014, pp. 61–72. DOI: `10.1137/1.9781611973198.7`.

[76] E. L. Lawler. “A Procedure for Computing the K Best Solutions to Discrete Optimization Problems and Its Application to the Shortest Path Problem.” In: *Management Science* 18 (1972), pp. 401–405. DOI: `10.1287/mnsc.18.7.401`.

[77] L. Libkin. *Elements of Finite Model Theory. Texts in Theoretical Computer Science.* Springer, Berlin, Heidelberg, 2004. DOI: `10.1007/978-3-662-07003-1`.

[78] E. d. Q. V. Martins and M. M. B. Pascoal. “A new implementation of Yen’s ranking loopless paths algorithm.” In: *Quarterly Journal of the Belgian, French and Italian Operations Research Societies* 1.2 (2003), pp. 121–133. DOI: `10.1007/s10288-002-0010-2`.

[79] E. d. Q. V. Martins, M. M. B. Pascoal, and J. L. E. D. Santos. “Deviation Algorithms for Ranking Shortest Paths.” In: *International Journal of Foundations of Computer Science* 10.3 (1999), pp. 247–262. DOI: `10.1142/S0129054199000186`.

[80] C. C. McGeoch. *A Guide to Experimental Algorithmics.* Cambridge University Press, 2012.

[81] S.-P. Miaou and S.-M. Chin. “Computing k-shortest path for nuclear spent fuel highway transportation.” In: *European Journal of Operational Research* 53 (1991), pp. 64–80.

[82] K. G. Murty. “Letter to the Editor – An Algorithm for Ranking all the Assignments in Order of Increasing Cost.” In: *Operations Research* 16.3 (1968), pp. 682–687. DOI: `10.1287/opre.16.3.682`.

[83] D. Naor and D. L. Brutlag. “On Near-Optimal Alignments of Biological Sequences.” In: *Journal of Computational Biology* 1.4 (1994), pp. 349–366. DOI: `10.1089/cmb.1994.1.349`.

[84] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization. Wiley interscience series in discrete mathematics and optimization.* Wiley, 1988.

[85] M. M. B. Pascoal. *Implementations and empirical comparison of K shortest loopless path algorithms.* 9th DIMACS Implementation Challenge Workshop: Shortest Paths. 2006. URL: `http://www.inescc.pt/documentos/RR2006_09.pdf`.

[86] S. Pettie. “A new approach to all-pairs shortest paths on real-weighted graphs.” In: *Theoretical Computer Science* 312.1 (2004), pp. 47–74.

[87] S. Pettie. “Towards a Final Analysis of Pairing Heaps.” In: *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS).* IEEE Computer Society, 2005, pp. 174–183. DOI: `10.1109/SFCS.2005.75`.

[88] P. A. Pevzner and S. Sze. “Combinatorial Approaches to Finding Subtle Signals in DNA Sequences.” In: *8th International Conference on Intelligent Systems for Molecular Biology (ISMB).* Ed. by P. E. Bourne, M. Gribskov, R. B. Altman, N. Jensen, D. A. Hope, T. Lengauer, J. C. Mitchell, E. D. Scheeff, C. Smith, S. Strande, and H. Weissig. AAAI, 2000, pp. 269–278.

[89] M. Pollack. “The kth best route through a network.” In: *Operations Research* 9 (1961), pp. 578–580.

[90] N. Robertson and P. D. Seymour. “Graph Minors. XX. Wagner’s conjecture.” In: *Journal of Combinatorial Theory, Series B* 92.2 (2004), pp. 325–357. DOI: `10.1016/j.jctb.2004.08.001`.

[91] L. Roditty. “On the k Shortest Simple Paths Problem in Weighted Directed Graphs.” In: *SIAM Journal on Computing* 39.6 (2010), pp. 2363–2376.

[92] L. Roditty and U. Zwick. “Replacement paths and k simple shortest paths in unweighted directed graphs.” In: *ACM Transactions on Algorithms* 8.4 (2012), 33:1–33:11. DOI: 10.1145/2344422.2344423.

[93] J. M. M. van Rooij, H. L. Bodlaender, and P. Rossmanith. “Dynamic Programming on Tree Decompositions Using Generalised Fast Subset Convolution.” In: *17th Annual European Symposium on Algorithms (ESA)*. Ed. by A. Fiat and P. Sanders. *Lecture Notes in Computer Science* 5757. Springer, 2009, pp. 566–577. DOI: 10.1007/978-3-642-04128-0_51.

[94] S. Sahni. *Data Structures, Algorithms, and Applications in C++*. 1st ed. McGraw-Hill, 1999.

[95] R. Schwartz and Y. L. Chow. “The N-best algorithms: an efficient and exact procedure for finding the N most likely sentence hypotheses.” In: *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 1990, 81–84 vol.1. DOI: 10.1109/ICASSP.1990.115542.

[96] A. Sedeño-Noda. “An efficient time and space K point-to-point shortest simple paths algorithm.” In: *Applied Mathematics and Compututation* 218.20 (2012), pp. 10244–10257.

[97] T. Shibuya and H. Imai. “Enumerating suboptimal alignments of multiple biological sequences efficiently.” In: *2nd Pacific Symposium on Biocomputing (PSB)*. World Scientific, 1997, pp. 409–420.

[98] T. Shibuya and H. Imai. “New Flexible Approaches for Multiple Sequence Alignment.” In: *Journal of Computational Biology* 4.3 (1997), pp. 385–413. DOI: 10.1089/cmb.1997.4.385.

[99] Y. Shih and S. Parthasarathy. “A single source k-shortest paths algorithm to infer regulatory pathways in a gene network.” In: *Bioinformatics* 28.12 (2012), pp. 49–58. DOI: 10.1093/bioinformatics/bts212.

[100] F. K. Soong and E. F. Huang. “A tree-trellis based fast search for finding the N-best sentence hypotheses in continuous speech recognition.” In: *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 1991, pp. 705–708. DOI: 10.1109/ICASSP.1991.150437.

[101] K. Sugimoto and N. Katoh. “An algorithm for finding k shortest loopless paths in a directed network.” In: *Transactions – Information Processing Society Japan* 26 (1985). In Japanese, pp. 356–364.

[102] I. Wegener. *Komplexitätstheorie: Grenzen der Effizienz von Algorithmen*. Springer, Berlin, Heidelberg, 2003.

[103] V. V. Williams and R. Williams. “Subcubic Equivalences between Path, Matrix and Triangle Problems.” In: *51st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 2010, pp. 645–654. DOI: 10.1109/FOCS.2010.67.

[104] W. Xu, S. He, R. Song, and S. S. Chaudhry. “Finding the K shortest paths in a schedule-based transit network.” In: *Computers & Operations Research* 39.8 (2012), pp. 1812–1826. DOI: `10.1016/j.cor.2010.02.005`.

[105] J. Y. Yen. “Finding the K shortest loopless paths in a network.” In: *Management Science* 17.11 (1971), pp. 712–716.