Doctoral Thesis

Measuring hardness of complex problems
Approximability and exact algorithms

Author(s):
Steinová, Monika

Publication Date:
2012

Permanent Link:
https://doi.org/10.3929/ethz-a-007089803

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Measuring Hardness of Complex Problems: Approximability and Exact Algorithms

A dissertation submitted to
ETH ZURICH

for the degree of
DOCTOR OF SCIENCES

presented by
MONIKA STEINOVÁ
Mgr., Comenius University Bratislava
born on March 23rd, 1984
citizen of Slovakia

accepted on the recommendation of
Prof. Dr. Juraj Hromkovič, examiner
Prof. Dr. Peter Widmayer, co-examiner
Prof. Dr. Peter Rossmanith, co-examiner

2012
Abstract

After decades of research, it turned out that many problems of practical importance are hard, in the sense that scientists are still unable to solve these problems by fast algorithms. The need of handling tasks in practice over years has stimulated the development of different strategies how to tackle such problems. The most classical approaches are approximation algorithms, exact exponential algorithms with low exponentiated constant and parameterized complexity theory where problems are studied with respect to some parameters. This variety of methods is useful and necessary since, for each single method, also lower bound methods have been devised, showing that not every method can be successfully applied to every problem.

In this thesis, we study several hard problems that arise in operations research, bioinformatics and distributed computing. The problems we consider here are often well-known and extensively studied as they are arising in a large variety of naturally occurring situations. Our study attempts to better understand these notorious hard problems by improving known results, tackling subproblems and by investigating the hardness of modifications of these problems. We achieve our results by using all aforementioned approaches.

From operations research, we consider modifications of the Traveling Salesman Problem (TSP). In its ordered version, where \( k \) special vertices must be visited in prescribed order and edge costs satisfy the \( \beta \)-relaxed triangle inequality, we first improve the best known approximation algorithm to ratio \( k\beta \log_4(3k-3) \). Further we enhance the result for the most interesting instances with \( \beta \) close to 1 and \( n \geq 4k + 2 \).

The Minimum Steiner Cycle Problem is a modification of TSP such that only \( k \) given special vertices must be present exactly once in the constructed cycle (non-special vertices can be used at most once). We consider both cases where \( k \) is constant and non-constant (which is a generalization of TSP) as well as several edge cost metrics. For metric instances, we show that, if \( k \) is constant, the problem is solvable in polynomial time, otherwise we design an approximation algorithm that matches the best approximation ratios of TSP for both directed and undirected graphs. For \( \beta \)-relaxed triangle inequality and undirected graphs, we obtain a \((\beta^2 + \beta)\)-approximation algorithm which matches one of the best approximation algorithms for TSP under these conditions. If edge costs are without restrictions,
we show that the directed case, even with \( k = 4 \), cannot be approximated by any polynomial approximation ratio.

From the area of distributed computing, we study a problem of designing a scalable overlay network for topic-based communication which has a great importance nowadays. We are given a set of users together with a list of topics they are interested in and we aim to connect these users by the minimum number of edges such that every graph induced by users interested in a common topic is connected. Furthermore, we assume that at most \( d > 2 \) users are interested in a common topic. We give various hardness results for this problem. The problem is \( \text{APX} \)-complete, even if \( d = 3 \) when it inherits the approximation hardness of well-known minimum vertex cover problem. Furthermore, we design the first constant approximation algorithm and we present a first nontrivial parameterized algorithm for the problem parameterized by the size of the output.

Suppose that an input instance of a hard problem is obtained from experimental data which is corrupted and hence the input instance does not possess the expected properties. We investigate this scenario through a new operation, called splitting, that aims to repair the input instance. In general, one can view the splitting operation as a modification of the input instance by duplicating a single elementary element with the goal of minimizing the number of such splits such that the instance becomes corrected. For Boolean formulas, we define splitting on variables – each occurrence of a split variable is replaced by one of two new variables (the choice is ours). In graphs, we define splitting for vertices – the split vertex is replaced by two new vertices and the edges incident to the split vertex are reconnected to exactly one of the two new vertices (the choice is ours). First, we consider the minimization of the number of splits of variables in a 2-CNF formula in order to build a satisfiable formula, which proves to be \( \text{APX} \)-hard. Its dual problem, where the goal is to assign maximum number of variables at once without causing a contradiction in the partially assigned 2-CNF formula on \( n \) variables, turns out to be not approximable by a ratio of \( \frac{n}{2^{\Omega(\sqrt{\log n})}} \) (unless \( P = NP \)). We also consider splitting of vertices on graphs with \( n \) vertices with the goal to repair it to a Hamiltonian graph. In this case, we proved that no \( (n^{1-\varepsilon}) \)-approximation algorithm can exist (unless \( P = NP \)).

Analyzing genomic data for finding those gene variations which are responsible for hereditary diseases is one of the great challenges in modern bioinformatics. We propose a simple combinatorial model for classifying the set of haplotypes (i.e., the two copies of every gene inherited from the two parents) in a population according to their responsibility for a certain genetic disease. In this model, we are asked to find a satisfying assignment of a uniform 2-CNF formula with a minimum number of its \( n \) variables set to true. We present two exact algorithms – an \( O^*(1.25993^n) \) time algorithm that is further improved to \( O^*(1.2127^n) \) time. The second algorithm is also an \( O^*(2^k) \) time \( \text{FPT} \)-algorithm for the problem parameterized by the size of the solution.
Zusammenfassung


Aus dem Operations Research betrachten wir Varianten des Traveling-Salesman-Problems (TSP). Für die Variante, in der $k$ ausgezeichnete Knoten in einer vorgegebenen Reihenfolge besucht werden müssen und die Kantenkosten eine Relaxierte $\beta$-Dreiecksungleichung erfüllen, verbessern wir den besten bisher bekannten Approximationsalgorithmus auf eine Güte von $k\beta \log_2(3k-3)$. Für die interessanteste Teilmenge von Instanzen mit $\beta$ in der Nähe von 1 und $n \geq 4k + 2$ können wir die Approximationsgüte sogar noch weiter verbessern.

Das Minimum-Steiner-Kreis-Problem ist eine Modifikation des TSP, bei der nicht alle, sondern nur $k$ ausgezeichnete Knoten genau einmal in dem konstruierten Kreis vorhanden sein müssen (die nicht ausgezeichneten Knoten dürfen höchstens einmal besucht werden). Wir unterscheiden zwei Fälle danach, ob $k$ konstant ist oder nicht (letzteres führt zu einer Verallgemeinerung des TSP). In beiden Fällen
untersuchen wir jeweils verschiedene Anforderungen an die Kantenkosten. Für metrische Instanzen zeigen wir, dass für konstantes $k$ das Problem in Polynomzeit lösbar ist. Für nicht konstantes $k$ entwerfen wir einen Approximationsalgorithmus, der sowohl für gerichtete wie auch für ungerichtete Graphen dieselbe Approximationsgüte erreicht wie die besten bekannten Approximationsalgorithmen für das TSP. Für ungerichtete Graphen, die die relaxierte $\beta$-Dreiecksungleichung erfüllen, erhalten wir einen $(\beta^2 + \beta)$-approximativen Algorithmus, der dieselbe Güte erreicht wie der beste bekannte TSP-Approximationsalgorithmus für die gleiche Klasse von Graphen. Für gerichtete Graphen ohne Einschränkung der Kantenkosten zeigen wir, dass sogar für $k = 4$ kein Approximationsalgorithmus mit polynomialer Güte möglich ist.


Graphen aufzuspalten, so dass der resultierende Graph einen Hamiltonkreis enthält. Hierfür zeigen wir, dass es keinen \((n^{1-\varepsilon})\)-approximativen Algorithmus gibt, falls \(\mathcal{P} \neq \mathcal{NP}\).

Die Analyse von Genom-Daten zum Auffinden solcher Gen-Varianten, die für Erbkrankheiten verantwortlich sind, ist eine der großen Herausforderungen der modernen Bioinformatik. Wir schlagen ein einfaches kombinatorisches Modell vor, um die Menge der Haplotypen (d. h. der zwei Kopien jedes Gens, die von einem der beiden Elternteile vererbt wurden) in einer Population danach zu klassifizieren, inwieweit sie für eine bestimmte Erbkrankheit verantwortlich sind. Dieses Modell führt uns zu der Aufgabe, eine erfüllende Belegung für eine uniforme 2-CNF-Formel zu finden, in der eine minimale Anzahl der \(n\) Variablen auf 1 gesetzt ist. Wir entwerfen zwei exakte Algorithmen hierfür – einen Algorithmus mit Laufzeit in \(O^*(1.25993^n)\), den wir weiter verbessern können zu einem Algorithmus mit Laufzeit in \(O^*(1.2127^n)\). Der zweite Algorithmus ist außerdem ein \(FPT\)-Algorithmus mit einer Laufzeit in \(O^*(2^k)\), wobei die Größe der Ausgabe als Parameter betrachtet wird.
Acknowledgments

I would like to express my deepest gratitude to my supervisor Juraj Hromkovič for his guidance and support, and for nursing me in the scientific life. His optimistic attitude towards life will be my inspiration for many years.

I am in debt to Hans-Joachim Böckenhauer for all the valuable discussions we had during the preparation of this thesis, for the translation of the abstract to German and for his exceeding patience with proofreading of the draft of this thesis.

I would like to thank to all the past and present members of the Chair of Information Technology and Education with whom I spent great moments and wonderful time during my stay at ETH.

I am thankful to Peter Rossmanith and Peter Widmayer for reviewing this thesis and for agreeing to become my referees.

I appreciate all the guidance and support of Alexander Švarc with the preparation of the graphic on the cover of this thesis.

I thank also all my friends for stimulation which allowed me to become what I am now.

Last, but definitely not least, I am very grateful to my parents and my sister Eva for their care and support during the many years of my maturing.
## Contents

1 Introduction ................................. 1

2 Basic Notions and Preliminaries .......... 7
   2.1 Essentials of Mathematics and Computer Science .... 7
   2.2 Graphs and Algorithms .......................... 9
   2.3 Formulas ....................................... 12
   2.4 Decision vs. Optimization Problems .......... 14
   2.5 Approximation Techniques for Hard Problems .... 19
      2.5.1 Approximation Techniques .................. 19
      2.5.2 Classification of Optimization Problems .......... 20
      2.5.3 Lower-Bound Techniques ................... 22
   2.6 Exact Algorithms .............................. 25
      2.6.1 Branch-and-Reduce Technique ............... 26
   2.7 Parameterized Complexity Theory .......... 27
   2.8 Overview of Some Well-Known Problems .... 30
      2.8.1 Variants of the Traveling Salesman Problem .... 30
      2.8.2 The Minimum Steiner Tree Problem .......... 40
      2.8.3 The Satisfiability Problem .................. 43
      2.8.4 The Minimum Set Cover Problem and the Minimum Hitting Set Problem . 44
      2.8.5 The Minimum Vertex Cover Problem and the Maximum Independent Set Problem ..... 51

3 The Ordered Traveling Salesman Problem 55
   3.1 Problem Definition, Known Results and Outline of Our Results .... 55
   3.2 A New Approximation Algorithm for $k$-$\Delta_\beta$OTSP .......... 56
   3.3 An Improvement of the Algorithm of Böckenauer et al. ........... 63
   3.4 Further Improvements ............................ 72
   3.5 Summary ....................................... 74

4 The Steiner Cycle Problem .................. 77
   4.1 Definitions and Overview ...................... 77
   4.2 Hardness of $k$-SCP on Directed Graphs .......... 78
CONTENTS

4.3 Approximating the Minimum Steiner Cycle Problem . . . . . . 81
  4.3.1 Approximation of Directed and Undirected \( k-\Delta SCP \) and \( \Delta SCP \) . . . . . . . . . . . . . . . . . . . . . . . . . . 81
  4.3.2 Approximation of Undirected \( \Delta_2 SCP \) and \( k-\Delta_2 SCP \) . . 82
4.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

5 The Minimum Topic-Connected Overlay 87
  5.1 Preliminaries and Definitions . . . . . . . . . . . . . . . . . . . . 87
  5.2 Hardness of \text{Min-TCO} and \text{Min-}d\text{-TCO} . . . . . . . . . . . . . . 89
  5.3 A Constant Approximation Algorithm for \text{Min-}d\text{-TCO} . . . . . . . 92
  5.4 A Parameterized Algorithm for \text{Min-}d\text{-TCO} . . . . . . . . . . . . 95
  5.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

6 Splitting Problems 97
  6.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
  6.2 Satisfiability of Formulas . . . . . . . . . . . . . . . . . . . . . . 98
  6.3 Hamiltonicity of Graphs . . . . . . . . . . . . . . . . . . . . . . . 103
  6.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

7 Exact Algorithms for Uniform Min-Ones-2-SAT 111
  7.1 Definitions and Overview . . . . . . . . . . . . . . . . . . . . . . 111
  7.2 Our First Exact Algorithm . . . . . . . . . . . . . . . . . . . . . 113
  7.3 A Faster Exact Algorithm . . . . . . . . . . . . . . . . . . . . . . 119
    7.3.1 Reduction Rules . . . . . . . . . . . . . . . . . . . . . . . . 120
    7.3.2 The Algorithm and its Branchings . . . . . . . . . . . . . . 121
    7.3.3 Time Complexity and Parameterization . . . . . . . . . . . 127
  7.4 Application to a Haplotype Classification Problem . . . . . . . . 128

Bibliography 131
Chapter 1

Introduction

Modern computer science, as we know it today, was born in the first half of the 20th century. Prior to this period, the elements nowadays accepted as parts of computer science were scattered among the natural sciences – the methods of calculating mathematical expressions or creating chemical compounds were generalized into algorithms, the first (mainly) mechanical calculators evolved into computers, etc.

The computers were gradually developed from the first in the 1940s through the room-size computers in the 1960s to the first personal computers around the 1980s.

The hunger of knowledge about the potential of computers (that were not yet built) lead to studies of theoretical models of a computer developed in parallel with the physical computers. In 1936, Alan Turing described the Turing machine – the most famous and a very powerful theoretical model on which any computer computation can be simulated (even Turing machines can be simulated on a Turing machine – such a property of a model is called Turing-completeness of the model). In parallel and subsequently, many other equally powerful models such as lambda calculus, register machines and unrestricted grammars were discovered. Turing also described the first problem (the Halting problem) in his work in 1936 that cannot be solved by any algorithm. This discovery was an important milestone in algorithmics which shattered the hopes in the systematic deduction of all mathematical proofs.

Algorithms were in the spotlight from the very beginning. Their importance was confirmed also in practice: the machine time of the early computers was expensive and the number of scientists that computers could comfort was limited. Hence, the design of fast and efficient algorithms that were very cleverly optimized for the used computer was an important task of the period.

Soon scientists discovered problems of practical importance for which they were unable to find algorithms that would run in the limited time of the computers. In theoretical research, these problems were reflected by the concept of \( \mathcal{NP} \)-completeness of decision problems introduced by Stephen Cook in 1971 [38] (a similar concept was introduced by Leonid Levin in the former USSR at the approximately same time – see the Russian original [81] and its translation [103]).
Following this concept, an “easy” decision problem can be solved by an algorithm in time polynomial in its input size. All such easy problems are grouped in the class $\mathcal{P}$. Another class of problems is the class $\mathcal{NP}$. For each instance of a problem from $\mathcal{NP}$, it is verifiable whether it is a yes-instance in a time polynomial in its input size. As solving a problem in polynomial time is also a verification, $\mathcal{P} \subseteq \mathcal{NP}$. For many problems in $\mathcal{NP}$, it is not yet known whether such a problem admits a polynomial-time solution which would include the problem to $\mathcal{P}$. Nevertheless, for each problem of $\mathcal{NP}$, one can exhaustively search the solution space for any of its instances to find the answer, but this search is very expensive – it costs exponential time.

To compare the hardness of problems, the polynomial-time reductions allowing to transform a problem $P_1$ to a problem $P_2$ in a polynomial time were devised. Such a reduction shows that $P_1$ can be solved via the reduction by $P_2$ and this approach takes time just polynomially larger than the time complexity of an algorithm for $P_2$. In other words, solving the first problem $P_1$ cannot be harder than solving the second problem $P_2$ with respect to polynomial-time solvability. The problems to which any problem of $\mathcal{NP}$ can be reduced to in a polynomial time are called $\mathcal{NP}$-hard. Thus, the solution of any $\mathcal{NP}$-hard problem in polynomial time yields a polynomial-time algorithm for any problem from $\mathcal{NP}$. Note that there are harder problems that are even not present in $\mathcal{NP}$ and that are $\mathcal{NP}$-hard. The hardest kernel of the entire class $\mathcal{NP}$ are the $\mathcal{NP}$-complete problems that are the $\mathcal{NP}$-hard problems from $\mathcal{NP}$.

After the discovery of the concept of $\mathcal{NP}$-complete problems, there was a big boom and many problems that scientists could not solve by sufficiently fast algorithms turned out to be $\mathcal{NP}$-complete. Recently, scientists know thousands of practical problems that are hard.

Obviously, $\mathcal{P} \subseteq \mathcal{NP}$. However, for several decades, the most brilliant scientists were unable to answer whether $\mathcal{P} \supseteq \mathcal{NP}$ is true, i.e., whether one can solve any $\mathcal{NP}$-hard problem in a time polynomial in the input size. It is believed that the contrary is true and $\mathcal{P} \neq \mathcal{NP}$. The answer for this question would be a groundbreaking result for the entire computer science community.

Once a problem is proven to be $\mathcal{NP}$-complete, it is a good indicator that alternative methods of delivering reasonable good outcomes for the problem should be used and one should not hope that a polynomial-time algorithm computing an exact and precise solution of the problem can be easily devised.

Another type of problems that was considered in parallel with the decision problems are optimization problems. In an optimization problem, one searches for a minimum-cost solution or a maximum-cost solution with respect to some weight/cost defined in the problem. Each optimization problem can be transformed into a decision problem – one just asks whether the given instance of the optimization problem admits a solution of the weight/cost of at most/at least some initially fixed constant. Not surprisingly, if such a decision version of an optimization problem is $\mathcal{NP}$-complete, we cannot hope that the optimization problem is easy.
In this thesis, we consider several different methods that can be applied to attack hard problems. They include approximation algorithms, exact exponential algorithms and parameterized complexity theory. Each of these approaches deals with a hard problem in its own way and all are important as there is no universal method that can handle well all hard problems. Thus, we have to always choose the one that suits our needs the best or to combine some of them. There are also other techniques, which are not discussed in this thesis, and that can be combined with these methods such as randomized algorithms or heuristic algorithms.

One of the methods of tackling hard optimization problems are the so-called approximation algorithms. The general idea of such algorithms is to deliver a solution that is not extremal, but is not too far away of an extremal one (concerning the weight/cost of the two solutions) and, moreover, its time complexity is polynomial in the input size. Hence, by agreeing that the delivered solution is not optimal but with still some guarantees on its cost, the algorithm is speeded up to a reasonable time complexity.

Another approach to hard problems works as follows. Instead of accepting non-optimal solutions, one allows algorithms to use an exponential running time. Such algorithms are referred to as exact algorithms or exact exponential algorithms. The main goal in the design of an exact algorithm is to prune the branches of the algorithm that are not needed for the computation of some exact solution and hence to decrease its time complexity which is reflected in the decrease of the constant that is exponentiated in the estimated time complexity. Similarly as the story $P$ vs. $NP$, for exact exponential algorithms there is the Exponential Time Hypothesis. It basically states that there is no subexponential exact algorithm solving any $NP$-hard problem. The claim of the hypothesis is even stronger than $P$ vs. $NP$ and allows to prove stronger lower bounds which one was unable to prove under the $P \neq NP$ assumption.

Both above mentioned approaches are accounting with inputs of no special attributes. However, in practice, this is often not the case. Real-world instances have often certain additional properties that can make a generally very hard problem easier. Parameterized complexity theory is reflecting such a view on hard problems. In general, one assumes extra parameters of input instances of a problem and the parameterized algorithm solves the problem exactly, but its time complexity is dependent on the parameters (usually by a superpolynomial function). Such algorithms allow fast solutions for instances with low parameters, which is sometimes the case for real-world instances.

Of course, also compounds of two or more strategies can be considered to attack hard problems.

The Contribution of This Thesis

Our aim in this section is to discuss the contributions of our thesis and to briefly mention contributions of other people we have collaborated with in common papers. Unless it is stated otherwise, all the listed results are our own work.
In this thesis, we examine several hard problems which were either already investigated before, or are established by modifying already well-known problems. Our results are presented in Chapter 3 through Chapter 7.

Our results are mainly concerning approximability and hardness of problems. We also present some exact exponential algorithms. The parameterized complexity theory is not in the center of our thesis and the results we come across in this field are rather corollaries of our main results reflected into this theory.

In several cases, we have improved the best (at that time) known algorithm. For the new modifications of well-known problems, in several cases, we give their first characterization concerning hardness and approximability.

We discuss the modifications of the Traveling Salesman Problem (TSP) in Chapter 3 and Chapter 4.

The TSP with precedence constraints, where some vertices of a graph are required to be visited in a given order in a feasible solution, is considered in Chapter 3. We present a gradual improvement of the previously best known approximation algorithm [23]. Our first improvement from Section 3.2 was published in [25], its further improvement (for the most interesting cases), which has not been yet published, is presented in Section 3.3 and Section 3.4.

A new problem, called Steiner Cycle Problem, which is presented in Chapter 4, is a modification of TSP to a Steiner tree alike problem. Our results partially characterize the problem and several its subproblems. We present several approximation algorithms and hardness results. The results were published in [101].

Chapter 5 investigates the Minimum Topic-Connected Overlay – a problem of building a certain network used to broadcast information. Problems of building overlay networks with specific properties are recently keeping attention of scientists. We prove \( \mathcal{APX} \)-completeness of the problem restricted to a special class of its instances and we design the first nontrivial approximation algorithm for it as well. Our results, interpreted from the viewpoint of parameterized complexity theory, yield the first non-trivial parameterized algorithm for our problem parameterized by the size of the output. Our results were published in [67] and [102].

Our results in an ongoing research on the splitting operation are presented in Chapter 6. The operation aims to model a type of a reparation of problem instances that are damaged and hence not feasible. We consider three problems that are derived from the satisfiability and Hamiltonicity decision problems. All our results are hardness results and are showing that the problems are indeed very hard – in two cases we prove non-approximability by any sublinear approximation ratio (unless \( P = \mathcal{NP} \)) and, in the third case, we present an \( \mathcal{APX} \)-hardness result.

Last but not least, in Chapter 7, we present two exact algorithms for a variation of the satisfiability problem on clauses in 2-CNF where we aim for a minimum number of true variables in solutions. The second algorithm is a fixed-parameter tractable algorithm from the parameterized complexity theory viewpoint.

The exact algorithm in Section 7.2 is part of the joint work with Ján Oravec and was published as a technical report of ETH [26]. The further improvement of the exact algorithm is presented in Section 7.3. This result is part of the joint work with Michal Forišek and was published in [21].
The Organization of This Thesis

Chapter 2 contains all possible preliminary items one could need in reading this thesis. We aim to make the thesis self-contained. Hence, this chapter contains all necessary definitions and we present here the famous hard problems that we are either referring to or building on in our research. We also summarize here known results (sometimes with proofs) together with citations to the original papers so one can find the exact statements we refer to.

Our results are presented in Chapters 3 to 7. Each chapter is devoted to a single problem that we study. In the content of each chapter, there can be identified an introductory section with all extra definitions, a brief overview on known results and additional observations we need for our research. The chapter is concluded with a section that summarizes our results, discusses possible open questions and other approaches or explains an extra practical motivation.
Chapter 2

Basic Notions and Preliminaries

This chapter is devoted to a brief overview of the notation we use in this thesis. We assume that the reader is familiar with notions such as graph, formula, approximation algorithm, etc. Nevertheless, we sum up these notions in the following sections that are organized by topics. The elementary notions and definitions are fixed in Section 2.1. In Sections 2.2 and 2.3, basics of graph theory with algorithms and Boolean logic are presented. Section 2.4 is devoted to general definitions for decision and optimization problems. The notion of approximability, techniques for proving the hardness of a problem together with the categorization of problems into classes by their hardness is elaborated in Section 2.5. Our thesis investigates also other approaches that may tackle hard problems. These approaches are summarized in Section 2.6 (exact algorithms solving hard problems) and in Section 2.7 (parameterized complexity theory). Finally, in Section 2.8, we present famous problems which we either build on or we improve on in this thesis. We also overview the known results for these problems in the depth required for our thesis.

2.1 Essentials of Mathematics and Computer Science

The goal of this section is to fix the elementary mathematical notations and definitions that we use in the entire thesis and to summarize the basic facts that we use in our calculations without explicitly noticing. For additional reading that is beyond the content of this section, we would like to point the reader to standard textbooks such as [39,60,69].

The infinite set of integers \{\ldots, -2, -1, 0, 1, 2, \ldots, \} is represented by \( \mathbb{Z} \). The nonnegative integers are denoted by \( \mathbb{N} \) and, if we explicitly omit the number 0 from \( \mathbb{N} \), we have the positive integers denoted by \( \mathbb{N}^+ \). The infinite set of all rational
numbers \( \mathbb{Q} \) is defined as \( \mathbb{Q} = \{ p/q \mid p, q \in \mathbb{Z} \land q \neq 0 \} \), the positive rational numbers (excluding 0) are referred to as \( \mathbb{Q}^+ \) and the real numbers and positive real numbers (again omitting 0) are represented by \( \mathbb{R} \) and \( \mathbb{R}^+ \), respectively. All together, we have \( \mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \) and \( \mathbb{N}^+ \subset \mathbb{Q}^+ \subset \mathbb{R}^+ \).

The integer part of a real number \( r \) is denoted by \( [r] \) and referred to as the floor of \( r \), if \( x \) is an integer, \( [x] = x \). The counterpart of the floor of \( r \) is the ceiling of \( r \), denoted by \( \lceil r \rceil \). For any integer \( x \), \( [x] = \lfloor x \rfloor = x \), but for a non-integer \( r \), \( \lceil r \rceil := [r] + 1 \). For a positive rational number \( p/q \) (with \( q > 0 \)), we can simply bound its floor and ceil as

\[
(p - q + 1)/q \leq \lfloor p/q \rfloor \leq p/q \quad \text{and} \quad p/q \leq \lceil p/q \rceil \leq (p + q - 1)/q.
\]

We extensively use the logarithm function in our calculations. The natural logarithm is denoted by \( \ln(\cdot) \). If we do not indicate the base of the logarithm, i.e., \( \log(\cdot) \), we represent a logarithm with arbitrary base – in such a case, the precise value of the base is unimportant and is neglected in our calculation.

We also extensively use standard set operations. For a set \( S \), the operation \( S^0 \) is defined as the empty set. The operation \( S^2 := S \times S \) represents a Cartesian product on \( S \), i.e., \( S^2 \) corresponds to a set of all (ordered) sequences \((a, b)\) such that \( a \in S \) and \( b \in S \). The definition is inductively extended for the set of all the sequences on length \( n > 2 \) on elements of \( S \) into \( S^n \). We also use the notion of \( S^* \) that represents the infinite set of all sequences of elements of \( S \) of any length, i.e., \( S^* := \bigcup_{i=0}^{\infty} S^i \). In particular, we use this notion for \( \{0, 1\}^* \) which represents the set of all possible binary sequences. The power set \( 2^S \) of the set \( S \) is the set of all possible subsets of \( S \), i.e., \( 2^S := \{ X \mid X \subseteq S \} \). To point out that a set is a set of sets, we sometimes call such a set to be a family of sets. The size of a set \( S \) is denoted as \( |S| \) and corresponds to the number of elements of \( S \). The sets \( S_1, \ldots, S_p \) are a partition of the set \( S \) if \( S = \bigcup_{i=1}^{p} S_i \) and, for all \( 1 \leq i < j \leq p \), we have \( S_i \cap S_j = \emptyset \).

Some of our estimations are expressed through recurrence relations, or recurrences for short. In particular, in this thesis, we employ linear recurrence relations, that can be generally described as

\[
T(n + 1) = a_0 T(n) + a_1 T(n - 1) + \cdots + a_n T(0) + f(n),
\]

where \( n \geq 0 \), \( a_0, \ldots, a_n \) are constants independent of \( n \), and \( f(n) \) is a function independent of the function \( T \). Each recurrence also must have defined an initial condition, such as \( T(0) = 47 \). However, in our thesis, we usually omit them when they are trivial. More details about recurrences and about different methods of calculating their closed forms can be found in [60].

In some of our estimations we use the asymptotic analysis of functions in which one often emphasizes the essential parameters and neglects everything that becomes unimportant for sufficiently large parameters. We refer to the three asymptotic functions \( O, \Omega \) and \( \Theta \), the additional notions are defined, for instance, in [39]
2.2. Graphs and Algorithms

or [60]. Consider functions $f(n)$ and $g(n)$ on an integer parameter $n$, the three asymptotic functions are defined as

\[
\mathcal{O}(g(n)) = \{ f(n) | \exists n_0, c > 0 \text{ such that } \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n) \},
\]

\[
\Omega(g(n)) = \{ f(n) | \exists n_0, c > 0 \text{ such that } \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq f(n) \},
\]

\[
\Theta(g(n)) = \{ f(n) | f(n) \in \mathcal{O}(g(n)) \text{ and } f(n) \in \Omega(g(n)) \}.
\]

We write $f(n) = \mathcal{O}(g(n))$ to indicate that $f(n)$ is a member of $\mathcal{O}(g(n))$. The equality instead of the membership is often used for all the asymptotic functions. However, it is only one-directional, hence expressions such as $\mathcal{O}(g(n)) = f(n)$ are incorrect. The $\mathcal{O}^\ast$-notation is defined in a similar manner but neglects all polynomial factors and stresses out the exponential factors. In this thesis, we only use the function

\[
\mathcal{O}^\ast(g(n)) = \{ f(n) | f(n) \in \mathcal{O}(g(n) \cdot p(n)) \text{ for some polynomial } p \}.
\]

By an instance of a problem, we understand the complete description of a single input for the problem. The encoding of the instance will be discussed later and is usually in binary. The size of the instance $|I|$ represents the total space its description consumes. When we refer to a polynomial/exponential-time and a polynomial/exponential-space algorithm, we mean an algorithm with running time and consumed space polynomial/exponential in the size of the input, respectively.

2.2 Graphs and Algorithms

Most of the problems presented in this thesis are modeled by graphs. We partially use notation similar to the one of [42].

A graph is a pair $G = (V, E)$ of a set of vertices and a set of edges such that $E \subseteq V \times V$ for directed graphs and $E \subseteq \{ \{x, y\} | x, y \in V \land x \neq y \}$ for undirected graphs. If the set of edges of a graph is a multiset (i.e., may contain an element multiple times), we call such a graph to be a multigraph. The generalization of a graph into more dimensions is referred to as a hypergraph. The vertex set of a hypergraph is defined as in regular graphs. However, the edge set, instead of pairs of vertices, contains sets of vertices for undirected hypergraphs and ordered tuples of vertices for directed hypergraphs. The maximum size of such a set in the edge set of a hypergraph defines its dimension. Each set of $E$ defines a hyperedge of the graph and it simply connects all the vertices of the set together. Hence, the set of hyperedges $E$ is a family of sets or tuples of vertices. Most of the following definitions can be extended also to hypergraphs. We use the notion of hypergraph only in the undirected sense. If $V$ can be partitioned into two sets $U$ and $W$ such that every edge connects a vertex from $U$ to a vertex from $W$, then the graph is said to be bipartite. We often refer to the vertex set and the edge set of $G$ by $V(G)$ and $E(G)$, respectively. Until it is not stated otherwise, all of the following definitions hold for both directed and undirected graphs.
Instead of the ordered pair \((u, v)\) we often use a simplification \(uv\) for a directed edge and in the case of undirected graphs we simplify the pair \(\{u, v\}\) to an undirected edge \(uv\) (i.e., \(uv\) and \(vu\) denote the same edge). In this section, to generally refer to an edge in both types of graphs, we use the notation with an ordered pair.

Two vertices \(u\) and \(v\) are adjacent or neighbors in a graph if \(E(G)\) contains an edge with endpoints \(u\) and \(v\). In undirected graphs, we can speak of vertex incidence with an edge if the vertex is an endpoint of the edge. For undirected graphs, we denote the set of all adjacent vertices of a vertex \(u\) as \(N(u)\) and we denote \(N(u) \cup \{u\}\) as \(N[u]\). We assume that our graphs do not contain loops, i.e., edges \((u, u)\). The degree of a vertex in an undirected graph is the number of its neighbors. For directed graphs, we distinguish the in- and the out-degree of a vertex. The in-degree of a vertex \(u\) is the number of edges \(\overrightarrow{e}\) in which \(u\) appears as the second vertex in the ordered pair. The out-degree is defined similarly, just \(u\) must be the first vertex in the ordered pair corresponding to edge \(\overrightarrow{e}\). We say that a vertex is odd or even if its degree is odd or even, respectively. We say that the graph \((V', E')\) is a subgraph of graph \((V, E)\) if \(V' \subseteq V\) and \(E' \subseteq E\). An induced subgraph of a graph \((V, E)\) with respect to a set of vertices \(W \subseteq V\) is the graph \((W, E \cap (W \times W))\), for directed graphs, and \((W, E \cap \{\{u, v\} | u, v \in W \land u \neq v\}\), for undirected graphs.

A sequence \(v_0, \ldots, v_n\) of vertices such that \((v_i, v_{i+1}) \in E(G)\), for all \(0 \leq i < n\), is a walk in a graph \(G\). A walk where all edges are distinct is a trail. A trail where all vertices are distinct is called a path. The length of a walk/trail/path is the number of its edges. Two paths are vertex/edge disjoint if they do not share a common vertex/edge. By \(v_0 \sim v_n\), we denote a path starting in vertex \(v_0\) and ending in \(v_n\). A walk and a trail with identical first and last vertex is called closed walk and tour, respectively. The length of such a closed walk/tour is the length of the corresponding walk/trail. For \(n \geq 2\), a path \(v_0, \ldots, v_{n-1}\) together with an edge \((v_{n-1}, v_0)\) is referred to as a cycle of length \(n\). A cycle that contains each vertex of a graph is called a Hamiltonian cycle. A graph is Hamiltonian, if it contains a Hamiltonian cycle. A tour that contains each edge of a graph is called Eulerian tour. A graph is Eulerian, if it contains an Eulerian tour. If all vertices of an undirected graph are pairwise adjacent, then the graph is complete.

For a complete graph \(G = (V, E)\) and a path \(S = v_0, \ldots, v_n\) \((v_i \in V, 0 \leq i \leq n)\) we define a subpath as a path consisting of a subsequence of the vertices of \(S\). Formally, let \(j_0 < j_1 < \cdots < j_k\) \((k \leq n)\), then path \(v_{j_0}, v_{j_1}, \ldots, v_{j_k}\) is a subpath of \(S\). In the case that the subpath is contiguous, we explicitly mention it. We would like to note that this definition is not standard, but we employ this notion of subpath in several sections.

An undirected graph \(G\) is connected if any two of its vertices are linked by a path in \(G\). A maximal connected subgraph of graph with respect to the number of vertices is called a component of the graph. Until it is not stated otherwise, we assume in this thesis that all given undirected graphs are connected.

The graphs considered in the notions in this paragraph are always undirected. A tree \(T\) is a connected graph without cycles. By \(P_T(u, v)\) we denote the unique path in tree \(T\) between its two vertices \(u\) and \(v\). A tree that contains all the
vertices of a graph is called a spanning tree. A vertex incident to at most one neighbor in a tree is called a leaf. A rooted tree is a tree with one vertex selected to be its root. The root is just one of the tree vertices that is marked to be unique. A parent of a vertex \( v \) in a rooted tree is the unique neighbor that is the closest one to the root vertex. The root vertex does not have a parent. A child of a vertex is every neighbor that is not the parent. We can define the ancestors and the descendants of a vertex \( v \) recursively – the ancestors of \( v \) are the parent of \( v \) and all its ancestors, the descendants of \( v \) are all children of \( v \) and all their descendants. Note that the length of the path from the ancestor to the root is smaller than the lengths of the paths from all other neighbors of \( v \) to the root.

A matching of a graph is a set of its edges that pairwise do not share a common vertex. If the set of edges cannot be extended, the matching is maximal. Once the cardinality of the matching is the largest possible, we speak about a maximum matching. A perfect matching is a matching which matches all the vertices of the graph.

The function \( f : V \rightarrow \{0, \ldots, k - 1\} \) is a \( k \)-coloring of an undirected graph \( G \) if \( f(v) \neq f(u) \), for all edges \( \{u, v\} \in E \). Informally, each integer of \( \{0, \ldots, k - 1\} \) corresponds to a single color and neighboring vertices of \( G \) must be colored by distinct colors.

An edge cost function \( c \) is a function that assigns a weight to each edge in a graph. To emphasize the presence of the edge cost function we refer to such a graph as a weighted directed or undirected graph. The cost of a subgraph \( X \) of a graph is the sum of the costs of its edges and its denoted as \( c(X) \).

A minimum spanning tree and a minimum Hamiltonian cycle are a spanning tree and a Hamiltonian cycle, respectively, with a minimum cost.

The complete distance network of an undirected graph \( G = (V, E) \) with the edge cost function \( c \) on vertices \( S \subseteq V \) is a complete graph on the vertices of \( S \) that is denoted as \( G_D(S) \). For each pair of vertices, the edge cost function \( c_D \) of the network assigns the cost of the cheapest path in \( G \) with respect to \( c \). The complete distance network on \( S := V \) is often called the metric closure of the graph \( G \) with respect to the function \( c \).

A flow network is a directed graph with a capacity function that assigns a positive integer capacity to each edge. It has two special vertices called source and sink. A flow in a flow network is an integer assignment to each edge of the network with the following properties

- No flow in an edge can exceed the edge capacity.
- For each non-special vertex, the total incoming and outgoing flow has to be the same. The incoming flow to the source and the outgoing flow from the sink has to be the same, too.

A standard goal in a flow network is to find a maximum flow which corresponds to the difference of the outgoing and the incoming flow in the source vertex. If each edge is also assigned a cost for a unit of the flow that edge received, we speak about a weighted flow network. Here, the goal is often to find a maximum flow that has minimum cost.
Very informally, one can thought of a flow network as a network of pipes of different capacities. A flow can be think of as a water flow that is pumped into the network from the source and that flows out in the sink. The maximum flow in such a network is the maximum amount of water that one can send through the network to the sink.

The aim of the remainder of this section is to list the most common polynomial-time graph algorithms that one may need as a subroutine in the design of algorithms for hard problems. Since these algorithms are usually within the scope of undergraduate computer science lectures, we do not aim to present them, but we recall their names under which they can be found in the literature. Unless explicitly stated otherwise, the details of the referred algorithms can be found in [39].

Consider an undirected connected graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. In depth-first search, one explores vertices in long paths which are extended as long as a new vertices are found, whereas in the breadth-first search one searches the vertices sequentially in the increasing distance from the origin. Both searches can be performed in time $O(n + m)$ and can be used, for instance, to find a path between two vertices of $G$. There are several algorithms computing an Eulerian tour of a graph that are based on a search in the graph, e.g., the linear-time Hierholzer’s algorithm [52].

Assume that $G$ is weighted by an edge cost function $c: E \to \mathbb{Q}^+$. The standard algorithms computing the minimum spanning tree are Prim’s algorithm and Kruskal’s algorithm both with the time complexity of $O(m \log n)$.

Recall that the complete distance network of $G$ with edge cost $c$ is a graph where two vertices are connected by an edge if an only if the two vertices in $G$ are reachable and the edge cost of such an edge is the cost of the cheapest path between them. The cheapest paths between all the pairs of vertices in a graph is often computed by the Floyd-Warshall algorithm in time $O(n^3)$.

The algorithm for the maximum matching in general graphs is due to Edmonds [49]. Its simple $O(n^4)$ implementation was later improved by Micali and Vazirani in [83] to a time complexity of $O(m\sqrt{n})$. However, there are other algorithms with much easier implementation if bipartite graphs are concerned.

A very powerful tool for solving involved algorithmic problems is the minimum-cost maximum flow algorithm, see e.g., [3]. It can be used to compute a maximum flow in a flow network.

### 2.3 Formulas

In this section, we recall the elementary notions of Boolean logic that are mainly adopted from [68].

The basic keystones of Boolean logic are the Boolean values – true and false. In our thesis, we associate these two values with value 1 and value 0, respectively. We use three Boolean operations that manipulate Boolean values, namely: conjunction, disjunction and negation. The conjunction and the disjunction are binary
operators that are represented by symbols “∧” and “∨”, the unary negation is denoted by an overline above its argument.

**Definition 2.3.1.** A **Boolean variable** is any symbol to which one can associate either of the values 0 or 1. For Boolean variables in a set $X = \{x_1, \ldots, x_n\}$ ($n \in \mathbb{N}^+$), we define a Boolean function over $X$ as a mapping $f$ from $\{0,1\}^n$ to $\{0,1\}$. A literal is a negated or unnegated Boolean variable.

**Definition 2.3.2.** Let $f(x_1, \ldots, x_n)$ be a Boolean function over a set of Boolean variables $X = \{x_1, \ldots, x_n\}$. We say that an assignment $\alpha = (\alpha_1, \ldots, \alpha_n) \in \{0,1\}^n$ satisfies $f$ if $f(\alpha_1, \ldots, \alpha_n) = 1$. We say that $f$ is satisfiable if there exists an assignment $\alpha$ that satisfies $f$ and $\alpha$ is called a satisfying assignment to $f$.

**Definition 2.3.3.** Let $X$ be a finite set of Boolean variables. Any object that can be produced by recursive application of the following rules is called a **Boolean formula** over $X$.

i) The Boolean values 0 and 1 are Boolean formulas.

ii) For every Boolean variable $x \in X$, $x$ is a Boolean formula.

iii) If $F$ is a Boolean formula, then $\overline{F}$ is a Boolean formula.

iv) If $F_1$ and $F_2$ are Boolean formulas, then $(F_1 \land F_2)$ and $(F_1 \lor F_2)$ are Boolean formulas.

v) Only the expressions constructed by the previous rules are Boolean formulas over $X$.

**Definition 2.3.4.** Let $X$ be a set of Boolean variables, and let $F$ be a formula over $X$. Let $\alpha$ be an assignment to $X$. The value of $F$ under the assignment $\alpha$, denoted as $F(\alpha)$, is the Boolean value defined as follows.

i) $F(\alpha) = 0$ if $F = 0$.

ii) $F(\alpha) = 1$ if $F = 1$.

iii) $F(\alpha) = 1 - F' (\alpha)$ if $F = F'$.

iv) $F(\alpha) = (F_1 \land F_2)(\alpha) = \begin{cases} 1 & \text{if } F_1 (\alpha) = F_2 (\alpha) = 1, \\ 0 & \text{otherwise}. \end{cases}$

v) $F(\alpha) = (F_1 \lor F_2)(\alpha) = \begin{cases} 0 & \text{if } F_1 (\alpha) = F_2 (\alpha) = 0, \\ 1 & \text{otherwise}. \end{cases}$

For example $F(x_1, x_2, x_3, x_4) = (x_1 \lor x_2) \land (x_2 \lor (\overline{x_3} \land x_4)) \land x_4$ is a Boolean formula. Its value is 1 (i.e., true) for the assignment $(1,0,0,1)$, but it becomes 0 (i.e., false) if the assignment changes into $(1,0,1,1)$ or $(1,0,0,0)$.

A **clause** over a set of Boolean variables $X$ is a formula consisting of disjunctions of literals from $X$. Its **size** is the number of different literals that it contains. In the following, we often omit brackets in formulas if the notation is unambiguous and we often omit the word “Boolean” when referring to Boolean logic objects. To shorten the notation, we use $\bigwedge_{i=1}^m f_i$ instead of $f_1 \land \ldots \land f_m$ and $\bigvee_{i=1}^m f_i$ instead of $f_1 \lor \ldots \lor f_m$, where $f_1, \ldots, f_m$ are formulas over variables from a set $X$. 
Definition 2.3.5. Any Boolean formula over $X$ consisting of a conjunction of clauses is called to be in conjunctive normal form or CNF in short. For every positive integer $k$, a formula is in $k$-CNF if every of its clauses has size at most $k$.

Our exemplary formula is not in conjunctive normal form, but it can be altered into an equivalent formula $F(x_1, x_2, x_3, x_4) = (x_1 \lor x_2) \land (x_2 \lor \overline{x_3}) \land x_4$ that is in $k$-CNF for $k \geq 2$.

A $k$-CNF clause is called positive, if all literals in it are unnegated variables, it is called negative, if all literals are negated variables, and mixed, if it contains negated and unnegated variables.

2.4 Decision vs. Optimization Problems

There are many algorithmic problems that are studied from different points of view in the literature. We study, in this thesis, several such problems from the complexity theory viewpoint. If the reader is not familiar with this topic, we can point to [68] and [11], where the notions of this section are adopted from.

For a decision problem, we are interested whether the input satisfies a certain property. Thus the output of such a problem is a “yes” or “no” answer. As an easy example of such a problem consider the question: “Does the given set of integers contain the number 47?” The answer is simple, either yes or no. Formally we define such problems as follows.

Definition 2.4.1. A problem $P$, whose set of instances $I$ can be partitioned into two sets $I^+$ and $I^-$, is a decision problem if one asks, for any instance $x \in I$, to verify whether $x \in I^+$. We say that an algorithm $A$ solves a decision problem $P$, if it halts on every instance $x \in I$ and returns “yes” if and only if $x \in I^+$ and “no” otherwise.

The need of understanding why we are unable for decades to design reasonably fast algorithms for solving certain problems led us to study their complexity on a finer scale. The most basic and well known distinction among decision problems is the definition of the classes $\mathcal{P}$ and $\mathcal{NP}$:

$$\mathcal{P} = \text{class of problems solvable by a polynomial-time algorithm}$$

$$\mathcal{NP} = \text{class of problems verifiable by a polynomial-time algorithm}$$

In both classes, the time complexity is measured in the input size. For us, the problems from class $\mathcal{P}$ are considered to be “easy” as we consider algorithms that need a polynomial running time to be “fast”. On the contrary, the problems of $\mathcal{NP}\setminus\mathcal{P}$ are considered to be “hard” as there is not known any “fast” algorithm that would solve any of them. Note that there are other classes of decision problems in the time complexity hierarchy, however, we do not mention them here as we do not need them in the remainder of this thesis.

It is clear, that once we are able to decide a problem, this can be also viewed as the proof of the validity of the result, i.e., as a verification of the result, and
thus $P \subseteq \mathcal{NP}$. However, the other membership is a very famous open problem. It is conjectured that $P \neq \mathcal{NP}$. This would mean that there exist problems that are not exactly solvable in polynomial time and thus we need to tackle them by other approaches.

Distinguishing two classes of decision problems is reasonable. If one limits the problems that are “efficiently solvable” only to those that admit a solution with the time complexity bounded by a fixed degree polynomial, the resulting definition would not be robust enough. On one hand, for different accepted models of computation, the corresponding classes of efficiently solvable problems would be different. On the other hand, it can easily happen that in a couple of years the hardware will improve so that algorithms with higher degree polynomial-time complexity became reasonably fast. Moreover, it often happens that, once a polynomial-time barrier of a problem is broken, people design techniques to decrease the polynomial to a reasonable degree. An example of such an evolution is the polynomial-time algorithm for primality testing – until now, the degree of the polynomial was decreased from 12 [2] to 6 [80].

The design of reductions between problems is one of the methods that is used for their classification. The main idea is that a reduction is an algorithm transforming the problem $P_1$ to the problem $P_2$ thanks to which we are able to solve problem $P_1$, assuming we can solve the problem $P_2$. We also can see than once a reduction between two problems exists, the problem that is reduced to is at least as hard as the problem that is reduced from.

**Definition 2.4.2.** We say that a decision problem $P_1$ is polynomial-time reducible\(^1\) to decision problem $P_2$, and denote this as $P_1 \leq_P P_2$, if there exists a polynomial-time algorithm $A$ that computes a mapping of input instances $I_1$ of $P_1$ to input instances $I_2$ of $P_2$ such that, for every $x \in I_1$,

$$x \in I_1^+ \iff A(x) \in I_2^+. $$

The algorithm $A$ is then called a polynomial-time reduction from $P_1$ to $P_2$.

For a simple example of a polynomial-time reduction, consider the following two problems. Problem $P_1$: “Given a $k$-CNF Boolean formula ($k > 3$), is there an assignment of its variables such that the formula is true?” and problem $P_2$: “Given a 3-CNF Boolean formula, is there an assignment of its variables such that the formula is true?” When reducing $P_1$ to $P_2$ we divide the large clauses of a $k$-CNF formula into smaller ones in 3-CNF by introducing new variables. More precisely, we replace each clause $C = (l_1 \lor \ldots \lor l_k)$ by a conjunction of several new clauses (here $a_i$ denotes a new, yet unused auxiliary variable):

$$C' = (l_1 \lor l_2 \lor a_1) \land (a_1 \lor l_3 \lor a_2) \land \ldots \land (a_{k-4} \lor l_{k-2} \lor a_{k-3}) \land (a_{k-3} \lor l_{k-1} \lor l_k).$$

Observe that, if there is no assignment such that $C$ is true, there is also no assignment such that the formula $C'$ is true. (It is impossible to find assignment values

\(^1\)in literature also known as polynomial-time Karp-reducible
to the auxiliary variables $a_i$ to make $C'$ true.) On the other hand, if there exists an assignment such that $C$ is true, we can set the auxiliary variables to satisfy also $C'$. Hence the outcome of $P_1$ on $C$ is always the same as the outcome of $P_2$ on $C'$ and we are able to solve $P_1$ by using an algorithm for $P_2$.

**Definition 2.4.3.** We say that a problem $P_1$ is polynomial-time Turing-reducible to a problem $P_2$, and denote as $P_1 \leq_T P_2$, if there exists a polynomial-time algorithm $A$ that uses a polynomial-time algorithm solving problem $P_2$ as a subroutine to compute a solution for any input instance of $P_1$. The algorithm $A$ is then called a Turing reduction from $P_1$ to $P_2$.

An example of a Turing reduction is the process of finding a satisfying assignment for a given formula, given a procedure for deciding whether such an assignment exists. Imagine we are able to decide the problem $P_2$: “Is there an assignment that satisfies a given Boolean formula $F$?” in polynomial time. Then we can find such an assignment, i.e., solve the problem $P_1$: “Output a satisfying assignment of a satisfiable Boolean formula $F(x_1, \ldots, x_n)$.” in polynomial time as follows. First try to set $x_1 = 0$, simplify $F$ accordingly and find out whether the formula $F_1$ constructed in this way is satisfiable. If yes, output $x_1 = 0$ and continue the process with $F_1$ and $n - 1$ variables. Otherwise, since $F$ is satisfiable, $x_1$ has to be 1 and hence modify $F$ in accordance and continue on a formula with $n - 1$ variables. By such process, we always find a satisfiable assignment to $F$. Observe that the algorithm for $P_2$ is used inside our reduction several times, hence our reduction is not a polynomial-time Karp reduction.

Note that every polynomial-time Karp reduction is also a Turing reduction. But the contrary is unknown, it is open whether Turing reducibility implies Karp reducibility. Thus, in our thesis, if it is not stated otherwise, we always use the polynomial-time Karp reductions.

Once we are familiar with the notion of reduction, we can define a class of problems that are hard under this reduction:

**Definition 2.4.4.** We say that a problem $P$ is $\mathcal{NP}$-hard (under polynomial-time reductions) if, for every problem $P'$ from $\mathcal{NP}$, there exists a polynomial-time reduction such that $P' \leq_P P$.

The $\mathcal{NP}$-hardness of problems reflects their hardness with respect to the class $\mathcal{NP}$ – solving an $\mathcal{NP}$-hard problem in polynomial time would imply the solvability of any member of $\mathcal{NP}$ in polynomial time. However, an $\mathcal{NP}$-hard problem might belong to a much higher class in the hierarchy. Thus, solving it in polynomial time might be even more challenging that solving problems from the class $\mathcal{NP}$ in polynomial time. Therefore, we define the kernel of the class $\mathcal{NP}$ – the problems of the class $\mathcal{NP}$ that are hard to be solved and whose solvability in polynomial time implies solvability of all the problems from $\mathcal{NP}$ in polynomial time.

**Definition 2.4.5.** We say that a problem $P$ is $\mathcal{NP}$-complete (under polynomial-time reductions) if $P \in \mathcal{NP}$ and $P$ is $\mathcal{NP}$-hard.
In optimization problems, in contrast to decision problems, we are not interested in whether an input satisfies a certain property. Instead, we need to find a solution that is extremal, i.e., either minimal or maximal. A very famous optimization problem is the so-called minimum vertex cover problem that is discussed also in Section 2.8: Find the minimum-size set of vertices of a given graph such that every edge of the graph is incident with some vertex of this set, in this case, we say that every edge is covered by the set. Formally, an optimization problem is defined as follows.

Definition 2.4.6. An optimization problem is a 4-tuple $U = (L, M, \text{cost}, \text{goal})$, where

i) $L$ is a set of feasible problem instances,

ii) $M$ is a function that assigns a set of possible outcomes to every $x \in L$, i.e., $M(x)$ is called the set of feasible solutions for $x$,

iii) $\text{cost}$ is the cost function that assigns a positive real number $\text{cost}(u, x)$ to every pair $(u, x)$, where $x \in L$ and $u \in M(x)$.

iv) $\text{goal} \in \{\text{minimum}, \text{maximum}\}$.

In this thesis, we will omit the second parameter of the cost function if the notation is unambiguous to simplify it. We would also like to note that some of our optimization problems will be defined on weighted undirected graphs with an edge cost $c$ and their feasible solutions will be subgraphs of certain properties. In such a case, the cost function will be always defined as the sum of the edge cost of the edges in a feasible solution $S$, i.e., $\text{cost}(S) = \sum_{e \in E(S)} c(e)$.

Note that, formally, the definition of an optimization problem should contain also the description of the input and output encoding. However, in our thesis, this is not so important and thus we omit it. We assume that both the input and the output of any problem mentioned in this thesis are encoded in binary. Note that changes of the input and the output encoding between arbitrary $n$-ary systems, for $n > 1$, changes the size of the input only by a polynomial factor. Since this thesis handles problems of polynomial or higher time complexity, changes of the encoding are unimportant and can increase our measurements only by a polynomial factor. The only forbidden encoding is in unary. This is because the unary encoding blows up the size of the binary encoded input exponentially and thus, when measuring the time complexity of a computation by the size of the input, the exponential algorithm on binary inputs becomes polynomial in the size of the unary input.

Next, we introduce the classes $\mathcal{PO}$ and $\mathcal{NPO}$ of optimization problems which are the counterparts of classes $\mathcal{P}$ and $\mathcal{NP}$ for decision problems.

Definition 2.4.7. $\mathcal{NPO}$ is a class of optimization problems, where a problem $U = (L, M, \text{cost}, \text{goal})$ is a member of $\mathcal{NPO}$ if the following conditions hold:

i) the decision whether a string $x \in \{0, 1\}^*$ belongs to the set $L$ is in $\mathcal{P}$,

ii) there exists a polynomial $p_U$ such that

$$(a) \ \text{for every } x \in L, \text{ and every } y \in M(x), |y| \leq p_U(|x|),$$
(b) there exists a polynomial-time algorithm that, for every \( y \in \{0, 1\}^* \) and every \( x \in L \) such that \( |y| \leq pu(|x|) \), decides whether \( y \in M(x) \).

iii) the function \( \text{cost} \) is computable in polynomial time.

The first condition ensures us that one can efficiently verify whether a string is an instance of the problem \( U \). The following condition ensures that the size of the solutions is polynomial in the size of the problem instances and one can verify in polynomial time whether a string \( y \) is a feasible solution to any given input instance \( x \). The last condition ensures that the cost of any solution can be efficiently determined.

The conditions i) and iii) are natural, as the kernel of the problem we are interested in is optimization and neither the tractability of the decision whether the input describes an input instance of the problem nor the tractability of the computation of the cost function. The condition ii) is the part that corresponds to the definition of the class \( \text{NP} \) for the decision problems.

**Definition 2.4.8.** By \( \mathcal{PO} \) we denote the class of optimization problems \( U = (L, M, \text{cost}, \text{goal}) \) such that

i) \( U \in \mathcal{NP} \), and

ii) there is a polynomial-time algorithm that, for every input instance \( x \in L \), computes an optimal solution for \( x \).

The \( \mathcal{PO} \) vs. \( \mathcal{NP} \) story is the same story for optimization problems as the \( \mathcal{P} \) vs. \( \mathcal{NP} \) story for decision problems. The problems in the class \( \mathcal{PO} \) are the problems that are solvable in polynomial time (e.g., shortest path in a graph) and for problems of the class \( \mathcal{NP} \) feasibility of a solution can be verified in polynomial time.

For each optimization problem we can define its corresponding decision version as follows.

**Definition 2.4.9.** Let \( U = (L, M, \text{cost}, \text{goal}) \) be an optimization problem from \( \mathcal{NP} \). The problem \( P_U \) of deciding, for each pair \( (I, x) \) such that \( I \) is an instance of \( U \) and \( x \) is a binary encoded integer, whether the cost of an optimal solution of \( I \) is at most \( x \), for a minimization problem \( U \) and at least \( x \) for a maximization problem \( U \) is a decision problem that corresponds to the optimization problem \( U \).

As we will later see and employ, a decision problem that corresponds to the minimum vertex cover problem is the problem of deciding whether a given graph has a vertex cover of a certain size.

Note that, once \( U \in \mathcal{PO} \), its corresponding decision version is in \( \mathcal{P} \). On the other hand, if the decision version \( P_U \) of an optimization problem \( U \) is \( \mathcal{NP} \)-hard then \( U \notin \mathcal{PO} \), unless \( \mathcal{P} = \mathcal{NP} \). Thus we can speak also about \( \mathcal{NP} \)-hardness of an optimization problem. However, speaking about \( \mathcal{NP} \)-completeness of an optimization problem is incorrect as the class \( \mathcal{NP} \) is defined only for decision problems and thus an optimization problem cannot be its member.
2.5 Approximation Techniques for Hard Optimization Problems

In this section, we give the basic definitions related to optimization problems, and we present some techniques used to deliver positive and negative results. We discuss the classes of optimization problems based on how well they can be approximated. The notations are mainly adopted from [68], where more details on this topic can be found.

2.5.1 Approximation Techniques

Approximation algorithms are one of the most successful approaches for solving hard problems. Instead of requiring an exact and precise solution for a given hard problem, we allow the approximation algorithm a small deviation in the cost of the delivered solution. Thus, the solution is feasible, but its cost does not have to be optimal. By paying such, sometimes in practice not so high, price, the gain is huge – the algorithm is speeded up from an exponential running time to a reasonable polynomial running time. To distinguish the quality of such solutions and also the quality of an approximation algorithm, we define the approximation ratio of an algorithm as follows. Note that, if it is clear from the context, we omit the instance in the cost function to simplify the notation.

**Definition 2.5.1.** Let $U = (L, M, \text{cost}, \text{goal})$ be an optimization problem and let $A$ be a consistent\(^2\) algorithm for $U$. For every $x \in L$, with an optimal solution $\text{OPT}_U(x)$, the approximation ratio of $A$ on $x$ is defined as

$$R_A(x) = \max \left\{ \frac{\text{cost}(A(x))}{\text{cost}(\text{OPT}_U(x))}, \frac{\text{cost}(\text{OPT}_U(x))}{\text{cost}(A(x))} \right\}.$$

For any $n \in \mathbb{N}$, we define the approximation ratio of $A$ as

$$R_A(n) = \max \{ R_A(x) \mid x \in L \cap \{0, 1\}^n \}.$$

For any $\delta > 1$, we say that $A$ is a $\delta$-approximation algorithm for $U$ if $R_A(x) \leq \delta$, for every $x \in L$.

For every function $f : \mathbb{N} \to \mathbb{Q}^+$, we say that $A$ is an $f(n)$-approximation algorithm for $U$ if $R_A(n) \leq f(n)$, for every $n \in \mathbb{N}$.

The definition of the approximation ratio of an algorithm may look quite involved, however the contrary is true. The general idea how the quality of an algorithm on an input instance is measured, is to compare the cost of its solution with the cost of the optimal solution. Hence, the overall quality – the approximation ratio of the algorithm, is the worst case that can happen over all the inputs of a fixed size.

---

\(^2\)An algorithm $A$ solving $U$ is consistent if, for any input $x \in L$, it computes a solution $A(x) \in M(x)$. 
Usually, we cannot determine the cost of the optimal solution for all instances in polynomial time (otherwise, we would be able to design an optimal algorithm and the approximation need not be considered). A standard technique in the design of approximation algorithms is to deliver a feasible solution that is built from pieces whose cost can be bounded by the cost of the optimal solution and thereby to guarantee the overall approximation ratio.

The ratio of the algorithm expressed as a maximum over two fractions that are inverse to each other is used just for technical reasons so the ratio is always at least 1. If we are dealing with a minimization problem, then $R_A(x)$ is always $\frac{\text{cost}(A(x))}{\text{cost}(\text{OPT}_U(x))}$ and for a maximization problem it is always $\frac{\text{cost}(\text{OPT}_U(x))}{\text{cost}(A(x))}$. Sometimes, instead of the notion “approximation algorithm”, we just use the notion “approximation”.

There is a simple greedy algorithm for the minimum vertex cover problem that is a 2-approximation (and has yet the best known ratio independent of properties of the input). The algorithm sequentially picks both endpoints of an uncovered edge and inserts them into the covering set and then removes the endpoints together with their incident edges from the graph. A simple argument shows that, by such an approach, we can be off by a factor of at most 2: Every edge that the algorithm picked must be covered. Moreover, the picked edges are not neighboring, thus no two of them can be covered by the same vertex and hence the optimal solution must contain one vertex for each such edge. Our algorithm picked always two instead of one of such vertices and thus there are at most twice as many vertices in the covering set than in the optimal solution.

Finally, we would like to mention a crucial difference between approximation algorithms and heuristics. With an approximation algorithm, we always guarantee how far its solution can be away from the optimum. This holds for each instance of the problem with no difference if the instance is a real-world instance or something carefully designed in the “laboratory”. Heuristics often work well just for some sets of inputs with certain properties. Thus they can be efficient on many real-world instances, however there are no guarantees how well a heuristic performs in general – their quality for arbitrary instances is not known, its performance can be exponential in time or even the delivered solution might not be feasible anymore.

### 2.5.2 Classification of Optimization Problems by Their Approximability

In this subsection, we shortly summarize the different classes of optimization problems according to their approximability. Naturally, we can divide $\mathcal{NP}$ into many classes, however, due to the focus of this thesis, we will mention only the most prominent ones. Since every problem from $\mathcal{P}$ is exactly solvable in a polynomial time, we assume that $\mathcal{PO} \subseteq \mathcal{NP}$, otherwise our hierarchy would collapse. Note that this assumption is comparable with the usual $\mathcal{P} \subseteq \mathcal{NP}$ assumption.

We start with the classes that are the “closest” to $\mathcal{PO}$.
2.5. Approximation Techniques for Hard Problems

Definition 2.5.2. Let $U = (L, M, \text{cost, goal})$ be an optimization problem. An algorithm $A$ is called a polynomial-time approximation scheme (PTAS) for $U$, if, for every input pair $(x, \varepsilon)$ ($x \in L$ and $\varepsilon \in \mathbb{Q}^+$), $A$ computes a solution $A(x)$ with a ratio $R_A(x)$ that is at most $\varepsilon$-far from the optimal solution (i.e., $R_A(x) \leq 1 + \varepsilon$) and its time complexity can be bounded by a function that is polynomial in $|x|$. If the time complexity of such scheme can be bounded by a function polynomial in both $|x|$ and $\varepsilon^{-1}$, then we say that $A$ is a fully polynomial-time approximation scheme (FPTAS) for $U$.

Both PTAS and FPTAS are among the most useful techniques for solving hard problems on real-world data. The schemes guarantee that, for any small error $\varepsilon > 0$ we choose, we can always construct an approximation algorithm that delivers a solution with cost by at most $\varepsilon$ off the cost of the optimum. In PTAS, since we bound the time complexity only in terms of $n$, the actual running time of the algorithm may be as huge as exponential in $\varepsilon^{-1}$. Thus, for reasonably small errors, its running time can be still unacceptably high and unpractical. This is not the case in FPTAS, where the time complexity is polynomial also in $\varepsilon^{-1}$, and thus the algorithm may remain practical even for smaller errors and/or larger instances.

One, not very frequent, scenario where a PTAS is very useful for a discrete optimization problem is the following one. We might know the minimum difference between the costs of two feasible solutions of a problem. Hence, setting our error $\varepsilon$ small enough, we may be able to obtain an approximate solution that is also the optimal one as no other solution may exists within the “$\varepsilon$-neighborhood” of the optimal solution.

Another scenario that may arise in practical applications is that, even though we are computing everything precisely, the obtained result is further used in the process of delivering the output to the user, i.e., our hard problem is just an intermediate step. In this process, we might need to round the result or the process inherently rounds it. Thus the solution of our problem need not be exact and can be calculated by a PTAS with an error small enough to neglect all the imprecisions that will anyway come up in the later rounding.

Sadly, the above scenarios for PTAS in real-world applications cannot be used often. This is mainly because the time complexity of an approximation algorithm with a small required error is often huge and thus the algorithm is too slow. However, at least the first scenario can give us new theoretical results. For example, if we assume by a contradiction that a PTAS for an optimization problem exists, one could use the PTAS with certain error to decide an $\mathcal{NP}$-hard decision problem.

The classes of all optimization problems that admit a PTAS or an FPTAS are denoted by $\mathcal{PTAS}$ and $\mathcal{FPTAS}$, respectively. Obviously, any FPTAS is also a PTAS, and thus $\mathcal{FPTAS} \subseteq \mathcal{PTAS}$. One of the representatives of $\mathcal{FPTAS}$ is the famous knapsack problem: given is a set of items with their integer sizes and values and a positive integer – the capacity of a knapsack. We have to select some of these items and pack them into the knapsack maximizing the sum of their values such that the overall sum of the sizes of the selected items is at most the capacity of the knapsack. Under the assumption that $\mathcal{P} \neq \mathcal{NP}$, there are also
problems that admit a PTAS but do not have an FPTAS. One of such problems is an extension of the knapsack problem to more dimensions (actually, extension to two dimensions suffices). More precisely, for each item, in addition, we are given its integer weight, and an overall (integer) limit of the weights of items the knapsack can contain. The goal is still to maximize the overall value of the items packed into the knapsack while their overall size and weight cannot exceed the given size and weight limit. For more details on the knapsack topic, see [74].

We further continue with building of our hierarchy by introducing the classes of problems that are above PTAS. We start with the most prominent class APX.

**Definition 2.5.3.** The class APX contains every optimization problem \( U \in \mathcal{NPO} \) for which there exists a polynomial-time \( \delta \)-approximation algorithm, for some \( \delta > 1 \).

Similarly to \( \mathcal{FPTAS} \) and \( \mathcal{PTAS} \), there are problems that are in \( \mathcal{APX} \) but do not admit PTAS, unless \( \mathcal{P} = \mathcal{NP} \). A very famous such problem is the metric traveling salesman problem which is further discussed in Subsection 2.8.1.

**Definition 2.5.4.** The class \( \mathcal{LOGAPX} \) contains every optimization problem \( U \in \mathcal{NPO} \) for which there exists a polynomial-time \( f(n) \)-approximation algorithm, where \( f(n) \in \Theta(\log n) \).

Not surprisingly, a similar relation as for PTAS vs. APX holds also for APX vs. LOGAPX. The problems minimum set cover and minimum hitting set are examples of such problems. Both problems are in \( \mathcal{LOGAPX} \), but do not admit a constant approximation algorithm. Moreover, both problems, as we discuss in Subsection 2.8.4, are \( \mathcal{LOGAPX} \)-complete and thus neither problem admits a better approximation algorithm than a logarithmic one, unless \( \mathcal{P} = \mathcal{NP} \).

When we drop the assumption on metric edge costs in the metric traveling salesman problem, we obtain the general version, called traveling salesman problem. Assuming that \( \mathcal{P} \neq \mathcal{NP} \), this problem does not admit a polynomial-time approximation algorithm with a polynomial approximation ratio, even though it is a member of \( \mathcal{NPO} \).

Thus, altogether, if \( \mathcal{P} \neq \mathcal{NP} \), we have

\[
\mathcal{PO} \subseteq \mathcal{FPTAS} \subseteq \mathcal{PTAS} \subseteq \mathcal{APX} \subseteq \mathcal{LOGAPX} \subseteq \mathcal{NPO}.
\]

Similarly as \( \mathcal{NP} \)-hardness and \( \mathcal{NP} \)-completeness, we can define similar notions for all of these classes. The formal definition for several of these classes is presented in the following subsection where first the appropriate reduction is introduced.

### 2.5.3 Lower-Bound Techniques

In this subsection, we shortly summarize the techniques used, under some reasonable assumption such as \( \mathcal{P} \neq \mathcal{NP} \), to prove lower bounds on the polynomial-time approximability for concrete optimization problems.
Reduction to \(\mathcal{NP}\)-hard Decision Problems

To be able to clearly formulate the ideas of the following reduction, we need an extra definition.

**Definition 2.5.5.** Let \(U = (L, M, \text{cost}, \text{goal})\) be an optimization problem and let \(\text{OPT}_U(x)\) be an optimal solution of \(U\) for an instance \(x \in L\). For every function \(f: \mathbb{N} \to \mathbb{N}\), we define an \(f(n)\)-approximation problem for \(U\), as the problem of finding, for every input \(x \in L\), a feasible solution \(S(x) \in M(x)\) such that

\[
\max \left\{ \frac{\text{cost}(S(x))}{\text{cost}(\text{OPT}_U(x))}, \frac{\text{cost}(\text{OPT}_U(x))}{\text{cost}(S(x))} \right\} \leq f(|x|).
\]

Note that the above formula and the notion of a \(f(n)\)-approximation problem for optimization problem \(U\) are nothing else than the formal description of the question: “Is there a polynomial-time \(f(n)\)-approximation algorithm for \(U\)?” Indeed, the \(f(n)\)-approximation problem for \(U\) is in \(\mathcal{P}\) if and only if there exists a polynomial-time \(f(n)\)-approximation algorithm for \(U\).

The general idea of reductions to \(\mathcal{NP}\)-hard decision problems is as follows. Let \(D\) be an \(\mathcal{NP}\)-hard decision problem on a set of instances \(I = I^+ \cup I^-\), and let \(U_{f(n)}\) be an \(f(n)\)-approximation problem for the optimization problem \(U = (L, M, \text{cost}, \text{goal}) \in \mathcal{NP}O\). One has to find a reduction \(T: I \to L\), i.e., a polynomial-time transformation of instances of \(D\) to instances of \(U\) such that there exists an efficiently computable function \(t: \mathbb{N} \to \mathbb{N}\) with the following two properties:

i) for every instance \(x \in I^+, M(T(x))\) contains a solution \(y\) with \(\text{cost}(y) \leq t(|x|)\),

ii) for every instance \(x \in I^-,\) all solutions \(z \in M(T(x))\) have \(\text{cost}(z) > f(t(|x|))\).

The aim of the transformation is to separate instances of \(I\) to the ”yes” instances and to the ”no” instances by employment of the \(f(n)\)-approximation problem. Once we are able to construct a polynomial-time reduction \(T\), we can use an \(f(n)\)-approximation algorithm for \(U\), to decide \(D\): first transform the given instance of \(D\) by \(T\) to an instance of \(U\), apply the \(f(n)\)-approximation algorithm on this instance and then, by the relation of the cost of the delivered solution to \(t(|x|)\) and \(f(t(|x|))\), respectively, decide the membership in \(D\). Hence, if we consider a \(\mathcal{NP}\)-hard problem \(D\), and we assume that \(\mathcal{P} \neq \mathcal{NP}\), it directly follows that \(U_{f(n)}\) is not in \(\mathcal{P}\) and thus no polynomial-time \(f(n)\)-approximation algorithm for \(U\) can exist.

This type of reduction is used in Section 4.2 and Section 6.3 of this thesis.

**Approximation-Preserving Reductions**

The concept of approximation-preserving reductions is similar to the polynomial-time Karp and Turing reductions (for definitions see Section 2.4). We can design
a polynomial-time reduction between optimization problems but such a reduction would not be of much help. The problems would be transformed in a polynomial time back and forth, but we would not be able to relate the approximation ratios of the problems. Therefore, we have to introduce a slightly more powerful reduction that also gives us some guarantees on the approximation ratios.

Definition 2.5.6. Consider two optimization problems from $\mathcal{NP}_O$: $P = (L_P, M_P, \text{cost}_P, \text{goal}_P)$ with an optimal solution $\text{OPT}_P(p)$ instances $p \in L_P$ and $Q = (L_Q, M_Q, \text{cost}_Q, \text{goal}_Q)$ with an optimal solution $\text{OPT}_Q(q)$ for instances $q \in L_Q$. We say that $P$ is AP-reducible to $Q$, if there exist functions $f : L_P \times \mathbb{Q}^+ \rightarrow L_Q$ and $g : L_P \times L_Q \times \mathbb{Q}^+ \rightarrow L_P$ and a constant $\alpha > 0$ such that:

1. For any $x \in L_P$ and any $\epsilon > 0$, $f(x, \epsilon) \in L_Q$.
2. For any $x \in L_P$, for any $\epsilon > 0$, and any $y \in M_Q(f(x, \epsilon))$, $g(x, y, \epsilon) \in M_P(x)$.
3. The functions $f$ and $g$ are computable in polynomial time with respect to the sizes of instances $x$ and $y$, for any fixed $\epsilon$.
4. The time complexity of computing $f$ and $g$ is non-increasing with $\epsilon$ for all fixed instances of size $|x|$ and $|y|$.
5. For any $x \in L_P$, for any $\epsilon > 0$, and for any $y \in M_Q(f(x, \epsilon))$

$$\max \left\{ \frac{\text{cost}_Q(y)}{\text{cost}_Q(\text{OPT}_Q(f(x, \epsilon)))}, \frac{\text{cost}_Q(\text{OPT}_Q(f(x, \epsilon)))}{\text{cost}_Q(y)} \right\} \leq 1 + \epsilon$$ implies

$$\max \left\{ \frac{\text{cost}_P(g(x, y, \epsilon))}{\text{cost}_P(\text{OPT}_P(x))}, \frac{\text{cost}_P(\text{OPT}_P(x))}{\text{cost}_P(g(x, y, \epsilon))} \right\} \leq 1 + \alpha \epsilon.$$

The reduction that satisfies all the conditions above is called an approximation preserving reduction or an AP-reduction for short.

The above formula is the key property that describes the relationship between the approximation ratios. Informally, if there is an AP-reduction from the problem $P$ to the problem $Q$ and we have a polynomial-time $(1 + \epsilon)$-approximation algorithm $A$ for $Q$, we are able to design a polynomial-time $(1 + \alpha \epsilon)$-approximation algorithm for $P$: Reduce the given instance $I_P$ of $P$ by the AP-reduction to the instance $I_Q$ of $Q$ (the function $f$), compute a feasible solution $S_Q$ of $I_Q$ with $A$, and then finally, reduce $S_Q$ to the solution $S_P$ of $I_P$ in $P$ (the function $g$). The properties in the definition assure us that all the transformations can be done in a polynomial time, for each instance $I_P$ we are able to construct an instance of $Q$ and for each feasible solution $S_Q$ in $Q$ we are able to construct a feasible solution $S_P$ in $P$ and the ratio of the cost of the $S_P$ to the optimal solution on $I_P$ is not arbitrary large, it is at most $1 + \alpha \epsilon$.

Note that there are many other reductions that are more concrete or general, see e.g., [41]. However, they are not in the focus of this thesis and thus we omit
their further comparisons. For our needs, the AP-reduction is sufficiently general, and we use it in Chapter 5 for proving lower bounds on approximability.

We are able to properly define hardness and completeness of several classes in our approximation hierarchy.

**Definition 2.5.7.** We say that a problem $P$ is F-hard (under AP-reductions) if, for any problem $P'$ from the class $F$, there exists an AP-reduction that transforms any instance of $P'$ to an instance of $P$, where $F \in \{\text{APX}, \text{LOGAPX}\}$.

**Definition 2.5.8.** We say that a problem $P$ is F-complete (under AP-reductions) if $P \in F$ and $P$ is F-hard, where $F \in \{\text{APX}, \text{LOGAPX}\}$.

Another successful reduction for proving inapproximability is the so-called gap-preserving reduction. Its crucial idea is to consider gap problems, that are special decision problems corresponding to optimization problems. A gap problem asks to decide whether the ratio of an optimal solution for a given instance and the size of the instance is below or above some threshold values $s$ and $c$. Once we have a $\frac{c}{s}$-approximation algorithm for our optimization problem, we can prove that the corresponding gap problem with the threshold values $(s, c)$ is in $\mathcal{P}$. On contrary, if the gap problem with the threshold values $(s, c)$ is $\mathcal{NP}$-hard to decide, the corresponding optimization problem does not admit a polynomial-time $\frac{c}{s}$-approximation algorithm (under some reasonable assumption such as $\mathcal{P} \neq \mathcal{NP}$). In the study of inapproximability, many gap problems were proven to be $\mathcal{NP}$-hard and gap-preserving reductions are used to propagate the $\mathcal{NP}$-hardness of gap problems.

## 2.6 Exact Algorithms

There are many situations where we need to calculate an exact solution, or the approximation guarantees are insufficient, or there even might not exist any (under reasonable assumptions) for certain hard problems. In all this cases, the design of the exact algorithms is one of the ways to tackle such hard problems.

As we currently do not know whether there are algorithms that optimally solve $\mathcal{NP}$-complete problems in polynomial time, exact algorithms must account for this and thus they usually need an exponential time in the size of the input. This is also supported by the Exponential Time Hypothesis (ETH) that claims that $\mathcal{NP}$-complete problems cannot be exactly solved in subexponential time. The ETH was introduced as a stronger conjecture than $\mathcal{P} \neq \mathcal{NP}$ to prove lower bounds we are not able to prove under milder assumption $\mathcal{P} \neq \mathcal{NP}$.

The assumption of ETH carries over to many optimization problems through their $\mathcal{NP}$-complete decision versions. Furthermore, there is an entire theory evolving (see, e.g., [71]) for proving the unlikelihood of an existence of subexponential algorithms for many optimization problems (unless ETH fails). In this theory, the so-called SERF-reduction is introduced to propagate this unlikelihood further on problems. Most of the standard reductions between $\mathcal{NP}$-complete problems can be easily modified to become SERF-reductions and thus famous problems
such as minimum vertex cover and minimum set cover do not admit subexponential exact algorithms, unless ETH fails. Hence, the realistic goal in the design of exact algorithms is to design faster and faster (but still exponential) algorithms.

To denote the time complexity of an exact algorithm, we use the so-called $O^*$-notation (see Section 2.1). The notation is similar to the standard $O$-notation, but instead of the most significant polynomial factors, we emphasize the most significant exponential factors and neglect all polynomial factors.

There are several different methods and techniques in the design of exact algorithms either for pruning the branching and/or for improving the estimation of the overall time complexity. The recent books of Fomin and Kratsch [54] and Gaspers [57] nicely survey these techniques.

Our results presented in Chapter 7 are based on the branch-and-reduce technique that is discussed in more detail in the remainder of this section.

### 2.6.1 Branch-and-Reduce Technique

Branching is one of the basic algorithmic techniques used in the design of fast exponential-time algorithms. For many NP-hard problems, branching algorithms are often the fastest known algorithms. Compared to other techniques, this method has some nice properties: a typical algorithm needs only polynomial space, on many instances it is extremely fast, and it allows natural improvements to certain inputs so that its running time is speeded up, even though the worst-case time complexity stays unchanged. These properties make branching algorithms popular and widely used in real-world applications, where data may often possess nice attributes that the general instances do not have and thus the algorithm is notably faster than the expected worst case.

In our thesis, we refer to algorithms based on branch-and-reduce paradigm as branching algorithms. In general, the branching algorithm is applied on a problem instance recursively and uses two types of rules:

- A **reduction rule** is used to simplify the instance or to halt the algorithm and is often performed in polynomial time. The solution of the reduced instance can be extended back to the original instance in polynomial time.

- A **branching rule** is used to solve problem instance by recursively solving the smaller instances. These rules are the most expensive ones and are the main cause of the exponential time complexity of the algorithm.

Of course, the rules usually cannot be applied in arbitrary order and we have to carefully choose their order and the types of instances which they should be applied to. This is the crucial part in the design of branching algorithms. A typical example of such an ordering are preference rules such as “apply the rule on the minimum/maximum degree vertex of the instance”.

The correctness of the algorithm follows from the rules and thus we often have to prove their correctness. A typical argument often used is that, even though our reductions may overlook some optimal solutions, there is still at least one that is preserved and later found by our algorithm.
2.7 Parameterized Complexity Theory

The worst-case time complexity of a branching algorithm is often estimated through branching rules. A standard branching algorithm performs both the reduction and the branching rules in polynomial time and thus, de facto, only the size of the branching comes to the game. Its size can be estimated from the so-called search tree describing the course of algorithm’s branching on the instance. To bound the size of such tree, one builds up a bunch of recurrences, each describing the time complexity of a computation where the algorithm recursively computes a solution on the smaller instances that are given by one branching rule. The system of such recurrences is then solved and the exponential worst-case running time is obtained. There are some more advanced techniques such as the measure and conquer technique (see, e.g., [54] for details) that may be helpful for a finer estimation of the worst case.

We give a flavor of the branching technique by the following simple algorithm that solves the minimum vertex cover problem. We have only one reduction rule – an isolated vertex is removed and not inserted into the covering set. In our only branching rule we pick an edge $(u, v)$ of the graph with $n$ vertices. We branch and consider the two possibilities:

- either $u$ is in the covering set and thus we remove it with all its incident edges from the graph resulting into a graph with $n - 1$ vertices,
- or $u$ is not and thus $v$ must be in the covering set. In this case, we add all the neighboring vertices of $u$ to the covering set as well and we remove $u$, $v$ and all vertices from $N(u)$ from the graph which results in a graph with at most $n - 2$ vertices.

In either case, we continue on a smaller graph. From all the found solutions, we pick the one with the minimum covering set. Our algorithm always finds an exact and minimum solution as, in each step, we consider all the possibilities how an edge can be covered and we do not remove any uncovered edge or vertex without determining its covering-set membership from the graph. Our recurrence, in the worst-case (when the reduction rule cannot be applied), is $T(n) = T(n - 1) + T(n - 2) – we always solve instances by at least one and two vertices smaller. The recurrence corresponds to the recurrence of the Fibonacci numbers (see, e.g., [60] for details) and hence $T(n) = \mathcal{O}^*((1+\sqrt{5})^n) = \mathcal{O}^*(1.61804^n)$.

2.7 Parameterized Complexity Theory

After two decades of research on hard problems, researchers were not completely satisfied with their studies. Most of the problems were either in $\mathcal{P}$ or $\mathcal{NP}$, many of them were $\mathcal{NP}$-hard. The study of approximability also failed for many important hard problems – the problems were either inapproximable or at best approximable by some very impractically high ratio. This desire for better understanding of the kernel of hardness of such problems gave birth to the parameterized complexity theory that was first introduced by Downey and Fellows in [45–47].
Chapter 2. Basic Notions and Preliminaries

In this field, the hard problem is sliced into subproblems by allowing parameters to come into the game. The problem is then studied according to its inherent difficulty with respect to one or multiple parameters. The parameterization of the problem by a natural parameter allows us to study the problem to a finer grade and to better capture the hard classes of its instances. Of course, the complexity of the problem is then measured as a function not only of the input size, but also of the parameters.

Any decision or optimization problem can be changed into a parameterized version by introducing one or more parameters into the input and possibly modifying the goal accordingly to reflect these parameters. The parameterized problems can be parameterized by properties of the input and/or the output. For instance, in parameterization of the input, one can assume that the input is a graph of bounded degree, bounded treewidth, or with some other special structural property. A very frequent parameterization of the output is to aim for a solution with the cost bounded by the parameter.

Our thesis is not focusing on this field and we come across some results in parameterized complexity theory only as direct corollaries of our main results. The aim of this section is to sketch the very basics to the extent sufficient to understand our results. For more details related to the topic, we would like to refer to the monography of Downey and Fellows [47] and more recent books by Flum and Grohe [53] and Niedermeier [87], from which our terminology is mainly adapted.

**Definition 2.7.1.** A parameterization of a set $S$ is a mapping $\kappa: S \to \mathbb{N}$ that is computable in polynomial time.

Informally, the parameterization of a set partitions its elements into groups according to some property that is captured by the parameterization. The image of the parameterization is interpreted as the value of the parameter for the originating element. For a simple example, let $G$ be the set of all possible graphs. We can parameterize $G$ by the number of colors by which its members are colorable, by the maximum vertex-degree, by the size of the maximum matching of each member graph, etc.

We define the parameterized problem generally for all problems.

**Definition 2.7.2.** A parameterized problem of a problem $P$ with decidable set of input instances is a problem $(P, \kappa)$, where $\kappa$ is a parameterization of the set of all feasible inputs for the problem $P$.

If the parameterized problem is a decision problem, we call it a parameterized decision problem.

A prominent class of the problems in parameterized complexity hierarchy that is often considered to be the parameterized counterpart of the class $\mathcal{P}$ is the class $\mathcal{FPT}$. It contains all the fixed-parameter tractable problems defined as follows.

**Definition 2.7.3.** A parameterized problem $(P, \kappa)$ is a fixed-parameter tractable problem ($\mathcal{FPT}$-problem) if
2.7. Parameterized Complexity Theory

i) there exists an algorithm \( A \) that solves \((P,\kappa)\),

ii) there exists a polynomial function \( p \) and an arbitrary computable function \( f \) such that the running time of \( A \) on any problem instance \( x \) of \( P \) is bounded by \( f(\kappa(x)) \cdot p(|x|) \).

Similarly as in the hierarchy of the decision and optimization problems, there exists a parameterized reduction that allows to reduce parameterized problems back and forth and hence to prove that certain parameterized problems are at least as hard as some other parameterized problems.

Not surprisingly, many parameterized problems do not admit a fixed-parameter tractable algorithm, unless \( P = \mathcal{NP} \). To reflect similar results, a complexity theory on parameterized problems was developed. The counterpart of \( \mathcal{NP} \) in this theory is the so-called class \( \mathcal{W}[1] \). The hierarchy continues by more and more general classes \( \mathcal{W}[2], \mathcal{W}[3], \ldots \), and the class \( \mathcal{X}\mathcal{P} \) that is above all the \( \mathcal{W} \) classes. These classes are structured into a hierarchy

\[
\mathcal{FPT} \subseteq \mathcal{W}[1] \subseteq \mathcal{W}[2] \subseteq \cdots \subseteq \mathcal{X}\mathcal{P}
\]

in which it is believed that all the subsets are proper.

Since parameterized complexity theory is not in the focus of our thesis and the results that we have come across are not very deep, we decided not to formally define any of the aforementioned classes. For this thesis, it is sufficient to be aware of the existence of the hierarchy and to have an intuitive idea about the hardness and the complexity of the member problems of the first few basic classes.

The hardness and completeness of the problems of the parameterized classes is defined similarly as for decision/optimization problems, but, as a reduction, the parameterized reduction has to be used.

The last concept of parameterized complexity theory that we would like to introduce is the notion of the \textit{kernel} of a parameterized problem and the \textit{kernelization technique}. We define both notions only for parameterized decision problems, since we would need additional formalism to generalize the notion of “yes-instances” in decision problems to general problems. However, the reader should get a flavor of kernelization from our definition.

\textbf{Definition 2.7.4.} Let \((P,\kappa)\) be a parameterized decision problem, let \( I \) be a set of feasible problem instances of \((P,\kappa)\) and let \( I^+ \) denote its set of yes-instances.

A polynomial-time computable function \( K : I \rightarrow I \) is a kernelization of \((P,\kappa)\) if there exists a computable function \( f : \mathbb{N} \rightarrow \mathbb{N} \) such that, for all problem instances \( x \in I \), such that, for \( k = \kappa(x) \),

i) The size of the problem instance \( K(x) \) is bounded from above by \( f(k) \),

ii) The parameter \( \kappa(K(x)) \) is bounded from above by \( k \), and

iii) \( x \) is a yes-instance of \((P,\kappa)\) if and only if \( K(x) \) is a yes-instance of \((P,\kappa)\).

For every instance \( x \) of \((P,\kappa)\), the image \( K(x) \) is called a kernel of \( x \) (under the kernelization \( K \)).


One can view the kernelization as a set of reduction rules that are employed to reduce an instance in polynomial time. Hence, in the backward process, one can transform the instance back and extend the built solution directly without need of examining too many possibilities.

For a fixed parameter, the parameterized problem usually contains infinitely many (very large) problem instances. But the kernel’s size is just a function of the parameter and hence it is constant if the parameter is fixed. Hence, the kernelization reduces a huge number of instances to a constant size kernel that is the hardest part of the problem.

The kernelization is a core tool that can be used to design \textit{FPT}-algorithms as follows. For an instance $x$, first apply the kernelization to reduce $x$ to a smaller instance $y$ with size dependent on the parameters, but not on $|x|$. The transformation is performed in polynomial time. Then consider all possible solutions of $y$ and find the best one. The time complexity of this part is dependent only on the parameters, but not on $|x|$. As a last step, after the best solution is found, perform the transformation backwards and extend the best found solution to a solution for $x$. Again, this part is performed in time polynomial in $|x|$. Overall, the time complexity of the whole process is polynomial in $|x|$, but may be superpolynomial in the parameters.

We conclude this section by the following theorem, which relates the class \textit{FPT} and kernelization. It is again stated only for parameterized decision problems, but it holds for general parameterized problems as well. The proof is omitted, but its version for general parameterized problems can be found, for instance, in [53].

\textbf{Theorem 2.7.5.} For every parameterized decision problem $(P, \kappa)$, the following are equivalent:

i) $(P, \kappa) \in \text{FPT}$,

ii) the set of feasible input instances of $P$ is decidable and $(P, \kappa)$ has a kernelization.

\section{Overview of Some Well-Known Problems}

In this section, we give an overview of some well-known problems that are mentioned or used in this thesis. We summarize the known results and we discuss a few of these results in more detail where we think it may help to understand the following chapters.

\subsection{Variants of the Traveling Salesman Problem}

Many relevant problems in operations research can be modeled by the famous Traveling Salesman Problem (TSP). It is notoriously hard in many settings and belongs, together with its many modifications, to the most studied hard problems. A good overview of the TSP can be found in [63].
Problem 2.8.1. Traveling Salesman Problem (TSP):

**Input:** A complete graph $G = (V, E)$ and an edge cost function $c: E \to \mathbb{Q}^+$. 

**Feasible solutions:** A Hamiltonian cycle $C$ in $G$. 

**Costs:** Cost of the edges in $C$, i.e., $\text{cost}(C) = \sum_{e \in E(C)} c(e)$. 

**Goal:** Minimization.

It is a well-known fact that the general TSP is $\text{NP}$-complete [98], i.e., is not approximable with any polynomial ratio. This brings us to a study of the hardness of its variations and modifications through which we can better understand the hard kernel of the problem and also find those of its subclasses that are still practically relevant and easier to approximate.

One of TSP’s natural generalizations is TSP with vertex precedence constraints on the resulting cycle (see, e.g., [32]). Such a requirement can be represented by any partial ordering on the vertices. However, we focus here on the case where the ordering of vertices is linear, i.e., we are given a sequence of vertices that need to be visited in $C$ in this order. Formally, the problem is defined as follows.

Problem 2.8.2. $k$-ordered Traveling Salesman Problem ($k$-OTSP):

**Input:** A complete graph $G = (V, E)$, an edge cost function $c: E \to \mathbb{Q}^+$ and a sequence $T = (s_1, \ldots, s_k)$ of special vertices from $V$. 

**Feasible solutions:** A Hamiltonian cycle $C$ on $G$ that contains the vertices from $T$ in the given order. 

**Costs:** Cost of the edges in $C$, i.e., $\text{cost}(C) = \sum_{e \in E(C)} c(e)$. 

**Goal:** Minimization.

There are other variants of TSP with requirements on the order in which the vertices have to be visited which we do not study in this thesis. In time-window TSP, each vertex is related with a time window in which it has to be visited (see [15]). A less strict variant is the so-called deadline TSP in which each vertex is associated with a time until when it has to be visited in any feasible solution (see [15, 24]). In these models, by time we mean the sum of the edge costs from some specified start vertex.

Another variant of TSP, that is studied in Chapter 4, is the Steiner cycle problem (SCP). The name of the problem is derived from two problems from which the problem inherits its goals – the Steiner tree problem and the Traveling Salesman Problem. The aim in the Steiner tree problem on graphs is to find a set of edges of minimum overall cost that connects a set of given special vertices by a tree (see details in Subsection 2.8.2). Hence, its feasible solutions do not have to use all the vertices of the graph. Similarly for cycles, SCP aims to find a cycle in a graph that visits all the given special vertices and its cost is minimum, i.e., the non-special vertices do not have to be present in a feasible cycle.
Orthogonal to the structural requirements in TSP-like problems are restrictions on the cost function. We can assume that the cost function is not arbitrary but respects some kind of metric. This is not such an unreasonable assumption, many measures in the real world are keeping some kind of metric (e.g., distances between points on a map). As we will see further, once we simplify the TSP to a subclass of instances with a certain metric, this restriction of the \(N\mathcal{P}O\)-complete problem becomes a member of \(APX\).

In the following subsections, we present some approximation algorithms for TSP and ordered TSP that are either modified or used in the algorithms we design in Chapter 3 and Chapter 4 of this thesis. Before giving an overview of the known results, we give a summary of the properties of the standard and the \(\beta\)-relaxed triangle inequality. Although these properties are not related to any particular problem (e.g., TSP), we find this place suitable to discuss them, as our thesis uses metric costs only when TSP or one of its modifications is considered.

**Restricting the Edge Costs by the Triangle Inequality**

The assumption that the cost of the edges in a graph satisfies a certain metric is a good model of many real-world situations in which the metric is inherent by the nature of the problem. This subsection is devoted to important properties of the metrics we assume. Formally, let \(G = (V,E)\) be an undirected complete graph and let \(c: E \rightarrow \mathbb{Q}^+\) be a cost function of its edges (if the graph is not complete, we can add the missing edges with infinite cost).

First, assume that the standard triangle inequality holds for the edge cost function, i.e., that, for all \(u,v,w \in V\),

\[c(uv) \leq c(uw) + c(wv).\]

In such a case, we see that a single edge cannot be more expensive than any bypass, i.e., using the direct edge \(uv\) cannot cost more than connecting vertices \(u\) and \(v\) through some intermediate vertex \(w\). What does this mean for our approximation algorithms? Imagine that we have a tour \(T\) that contains all the vertices of \(V\). We somehow need to modify it to obtain a cycle \(C\) that will still contain all the vertices of \(V\) and such that \(c(C) \leq c(T)\). Such cycle \(C\) is a Hamiltonian cycle in \(G\). Due to the standard triangle inequality, once we have \(T\), the task is simple. We start a walk on \(T\) and whenever we would visit a vertex \(v\) a second time, we just bypass it, i.e., we remove it from the tour, obtaining a smaller tour \(T'\). Let \(u, w\) denote the neighboring vertices before and after \(v\) in our walk in \(T\), i.e., \(uw \in E(T)\) and \(vw \in E(T)\). The only modification of \(T\) to \(T'\) was the bypass of \(v\), thus \(E(T) = E(T') \setminus \{uw\} \cup \{uv,vw\}\). Now we see that, since the triangle inequality holds, we have

\[c(T') = c(T) - c(uv) - c(vw) + c(uw) \leq c(T) - c(uv) - c(vw) + c(uw) + c(vw) = c(T).\]
The approach that we have just described is one of the basic techniques in the design of approximation algorithms on problems where the standard triangle inequality holds. Whenever considering graph problems where one is required to visit certain vertices only a limited number of times while minimizing the overall edge cost used for these visits, we can always try to build a structure that satisfies all the requirements, except that vertices may be visited more often than allowed, and then try to “shorten” it by bypassing vertices that are visited too often. We will refer to this technique as a “shortening” technique.

Assume now that the $\beta$-relaxed triangle inequality holds for the edge cost function in $G$, i.e., that, for all $u,v,w \in V$ and for some $\beta > 1$,

$$c(uv) \leq \beta \cdot (c(uw) + c(wv)).$$

Observe that direct edges may be more expensive than bypasses here. Thus, we cannot just bypass inconvenient vertices in the structures without possible increase of the cost of the structure. However, the following lemma shows that the cost of a solution with several consecutive vertices bypassed can be still estimated, but increases by an exponential factor dependent on $\beta$ and the number of bypassed consecutive vertices.

**Lemma 2.8.3** (Bandelt et al., [14]). Let $G = (V,E)$ be a graph with edge cost function $c$ that satisfies the $\beta$-relaxed triangle inequality. Let $W = v_0, \ldots, v_\ell$ be a walk in $G$. For $0 = a_0 < a_1 < \cdots < a_q = \ell$, where $1 \leq q \leq \ell$ holds, let $m := \max_{0 \leq i < q} \{ a_{i+1} - a_i \}$. Then

$$\sum_{i=0}^{q-1} c(v_{a_i}, v_{a_{i+1}}) \leq \beta \log_2 m \cdot c(W).$$

**Sketch of the proof.** In Figure 2.1 we have depicted a complete graph with maximal cost on edges. More precisely, the labels correspond to the maximal multiplier of the edge-cost with respect to the underlying part of the walk. For example, for each three consecutive vertices $v_i, v_{i+1}, v_{i+2}$ in $W$, the cost of the direct edge $v_iv_{i+2}$ is at most $\beta$ times more expensive than the sum of the cost of the two edges $v_iv_{i+1}$ and $v_{i+1}v_{i+2}$. Hence, we can apply such relation over and over to longer and longer consecutive bypasses and in the end we obtain the claim of our lemma.

For directed graphs, we can define the standard and $\beta$-relaxed triangle inequality for the edge cost function similarly, we just have to use the directed edges, i.e., for all $u,v,w \in V$

$$c(\overrightarrow{uv}) \leq c(\overrightarrow{uw}) + c(\overrightarrow{wv}).$$

All we have discussed here before for the triangle inequality on undirected graphs can be easily transformed to the case of directed graphs – just the correct direction on the edges must be preserved.
Chapter 2. Basic Notions and Preliminaries

Figure 2.1: A complete graph with maximal edge costs that still satisfies the $\beta$-relaxed triangle inequality. The function of $\beta$ on edges corresponds to the maximal multiplier by which the cost of the bypass can increase with respect to the cost of the underlying part of the walk in the walk $v_0v_2^k$.

Approximation Algorithms for TSP

To shorten the notation, the TSP with the standard triangle inequality we call metric TSP and denote it by $\Delta$-TSP, the version with relaxed triangle inequality is called near-metric TSP and is denoted as $\Delta_\beta$-TSP. First we assume that the standard triangle inequality holds for the cost functions, i.e., we are considering $\Delta$-TSP. The TSP in this setting belongs to one of the oldest optimization problems studied in the modern history – it was defined by Hamilton in the 19th century.

We start by presenting two approximation algorithms for $\Delta$-TSP on undirected graphs. The first is a very simple 2-approximation algorithm. Its origins are unknown, but one of the earliest works where it was extensively studied is the paper by Rosenkrantz, Stearns and Lewis [97].

Algorithm 1 (TSP 2-approximation)

Input: A complete undirected graph $G = (V,E)$ with edge cost function $c: E \to \mathbb{Q}^+$ that satisfies the standard triangle inequality.
1: Compute a minimum spanning tree $T$ in $G$ according to $c$.
2: Choose an arbitrary vertex $v \in V$ and perform a depth-first search of $T$ from $v$.
   Order the vertices in the order of visit in the search into a tour $C$.
3: Shorten the tour $E$ to a Hamiltonian cycle $H$.
Output: The Hamiltonian cycle $H$.

Theorem 2.8.4 (Rosenkrantz et al., [97]). Algorithm 1 is a polynomial-time 2-approximation algorithm for $\Delta$-TSP.

Proof. Let us denote by $OPT$ an arbitrary optimal solution for our given instance of $\Delta$-TSP.
Observe that, by removing the most expensive edge from a Hamiltonian cycle, we obtain a very special spanning tree – a path. Hence the minimum spanning tree, that is the cheapest out of all such trees, cannot be more expensive than the optimal solution, i.e., $\text{cost}(T) \leq \text{cost}(\text{OPT})$. The depth-first search visits every vertex at most twice and traverses every edge twice (once when a vertex is discovered and once when returning). Therefore, $\text{cost}(C) = 2 \cdot \text{cost}(T) \leq 2 \cdot \text{cost}(\text{OPT})$. The tour $T$ contains every vertex at most twice and thus by the shortening argument, the shortened cycle $H$ costs $\text{cost}(H) \leq \text{cost}(C) \leq 2 \cdot \text{cost}(\text{OPT}).$

The best known approximation algorithm for $\Delta$-TSP is due to Christofides [37]. Nobody was able to improve the ratio in the last 35 years, even though it is conjectured that the problem should admit a $4/3$ approximation. Hence, an improvement over this ratio by only a very little would be a tremendous result for the community.

**Algorithm 2** (Christofides’ algorithm)

**Input:** A complete undirected graph $G = (V, E)$ with edge cost function $c: E \to \mathbb{Q}^+$ that satisfies the standard triangle inequality.
1: Compute a minimum spanning tree $T$ in $G$.
2: Compute a minimum perfect matching $M$ on the odd vertices of $T$ in $G$.
3: Compute an Eulerian tour $C$ in $(V, M \cup E(T))$.
4: Shorten the Eulerian tour $C$ to a Hamiltonian cycle $H$.

**Output:** The Hamiltonian cycle $H$.

**Theorem 2.8.5** (Christofides, [37]). Algorithm 2 is a polynomial-time 1.5-approximation algorithm for $\Delta$-TSP.

**Proof.** Let us denote by $\text{OPT}$ an arbitrary optimal solution for our given instance of $\Delta$-TSP and let $W$ be the set of odd vertices of $T$.

Similarly as in the proof of Theorem 2.8.4, $\text{cost}(T) \leq \text{cost}(\text{OPT})$. Moreover, we can shorten $\text{OPT}$ to a cycle that contains only the vertices of $W$. Such a cycle contains two perfect matchings (we always pick every second edge to the matching) and thus one of these matchings cannot cost more than half of the overall cost of the cycle. Hence, since $M$ is a minimum perfect matching, $\text{cost}(M) \leq \text{cost}(\text{OPT})/2$, which implies that $\text{cost}(C) \leq 3/2 \cdot \text{cost}(\text{OPT})$. As shortening $C$ cannot increase the cost of $H$, the claim is proved.

The situation is more difficult in the directed TSP, also known as asymmetric TSP. For a long time, scientists are improving the logarithmic approximation ratio of directed $\Delta$-TSP little by little to the currently best known approximation ratio of $2/3 \log_2 |V|$. This is in contrast with the $\Delta$-TSP on undirected graphs, where an algorithm with a constant approximation ratio is known for many years.

**Theorem 2.8.6** (Feige and Singh, [51]). There is a polynomial-time $\left(\frac{2}{3} \log_2 n\right)$-approximation algorithm for $\Delta$-TSP on directed graphs with $n$ vertices.
Further on, we continue with the analysis of $\Delta_\beta$-TSP. There are currently three different approximation algorithms with ratios $\frac{3}{2}\beta^2$, $\beta^2 + \beta$ and $4\beta$ (see [6, 17, 22]) for undirected graphs. They should be chosen with respect to the $\beta$. The $\frac{3}{2}\beta^2$-approximation algorithm is the best for $\beta \leq 2$, the second algorithm is best for larger values of $\beta$ until $\beta \leq 3$ and the $4\beta$-approximation algorithm is the currently best known approximation algorithm for $\beta \geq 3$. The algorithms with ratios $\frac{3}{2}\beta^2$ and $4\beta$ we just mention as theorems as they are quite complex. The third one we analyze in more detail as our algorithms from Section 3.2 and Subsection 4.3.2 employ it.

**Theorem 2.8.7** (Bender and Chekuri, [17]). There exists a polynomial-time $4\beta$-approximation algorithm for $\Delta_\beta$-TSP.

**Theorem 2.8.8** (Böckenhauer et al., [22]). There exists a polynomial-time $\frac{3}{2}\beta^2$-approximation algorithm for $\Delta_\beta$-TSP.

Before presenting the algorithm of Andreae [6], we need to mention a simple definition.

**Definition 2.8.9.** Let $G$ be a graph with edge cost function $c$. An edge $e^* \in E(G)$ is called a locally minimal edge of $(G, c)$ if there exists one endpoint $x$ of $e^*$ such that the cost of $e^*$ does not exceed the cost of all other edges incident to $x$, i.e., $c(e^*) \leq c(xy)$, for all $y \in N(x)$.

Now we are ready to bound the cost of the outcome of Algorithm 3.

**Theorem 2.8.10** (Andreae, [6]). Algorithm 3 applied on a tree $T$ computes a $\Delta_\beta$-TSP solution $H$ on vertices $V(T)$ in time $O(|V(T)|^2)$ and with the cost

$$
\text{cost}(H) \leq (\beta^2 + \beta)\text{cost}(T).
$$

Hence, Algorithm 3 is a $(\beta^2 + \beta)$-approximation algorithm for $\Delta_\beta$-TSP with time complexity of $O(|V(T)|^2)$.

**Sketch of the proof.** It is easy to argue that the algorithm always produces a Hamiltonian cycle on $V(T)$. The proof can be done by induction on the size of the tree. The base case for $|V(T)| = 3$ is trivial. In the inductive step, where the two smaller Hamiltonian cycles are produced, the edge $(a_i, a'_i)$ is removed from each of them and the remaining two paths are merged on their endpoints by $e^* = (a_1, a_2)$ and $(a'_1, a'_2)$. Obviously, the produced graph is a cycle, no vertex is present here more than once and no vertex is forgotten to be added.

The proof of the approximation ratio of the algorithm is rather complex. Hence, we point the reader to the original paper for details.

The time-complexity of the algorithm is quadratic in the number of vertices as, by each division of the tree, we obtain two trees of size smaller by at least one vertex. Each call of the procedure takes $O(|V(T)|)$ time and the recursion is terminated after $O(|V(T)|)$ calls.
Algorithm 3 (HCT\textsuperscript{3} (refined) by Andreae, [6])

**Input:** A complete undirected graph $G = (V, E)$ with edge cost function $c: E \rightarrow \mathbb{Q}^+$ that satisfies the $\beta$-relaxed triangle inequality ($\beta > 1$), a tree $T$ with $|V(T)| \geq 3$ and a locally minimal edge $e^* = (a_1, a_2)$ of $T$.

1. For $i \in \{1, 2\}$, let $T_i$ be the component of $(V(T), E(T) \setminus \{e^*\})$ containing $a_i$.
2. for $i = 1$ to 2 do
3. if $|V(T_i)| \geq 2$ then
4. pick $a'_i \in V(T_i)$ such that $(a_i, a'_i)$ is a locally minimal edge in $T_i$.
5. let $e^*_i := (a_i, a'_i)$.
6. else
7. let $a'_i := a_i$.
8. end if
9. if $|V(T_i)| \geq 3$ then
10. recursively compute a Hamiltonian cycle $H_i$ of $V(T_i)$ using $e^*_i$.
11. let $P_i := H_i \setminus \{e^*_i\}$.
12. else
13. let $P_i := T_i$.
14. end if
15. end for
16. Merge paths $P_1, P_2$ and edges $e^*$ and $(a'_1, a'_2)$ into a cycle $H$.

**Output:** The Hamiltonian cycle $H$ containing $e^*$.

We would like to point out that relating the costs of the tree and the Hamiltonian cycle is crucial for our algorithms from Chapters 3 and 4. The property pays off in the approximation factor analysis, where it allows us to reasonably estimate the cost of our partial solution by the cost of the underlying subtree which is cheap. The other two algorithms for $\Delta_\beta$-TSP do not possess this property and thus the cost of the computed partial solution would be estimated by the cost of the optimal Hamiltonian cycle on the underlying subgraph which might be significantly more expensive than the cost of the optimal solution for our problems.

The problem seems to be again harder for the directed graphs. We are not aware of any approximation algorithm for directed TSP with $\beta$-relaxed triangle inequality.

**Approximation Algorithms for Ordered TSP**

Since OTSP is a generalization of TSP, it is at least as hard and thus, without an assumption on the cost function, it is $\mathcal{NP}$-hard. Hence, we define its subproblems where a metric is assumed. In the entire subsection we assume that all graphs are undirected, as we do not study the directed version of the problem.

We call the $k$-OTSP with standard triangle inequality the metric $k$-OTSP and denote it as $k$-\textDeltaOTSP. When the $\beta$-relaxed triangle inequality holds for the cost function, the corresponding $k$-OTSP is called near-metric $k$-OTSP and denoted as $k$-\textDeltabetaOTSP.
As we will see in the following theorem, \(k\)-\(\Delta\)OTSP can be approximated by a constant that is independent of \(k\). Note that \(k\)-\(\Delta\)OTSP for \(k \leq 3\) is the same as \(\Delta\)-TSP – any Hamiltonian cycle keeps the special vertices in the prescribed order. Hence, the Christofides’ algorithm is the currently best known approximation algorithm for these instances.

**Theorem 2.8.11** (Böckenhauer et al., [23]). For any \(k \geq 4\), there is a polynomial-time 2.5-approximation algorithm for \(k\)-\(\Delta\)OTSP.

**Sketch of the proof.** The general idea of the algorithm is simple: Connect the special vertices into a cycle \(D\). Its cost cannot be more expensive than the cost of the optimal solution as we can shorten it to the cycle \(D\). Using Christofides’ algorithm, connect all the non-special vertices and one special vertex \(s\) into a Hamiltonian cycle \(C\). Its cost cannot be higher than 1.5 times the cost of the optimal solution as Christofides’ algorithm finds an 1.5-approximation for the cheapest Hamiltonian cycle that does not have to satisfy any extra requirements on order of the vertices. Merge the two cycles \(C\) and \(D\) in vertex \(s\) and bypass vertex \(s\) by the direct edge between its neighbor in \(C\) and its neighbor in \(D\).

We would like to note that the result was improved in [25] and the currently best approximation ratio was decreased to \(2.5 - 2/k\) which suggests that the ratio 2.5 is not tight. The algorithm achieving the better ratio is quite involved and thus we decided to present the simpler but a little bit worse one.

Now we present the approximation algorithm for \(k\)-\(\Delta\beta\)OTSP that was published in [23] and was the best known until we improved it. In Section 3.2 we design a new algorithm with a better approximation ratio for \(k\)-\(\Delta\beta\)OTSP and in Section 3.3 we discuss improvements of the following algorithm that are improving the algorithm of Section 3.2 for values of \(\beta\) close to one. We would like to note that our presentation of the algorithm slightly differs from the one of [23]. In the original presentation, the subpaths \(H_i\) are concatenated into a Hamiltonian cycle \(H\), i.e., the subpaths do not share the endpoints and they are joined by an edge. We find it more convenient to include these edges into the subpaths and hence, the algorithm is presented with the subpaths \(H_i\) that share the endpoints and are merged into \(H\).

The general idea of Algorithm 4 is the following one. First, we connect all vertices into a Hamiltonian cycle. Secondly, we color the non-special vertices cyclically by \(k + 1\) colors, so we can use the bypassing argument, i.e., we can traverse the cycle including vertices of one color to our structures and bypassing the rest and guarantee that the cost of such cycle will not increase too much. Then we connect pairs of the consecutive special vertices by traversing the Hamiltonian cycle and by including the vertices of just one fixed color (for each pair of vertices a different one) in between the two special vertices to the path. This way, for each pair \((s_i, s_{i+1})\), \(1 \leq i \leq k - 1\), we create a subpath \(H_{i-1}\). Another subpath \(H_{k-1}\) is used to connect the special vertex \(s_k\) with a vertex \(x\) adjacent to \(s_1\) via vertices of color \(k\). The last subpath \(H_k\) then starts in the vertex \(x\) and traverses the entire cycle to the vertex \(s_1\) and uses the remaining vertices not yet included
to the subpaths. Hence, once all the subpaths are connected, we obtain a feasible solution as follows. Start in \( s_1 \) by \( H_0 \) use sequentially subpaths \( H_1, \ldots, H_{k-2} \) to visit all the special vertices in required order, continue the traversal by subpath \( H_{k-1} \) to reach the vertices that were not bypassed at this point and then by the last subpath \( H_k \) traverse the entire cycle, return back to \( s_1 \) and visit all the remaining vertices not yet included to the structure. Indeed, the structure is a Hamiltonian cycle that contains the special vertices in the prescribed order.

**Theorem 2.8.12** (Böckenhauer et al., [23]). Algorithm 4 is a polynomial-time \((k-\Delta_\beta)\)OTSP approximation algorithm that solves \( k-\Delta_\beta \)OTSP.

**Proof.** Observe that the subpaths are all edge-disjoint but they share the same endpoints, i.e., the endpoints of \( H_i \) and \( H_{(i+1) \mod (k+1)} \) are identical, for all \( 0 \leq i \leq k \). Moreover, they share the property that they result from a subpath of \( P \) by removing some vertices and the number of consecutive removed vertices is bounded.

More precisely, since we used a cyclic coloring of vertices in \( C \), any two neighboring vertices in \( H_i \) (for all \( 0 \leq i \leq k \)) have a distance at most \( 2k - 1 \) in \( P \). This
is true since, between two such vertices, there are at most $k$ non-special vertices colored by all the other colors and at most $k - 2$ special vertices (the remaining two special vertices are located in the start and endpoint of the subpath $H_i$). Hence, by Lemma 2.8.3, we have

$$\text{cost}(H_i) \leq \beta^{1+\log_2(k-1)} \cdot \text{cost}(C).$$

Since $H$ is just joining of all the subpaths $H_i$ (for all $0 \leq i \leq k$), we have

$$\text{cost}(H) \leq (k + 1) \cdot \beta^{1+\log_2(k-1)} \cdot \text{cost}(C).$$

Consequently, in step 1 of the algorithm, we use an approximation algorithm for $\Delta_\beta$-TSP which influences the overall ratio. The currently best known approximation algorithms from [6, 17, 22] (see Theorems 2.8.7, 2.8.8 and 2.8.10 for the details) achieve the approximation ratio of

$$\min \left\{ \frac{3}{2} \beta^2, \beta^2 + \beta, 4\beta \right\}$$

which gives us the claimed ratio. \qed

### 2.8.2 The Minimum Steiner Tree Problem

This subsection is devoted to a short summary of the results of the well-known Steiner tree problem.

**Problem 2.8.13.** Minimum Steiner Tree Problem (STP):

**Input:** A connected graph $G = (V, E)$, an edge cost function $c: E \to \mathbb{Q}^+$ and a set $K \subseteq V$ of special vertices, called terminals.

**Feasible solutions:** A tree $T$ that contains all the vertices of $K$.

**Costs:** Cost of the edges in $T$, i.e., $\text{cost}(T) = \sum_{e \in E(T)} c(e)$.

**Goal:** Minimization.

Probably the first study of a problem close to STP dates back to Pierre de Fermat [34] who studied the problem for three points in a plane: For a given coordinates of triangle vertices, one is asked to find the point with minimum total distance to the vertices. Fermat also studied the generalized version of the problem with many points.

The generalization of Fermat’s problem to the geometric version of STP in Euclidean plane was first studied by Carl Friedrich Gauss [34]. The geometric version uses the distance function as the cost function, omits the prescribed vertices $V \setminus K$ that can be used as intermediate vertices of the graph and can be formulated as follows. Given are the coordinates of cities in the plane, the goal is to build roads of minimum total length so that any pair of cities is connected by a path. Hence,
one is allowed to build new crosspoints where roads meet in order to minimize the overall length.

Recall that, the minimum spanning tree (MST) of a graph is a tree that contains all the vertices and the overall cost of its edges is minimum. Finding a minimum spanning tree in a graph is not a hard problem, there exist several different algorithms with time-complexity polynomial in the size of the graph (see, e.g., [39] for details). The STP in the graphs differs from the MST problem only in the set of the special vertices; the MST problem is the graph STP with $K = V$. As we will see, STP is a hard problem. This is one of the many examples where a small perturbation of the problem description changes an easy tractable problem into something extremely hard.

We mention here only the most important facts about STP. For a survey on results for various Steiner tree modifications, see e.g., [70,91,106].

The decision version of the STP is $NP$-complete and belongs to the list of Karp’s 21 hard problems [73]. In the geometric setting, the STP admits a $PTAS$ [8]. However, the STP on graphs is $APX$-hard unless $P = NP$, even for edges with costs 1 and 2 only [9,18]. There were many gradual improvements in its approximability until the currently best approximation algorithm with a factor of $1 + \ln 3/2 \approx 1.55$ which is independent of the edge cost function [94]. The lower bound is currently set on the ratio of $96/95$ due to [35]. Note that any spanning tree on the graph induced by the set of special vertices achieves a 2-approximation. When the number of the special vertices is a constant, the problem is known to be in $P$ due to the Dreyfus-Wagner algorithm that use the dynamic programming paradigm [48,91].

One may ask whether restricting the cost function to be metric can help us to obtain a better approximation algorithm for STP similarly as we have seen in case of the Traveling Salesman Problem. Sadly, this is not the case for STP. In Theorem 2.8.14, we present a method that shows that by approximating instances of STP with standard triangle inequality one can equally well approximate general instances of STP.

Algorithm 5 (Solving STP via a metric closure)

**Input:** A connected undirected graph $G = (V,E)$ with edge cost function $c: E \rightarrow \mathbb{Q}^+$ and a set of special vertices $K \subseteq V$.

1: Compute a complete distance network $G_D(V)$.

2: Use an $\alpha$-approximation algorithm $A$ to obtain an approximate solution $T_D$ on $G_D$ and special vertices $K$.

3: Let $G'$ be an expansion of $T_D$ in $G$, i.e., the edges of $G'$ correspond to the paths used as edges in $E(T_D)$ and the vertices of $G'$ are precisely the vertices used in these paths.

4: Compute a minimum spanning tree $T$ on $G'$.

**Output:** The Steiner tree $T$. 

Theorem 2.8.14. Let $A$ be an $\alpha$-approximation algorithm for minimum Steiner tree problem on instances where the standard triangle inequality holds. Let $G = (V, E)$ be an instance of STP with arbitrary edge cost function $c$. Then the tree $T$ computed by Algorithm 5 is an $\alpha$-approximation of the minimum Steiner tree problem on graph $G$ with $c$.

Proof. Let us denote by $OPT$ the optimal solution of STP on $G$ and the cost function $c$ and by $OPT_D$ the optimal solution of STP on $G_D(V)$ and the cost function $c_D$. By $\text{cost}$ and $\text{cost}_D$ we denote the cost function of STP on the graph $G$ and the distance network $G_D(V)$, respectively.

First, observe that $c_D$ in the distance network $G_D(V)$ satisfies the triangle inequality. This is true as the cost of the edge between a pair of vertices is replaced by the cheapest path. Hence the application of the $\alpha$-approximation algorithm in step 2 is sound and

$$\text{cost}_D(T_D) \leq \alpha \cdot \text{cost}_D(OPT_D).$$

The edges of $OPT$ in the graph $G_D(V)$ form a Steiner tree. Since $OPT_D$ is the Steiner tree of the minimum cost, we have

$$\text{cost}_D(OPT_D) \leq \text{cost}_D(OPT).$$

Moreover, the usage of the cost of the cheapest paths instead of the cost of the original edges can just decrease the cost and hence

$$\text{cost}_D(OPT) \leq \text{cost}(OPT).$$

The expansion performed in step 3 of the algorithm cannot increase the cost of $G'$. It can actually decrease the cost as the edges encountered several times (in different cheapest paths) in $T_D$ are present in $G'$ just once. Furthermore $G'$ contains all the special vertices from $K$. All in all we have

$$\text{cost}(G') \leq \text{cost}_D(T_D).$$

Finally, the spanning tree $T$ computed by the algorithm spans graph $G'$ that contains all the special vertices. It might not contain all the edges of $G''$, thus

$$\text{cost}(T) \leq \text{cost}(G').$$

This concludes the proof. The tree $T$ contains all the special vertices and

$$\text{cost}(T) \leq \text{cost}(G') \leq \text{cost}(OPT_D) \leq \alpha \cdot \text{cost}(OPT).$$

$\square$
We use a similar approach as presented in Algorithm 5 for obtaining an approximation algorithm for the Steiner cycle problem in Subsection 4.3.2.

As a final note, we would like to mention that even though Theorem 2.8.14 is quite comprehensive, there are still metrics that are special cases of the standard triangle inequality. Such metrics might be still practically relevant and STP can be better approximable here than in the general cases. Examples of such metrics are the sharpened triangle inequality (defined as the $\beta$-relaxed but with $\beta < 1$) that is not discussed in this thesis or the aforementioned Euclidean metrics where a PTAS exists.

2.8.3 The Satisfiability Problem

In computer science, by satisfiability (SAT), we often refer to the decision problem of determining whether a given Boolean formula can be assigned in such a way that it is evaluated to a true. A SAT can be identified in a wide range of naturally occurring problems and solving large SAT instances is useful in various practical areas such as automatic theorem proving, circuit design, etc. The deep study and its wide occurrence in real-world problems stimulated the writing of several monographies and an annual conference. For more details about the topic, we would like to point the reader to monographies such as [1,19,66]. The statements of this subsection are mainly adopted from [69].

Problem 2.8.15. The Satisfiability Problem (SAT):

**Input:** A Boolean formula $F$ over $X = \{x_1, \ldots, x_n\}$ in CNF.

**Decision:** Is there an assignment $\alpha$ to $X$ such that $F(\alpha) = 1$?

If the formula $F$ of SAT is in $k$-CNF, we refer to the SAT problem as $k$-SAT. We will have a particular interest in 2-SAT and 3-SAT.

SAT was the first known example of an $NP$-complete problem. The significant contribution in the proof of its $NP$-completeness was credited to American-Canadian scientist Stephen Cook [38]. In the USSR, results of similar significance were approximately at the same time presented by Leonid Levin [81] (translation in [103]). Hence, the following theorem is often referred to as Cook’s theorem or Cook-Levin theorem. Its proof is omitted due to the complexity, it can be found, e.g., in [69].

**Theorem 2.8.16** (Cook-Levin theorem, [38]). SAT is $NP$-complete.

There is an easy transformation reducing $k$-SAT instances ($k \geq 3$) into 3-SAT instances. The reduction was already discussed as an example of a polynomial-time Karp reduction in Section 2.4.

**Lemma 2.8.17.** There exists a polynomial-time reduction of a $k$-SAT formula ($k \geq 3$) into a 3-SAT formula.

The lemma leaves us with an important corollary.
Corollary 2.8.18. 3-SAT is $NP$-complete.

To make the analysis of $k$-SAT complete and also to identify the hardness of $k$-SAT, we present a positive result for 2-SAT.

Theorem 2.8.19 (Aspvall et al., [10]). 2-SAT is in $P$.

Since SAT is a hard problem, scientists studied many of its modifications where some extra properties on the formula hold, e.g., all its clauses are positive, each variable is present in the formula only constantly many times, etc.

Until now, we were discussing only the basic decision version of the satisfiability problem. However, it has many modifications that are rather optimization problems than decision problems. For instance, one can ask for the extremal number of clauses that can be satisfied in a given Boolean formula. One can also search for an assignment of a Boolean formula that has extremal number of true values. We study the restricted instances of the latter problem further in Chapter 7. In more generalized versions of the above problems, one has assigned weights to clauses/variables/literals of the given Boolean formula and is asked to minimize/maximize the overall cost with respect to the truth assignment to the variables. An overview of other optimization problems based on SAT, together with the summary of the known results, can be found in [56] and in [11].

2.8.4 The Minimum Set Cover Problem and the Minimum Hitting Set Problem

Other important optimization problems that appear in many natural environments are the minimum set cover problem and the minimum hitting set problem which we introduce in this subsection. The minimum set cover problem belongs to the list of Karp’s 21 $NP$-complete problems [73]. It has an important role in the field of approximation algorithms as its study led to a development of fundamental techniques in the entire field. As we discuss later in a separate paragraph, the two problems are dual, but certain properties are not preserved by this duality. This means that the hardness results achieved for one of the problems usually carry over to the other problem. However, depending on the considered parameters (such as $|U|$ or $|S|$), the precise calculations of estimations in both problems might differ by a polynomial factor.

Problem 2.8.20. Minimum Set Cover Problem (MIN-SC):

Input: A universe $U = \{u_1, \ldots, u_n\}$ and a finite family of sets $S = \{S_1, \ldots, S_m\}$ such that $S_i \subseteq U$, for all $1 \leq i \leq m$.

Feasible solutions: A set $Q \subseteq S$ that covers $U$, i.e., for all $u \in U$ there exists $Q \in Q$ such that $u \in Q$.

Costs: The number of the elements in $Q$, i.e., $|Q|$.

Goal: Minimization.
Problem 2.8.21. Minimum Hitting Set Problem (Min-HS):

Input: A universe $U = \{u_1, \ldots , u_n\}$ and a finite family of sets $S = \{S_1, \ldots , S_m\}$ such that $S_i \subseteq U$, for all $1 \leq i \leq m$.

Feasible solutions: A set $H \subseteq U$ that hits all the sets in $S$, i.e., for all $S \in S$ there exists $u \in H$ such that $u \in S$.

Costs: The number of the elements in $H$, i.e., $|H|$.

Goal: Minimization.

We refer to a feasible solution of Min-SC as a set cover and to a feasible solution of Min-HS as a hitting set.

To outline the two problems, we use a simple example where the universe and the family of sets is the same for both problems.

$$U = \{1, 2, 3, 4, 5\},$$
$$S = \{\{1, 2, 3\}, \{1, 2, 5\}, \{1, 4, 5\}, \{2, 3, 4\}, \{2, 4, 5\}, \{3, 5\}\}. \quad (2.1)$$

One possible feasible solution of Min-SC that covers all the elements of $U$ is the family of sets $\{\{1, 2, 3\}, \{2, 3, 4\}, \{3, 5\}\}$. However, this is not the optimal solution. To cover all the elements of $U$, one can pick the family of sets $\{\{1, 2, 3\}, \{2, 4, 5\}\}$ that contains less sets. Since there is no single set that contains all the elements of $U$, indeed, $\{\{1, 2, 3\}, \{2, 4, 5\}\}$ is one of the optimal solutions of our instance of Min-SC. In the Min-HS, we have a somehow dual problem. Instead of selecting sets that cover elements of $U$, we have to select elements of $U$ that hit the family of sets $S$. One of the feasible solutions of our instance of Min-HS is the set $\{1, 4, 5\}$. However, we can hit the set $S$ by fewer elements, e.g., $\{2, 5\}$, which is also one of the optimal solutions, as no element is present in all the sets of $S$.

The last problem by which we need to extend our problem base is the following subproblem of Min-HS.

Problem 2.8.22. Min-\(d\)-HS is the following optimization problem:

Input: An universe $U = \{u_1, \ldots , u_n\}$ and a finite family of sets $S = \{S_1, \ldots , S_m\}$ such that $S_i \subseteq U$ and $|S_i| \leq d$, for all $1 \leq i \leq m$.

Feasible solutions: Any set of elements $H \subseteq U$ such that $S_i \cap H \neq \emptyset$, for all $1 \leq i \leq m$.

Costs: The number of elements in $H$, i.e., $|H|$.

Goal: Minimization.

Min-\(d\)-HS can be formulated in terms of graph theory as a \(d\)-dimensional hypergraph. The vertices are identified with the elements of the universe and each set of the set family corresponds to a single \(d\)-dimensional hyperedge. The hitting
set of an instance is a collection of vertices such that each hyperedge contains (is covered by) at least one of the vertices of the collection.

The remainder of this subsection is divided into several paragraphs. In the first paragraph, we discuss the duality between Min-SC and Min-HS. The recently uprising unique games conjecture is presented in the second paragraph. Even though the conjecture can be applied on a broad range of problems and it could fill a section on its own, we present it only shortly. It is a part of this subsection as, in our thesis, we relate it only with the Min-HS and all our results related to this conjecture are derived directly from the Min-HS results. The following two paragraphs are devoted to the negative and positive known results for Min-HS that are used in Chapter 5 and, indirectly, in Chapter 7 of our thesis. The results for other fields such as parameterized complexity theory and exact algorithms that we are building on in our thesis, are summarized in the last paragraph.

Since all our results are derived from the known facts for Min-HS, the Min-SC is not discussed here and all the statements we summarize in this subsection are stated for Min-HS. Nevertheless, the statements for Min-HS can be transformed into the Min-SC setting preserving the hardness properties, but possibly changing the exact values of estimations. For a more detailed overview on known results and definitions of various subclasses of Min-SC and Min-HS, we would like to point the reader, for instance, to [11] and [56].

**Duality of the Two Problems**

Suppose we are given an instance of Min-SC

\[ U = \{u_1, \ldots, u_n\}, \]
\[ S = \{S_1, \ldots, S_m\} \]

Let us view the instance from a different perspective. Let our new universe \( U' \) consist of elements \( s_1, \ldots, s_m \) that are in one-to-one correspondence with the sets of \( S \). For each element \( u_j \) (\( 1 \leq j \leq n \)) of the original universe \( U \), we create a set of elements \( U_j \) of the new universe \( U' \): we pick the elements of \( U' \) that correspond to the sets of \( S \) that contain \( u_j \). Formally, our new instance can be denoted as

\[ U' = \{s_1, \ldots, s_m\}, \]
\[ S' = \{U_1, \ldots, U_n\} \]
\[ U_j = \{s_i \mid u_j \in S_i \land 1 \leq i \leq m\}, \text{ for } 1 \leq j \leq n. \]

Observe that a feasible solution \( Q \) of our instance (\( U, S \)) of Min-SC can be one-to-one transformed into a solution of the instance (\( U', S' \)) of Min-HS: for each set of \( Q \), the feasible solution of the Min-HS instance contains the corresponding element of \( U' \). Indeed, if \( S_j \in Q \) covers the element \( u_i \) of \( U \), i.e., \( u_i \in S_j \), then \( s_j \in U' \) hits the set \( U_i \) of \( S' \), i.e., \( s_j \in U_i \). Hence, we see that instances of Min-SC are instances of Min-HS viewed from a different perspective.
2.8. Overview of Some Well-Known Problems

The same transformation can be performed also to transform instances of Min-HS into instances of Min-SC, so that the feasible solutions of the two problems correspond again in a one-to-one manner. Very generally, the transformation just identifies the universe and the family set of the one problem to the family set and the universe of the second problem. The observation of this duality was first made by Ausiello et al. in [12] and is outlined by the following theorem.

**Theorem 2.8.23** (Ausiello et al., [12]). For every instance \((U, S)\) of the minimum set cover problem, the pair \((U', S')\) is an instance of the minimum hitting set problem and vice versa. Feasible solutions of both instances \((U, S)\) and \((U', S')\) are identical.

Due to the described duality, both problems are considered to be equal with respect to the hardness. However, we would like to note that the results concerning approximation ratios or lower bounds, where precise values come to the play, might not be the same. This is very dependent on the parameters in the calculations. Surely, the overall size of the instance is unchanged by the transformation. However, the transformation changes the sizes of \(U'\) and \(S'\) to \(m\) and \(n\), respectively. Hence the size of the universe might be polynomially blown up by the transformation (e.g., if \(m = \Theta(n^2)\)). This might effect the approximation ratio or other estimations if they are expressed only in one of the two parameters.

Moreover, if special instances with certain properties are considered, the transformation might not preserve these properties. For example, if \(|S_i| \leq k\), for each \(1 \leq i \leq m\) of the Min-SC instance, the transformation yields an instance of Min-HS in which each element of \(U'\) is present in the sets of \(S'\) at most \(k\) times. But we cannot state anything about the sizes of the sets of \(S'\). Hence, once calculations are performed or special instances of the two problems are considered, we have to be very careful how the calculations and the properties of instances carry over in the transformation.

**Unique Games Conjecture**

The unique games conjecture was first mentioned by Subhash Khot in 2002 by postulating that certain types of games are \(\text{NP}\)-hard. Since then, the study of the conjecture emerged and currently several different equivalent versions of the conjecture statement are known. In this paragraph, we state it in a form that we find to be the most suitable with respect to the content of this thesis.

The unique games conjecture (UGC) is currently accepted as one of the methods for proving lower bounds on the hardness of problems. Hence, it could have been presented in Subsection 2.5.3. However, since all our results considering UGC are just corollaries that follow from reductions of studied problems to the Min-HS, we find more appropriate to mention it briefly here. We adopt the presentation of UGC from [105]. More details related to UGC can be found in the survey of Khot [75].

Now we give a few basic definitions that allow us to formulate the unique games problem that is employed in the conjecture.
Definition 2.8.24. Let $U$ be a finite universe and let $X = \{x_1, \ldots, x_n\}$ be a set of variables over $U$. A binary constraint on variables $X$ with respect to $U$ is a function $f_{p,q} : U \times U \to \{0, 1\}$, where $1 \leq p < q \leq n$.

Informally, a binary constraint $f_{p,q}$ on $\{x_1, \ldots, x_n\}$ describes the relationship between variables $x_p$ and $x_q$. More precisely, for each pair of values that can be assigned to the two related variables, the outcome of the function describes whether the constraint is satisfied and hence the two variables are in accordance with the binary constraint or not. Formally, the satisfiability of a binary constraint is defined as follows.

Definition 2.8.25. A binary constraint $f_{p,q} : U \times U \to \{0, 1\}$ on variables $X = \{x_1, \ldots, x_n\}$ with respect to $U$ is satisfied for an assignment $(u_1, \ldots, u_n)$ if and only if $f_{p,q}(u_p, u_q) = 1$.

Problem 2.8.26. The unique games problem (UGP):

Input: A finite universe $U$, a set of variables $X = \{x_1, \ldots, x_n\}$ and $m$ binary constraints for $X$ on $U$ such that, in addition, for each constraint and a given value of one variable, there exists a single value of the other variable such that the constraint is satisfied.

Feasible solutions: An assignment $\alpha = (u_1, \ldots, u_n)$ of values of $U$ to variables of $X$.

Costs: The number of satisfied constraints for the assignment $\alpha$.

Goal: Maximization.

Informally, in UGP we are given variables and a set of its possible values together with constraints relating the values of pairs of the variables. Our goal is to find an assignment to the variables such that as many constraints as possible are satisfied.

Note that deciding whether all the constraints of a given unique games problem can be satisfied is not a hard problem. It can be decided in polynomial time as follows. First, transform the instance into a graph representation – the vertices correspond to the variables and each constraint connects the corresponding two vertices by an edge. W.l.o.g. assume that the graph is a single component, otherwise the approach is used on each of the components separately. Pick an arbitrary variable $x$ and sequentially try to assign it all the values of $U$. For each assignment of a value to $x$, deduce the unique values of the variables that are related with $x$ via a constraint to satisfy this constraint. These variables correspond to the vertices of the graph that are in distance 1 from vertex corresponding to $x$. Propagate the deduction of assignments of the, by definition unique, values to the variables related by constraints with already assigned variables. This corresponds to a breath-first search with value assignment on the graph, starting from the vertex corresponding to $x$. Once the propagation is finished (i.e., the component is searched), each variable corresponding to a vertex of the graph is assigned a
value, and all the constraints are satisfied, the assignment is a feasible solution of the UGC instance. If no feasible solution among all the tested assignments of $x$ is found, there is no solution as one of the tested values of $U$ has to be assigned to $x$ and the assignment of the rest of the variables is then uniquely determined through the constraints.

Informally, the unique games conjecture says that distinguishing instances in which many (or almost all of the) constraints can be satisfied and very few (or almost none) of the constraints can be satisfied is $\mathcal{NP}$-hard. Formally, the conjecture is formulated as follows.

**Conjecture 2.8.27 (Unique games conjecture (UGC)).** Given any $\epsilon, \delta > 0$, there exists some $k > 0$, depending on $\epsilon$ and $\delta$, such that, for the unique games problem with a universe of size $k$, it is $\mathcal{NP}$-hard to distinguish between instances in which at least a $(1-\epsilon)$-fraction of the constraints can be satisfied and instances in which at most a $\delta$-fraction of the constraints can be satisfied.

As a last remark of the paragraph, we would like to note that it is not known whether or not there exists an approximation algorithm with an outcome that would disprove the conjecture. However, if a given instance of UGP contains a $(1-\epsilon)$-fraction of satisfiable constraints, there is a polynomial-time algorithm known delivering a solution that satisfies a $1 - O(\sqrt{\epsilon \log n})$ fraction of the constraints (see [105] for details).

**Negative Results on the Approximability**

In this paragraph, we shortly summarize the known negative results for Min-HS and Min-$d$-HS that are mainly derived through the duality from the corresponding results for Min-SC. The proofs are omitted as our results for the examined problems of this thesis are not building on the techniques used in the proofs, but on their outcomes.

**Theorem 2.8.28** (Creignou et al., [40]). Min-HS is LOGAPX-complete.

**Theorem 2.8.29** (Alon et al, [4]). Min-HS cannot be approximated within a factor of $0.2267 \cdot \ln |S|$, unless $\mathcal{P} = \mathcal{NP}$.

**Theorem 2.8.30** (Papadimitriou and Yannakakis, [90]). The Min-$d$-HS is APX-complete for $d \geq 2$.

**Theorem 2.8.31** (Dinur et al, [43]). For arbitrary $\epsilon > 0$ and $d \geq 3$, it is $\mathcal{NP}$-hard to approximate Min-$d$-HS by a polynomial-time algorithm within a ratio of $d - 1 - \epsilon$.

**Theorem 2.8.32** (Khot and Regev, [76]). For arbitrary $\epsilon > 0$ and $d \geq 2$, Min-$d$-HS cannot be approximated by a polynomial-time algorithm within a factor of $d - \epsilon$, unless the unique games conjecture fails.

**Theorem 2.8.33** (Dinur and Safra, [44]). It is $\mathcal{NP}$-hard to approximate Min-2-HS by a polynomial-time algorithm within a factor of 1.3606.
Positive Results on the Approximability

We summarize the currently best known positive results for MIN-HS and MIN-$d$-HS in the following theorems. A simple proof for the $d$-approximability of MIN-$d$-HS is given, but the proofs of the remaining theorems are omitted as they are not vital for the results presented in this thesis.

**Theorem 2.8.34** (Johnson, [72]). There exists a polynomial-time approximation algorithm for MIN-HS with ratio of $1 + \ln |U|$.

**Theorem 2.8.35** (Bar-Yehuda and Even, [16]). For any $d \geq 2$, there exists a polynomial-time $d$-approximation algorithm for the MIN-$d$-HS.

*Sketch of the proof.* Let $G$ be a hypergraph that corresponds to the given instance $(U, S)$ of MIN-$d$-HS. The algorithm iterates over the set of hyperedges that correspond to the elements of $S$. It greedily adds all the vertices of a not yet covered hyperedge $S_i$ into the solution $H$. Then it removes all the added vertices from the graph together with all the hyperedges in which at least one of the removed vertices is present.

From the description of the algorithm, it is clear that each hyperedge has at least one of its vertices present in $H$ — all the vertices of the hyperedge are either added to $H$ in some step of the algorithm, or the hyperedge is removed as one of its vertices was added into $H$.

Observe that the hyperedges that are greedily selected do not have any vertex in common. Hence, to hit each of these hyperedges, at least one of its vertices has to be selected into an optimal solution. Our algorithm selects all the vertices of such hyperedge and hence, it selects at most $d$-times more of the vertices into the solution than an optimal solution does.

**Theorem 2.8.36** (Halperin, [64]). For arbitrary $d \geq 2$, there exists a polynomial-time $\left( d - \frac{(d-1)\ln \ln |U|}{\ln |U|} \right)$-approximation algorithm for MIN-$d$-HS.

Parameterized Complexity Results for MIN-$d$-HS

This paragraph briefly summarizes the recent results in the field of parameterized complexity theory with respect to MIN-$d$-HS and the parameter that we are considering in Chapter 5 and Chapter 7.

Before stating the theorems, we have to fix the parameterized version of MIN-$d$-HS in which we are interested.

**Problem 2.8.37.** MIN-$d$-HS-SOL($k$) is a parameterized decision problem where:

**Input:** An instance of MIN-$d$-HS and a parameter $k$.

**Decision:** Is there a feasible solution of the MIN-$d$-HS instance of cost at most $k$?

Deciding whether, for the parameter $k$, there exists a feasible solution of MIN-$d$-HS-SOL($k$) is $NP$-complete, even if the size of every set in $S$ is bounded by 2 [56].
Therefore, indeed, our parameterized version of the Min-\(d\)-HS problem is hard and hence the study of the problem is well motivated. From the viewpoint of parameterized complexity theory, the parameterized problem of deciding whether the general Min-HS admits a solution of cost at most \(k\) is \(W[2]\)-complete (see, e.g., [47]). However, it can be easily shown that Min-\(d\)-HS-sol(\(k\)) is in \(\mathcal{FPT}\). The currently best known fixed-parameter tractable algorithm for Min-\(d\)-HS is due to Niedermeier and Rossmanith.

**Theorem 2.8.38** (Niedermeier and Rossmanith, [88]). For an arbitrary \(d \geq 3\), there exists a fixed-parameter algorithm solving Min-\(d\)-HS-sol(\(k\)) in time \(O(\alpha^k + n)\), where \(n\) is the number of elements of the universe and

\[
\alpha = \frac{d - 1}{2} + \frac{d - 1}{2} \sqrt{1 + \frac{4}{(d - 1)^2}} = d - 1 + \mathcal{O}(d^{-1}).
\]

For Min-2-HS-sol(\(k\)), we can do even better.

**Theorem 2.8.39** (Chen et al., [33]). There exists a fixed-parameter algorithm solving Min-2-HS-sol(\(k\)) in time \(O(kn + 1.2852^k)\), where \(n\) is the size of the instance.

### 2.8.5 The Minimum Vertex Cover Problem and the Maximum Independent Set Problem

Another prominent member of Karp’s famous list of 21 \(NP\)-complete problems [73] and a very frequently appearing problem with many applications is the minimum vertex cover problem.

**Problem 2.8.40.** Minimum Vertex Cover Problem (Min-VC):

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

**Feasible solutions:** A set \(C \subseteq V\) such that, for all \(uv \in E\), at least one of the endpoints is in \(C\), i.e., \(u \in C\) or \(v \in C\).

**Costs:** The number of elements in \(C\), i.e., \(|C|\).

**Goal:** Minimization.

We refer to a feasible solution of Min-VC as a vertex cover of the given graph.

Observe that Min-VC is identical to the minimum hitting set problem where the size of each set of the family set is two (i.e., Min-2-HS): identify the vertices \(V\) with the universe \(U\) and each edge of \(E\) with a single set from the family set \(S\). Hence, in this subsection, we briefly list the known results valid only for the Min-VC. For more details, review Subsection 2.8.4 with known facts about Min-\(d\)-HS, or one of the textbooks on hard problems and complexity theory, e.g., [11, 56, 68].
The MIN-VC can be approximated by a well known 2-approximation algorithm (see Theorem 2.8.35). It is $APX$-complete (Theorem 2.8.30), it is $NP$-hard to approximate within a ratio of 1.3606 (Theorem 2.8.33) and is not approximable within a ratio of $2 - \varepsilon$, unless the unique games conjecture fails (Theorem 2.8.32).

Another, in some sense dual problem to MIN-VC is the so-called maximum independent set problem, that can be formulated as follows.

**Problem 2.8.41.** Maximum Independent Set Problem (Max-IS):

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

**Feasible solutions:** A set $C \subseteq V$ such that, for all $uv \in E$, at most one endpoint is in $C$, i.e., if $u \in C$ then $v \notin C$.

**Costs:** The number of the elements in $C$, i.e., $|C|$.

**Goal:** Maximization.

There are two standard definitions for MAX-IS generalized to hypergraphs. In this thesis, we consider the so-called (strong) maximum independent set problem where each hyperedge is required to contain at most one element of the feasible solution. However, until it is explicitly stated otherwise, we consider only standard graphs further.

We refer to a feasible solution of MAX-IS as an independent set of a given (hyper)graph.

In standard graphs, MAX-IS and MIN-VC are dual in the following sense. Note that this duality does not hold for our definition of independent set on hypergraphs.

**Theorem 2.8.42.** Let $G = (V, E)$ be a undirected graph. A set $C \subseteq V$ is an independent set of $G$ if and only if $V \setminus C$ is a vertex cover of $G$.

**Proof.** If $C$ is an independent set, then no edge $e \in E$ can contain both its endpoints in $C$ and hence, at least one of its endpoints is in $V \setminus C$ which is covering $e$.

Similarly, if $V \setminus C$ is a vertex cover of $G$, then at least one endpoint of each edge $e \in E$ is in $C$ and hence at most one endpoint of $e$ can be present in $C$.

The duality can be generalized to MAX-IS and MIN-VC – the set of vertices that are not present in an optimal solution of MAX-IS forms an optimal solution of MIN-VC, and vice versa. This allows us to transfer exact solutions between the two problems. However, we cannot use it to transfer approximate solutions back and forth – the bounds on the cost of such solutions are not preserved.

Perhaps, not surprisingly, MAX-IS with $n$ vertices is a hard problem. It is not approximable by a factor of $n^{1-\varepsilon}$, unless $\mathcal{P} = \mathcal{NP}$ [65], but there exists an $O(n/(\log n)^2)$-approximation algorithm [28]. It is an extensively studied problem under different assumptions on the input graph, but we state only the known facts that we apply in our results.

**Theorem 2.8.43** (Trevisan, [104]). The maximum independent set problem on a hypergraph with $n$ vertices cannot be approximated by a polynomial-time algorithm within a factor of $\frac{n}{2^{O(\sqrt{\log n})}}$, unless $\mathcal{P} = \mathcal{NP}$.
In the remainder of this subsection, we refer to exact algorithms for MIN-VC that are obtained through the duality of MIN-VC and MAX-IS.

**Theorem 2.8.44** (Bourgeois et al., [29]). There is an $O^*(1.2127^n)$ time exact algorithm solving the minimum vertex cover problem with $n$ vertices in polynomial space.

**Theorem 2.8.45** (Robson, [95]). There is an $O^*(1.2109^n)$ time exact algorithm solving the minimum vertex cover problem with $n$ vertices in exponential space.

We are aware of the fact that Robson claims in his technical report [96] a better running time $O^*(2^{n/4}) \approx O^*(1.189208^n)$ of the algorithm in both polynomial and exponential space. However, the description of this new algorithm takes almost 18 pages and was partially generated by a computer. Moreover, we are not aware that the result was published in any peer-reviewed paper. Thus, we decided to employ the algorithms with worse running time instead.
Chapter 3

The Ordered Traveling Salesman Problem

Many practically relevant problems in operations research can be represented by the traveling salesman problem (TSP) or one of its many generalizations [63]. We consider here its variant where the set of feasible cycles is restricted by some precedence constraints, we require that certain vertices have to be visited before certain others. Although such precedence constraints can be expressed by an arbitrary partial ordering on the vertices, we will restrict our attention to a simple case, the so-called near-metric ordered TSP, where a linear order on a subset of \(k\) vertices is given as a part of the input and a feasible solution has to contain these special vertices in the prescribed order. The restriction on the edge cost that we consider is the \(\beta\)-relaxed triangle inequality.

3.1 Problem Definition, Known Results and Outline of Our Results

The variant we consider is introduced in Subsection 2.8.1 first for the metric instances as Problem 2.8.2. It is followed by the study of the near-metric instances, in which we are here interested in, and that we denoted as \(k\)-\(\Delta_\beta\)OTSP. The Subsection 2.8.1 contains the overview of the problem with known results that we refer to in this chapter.

The previously best algorithm for \(k\)-\(\Delta_\beta\)OTSP is due to Böckenhauer et al. [23], achieving an approximation ratio of

\[(k + 1) \cdot \min \{4\beta^2 + \log_2(k-1) , 1.5\beta^3 + \log_2(k-1) , (\beta + 1)\beta^2 + \log_2(k-1) \} .\]

The details of this algorithm are presented in Subsection 2.8.1 as Algorithm 4. In this chapter, we discuss several improvements of the algorithm.

First, in Section 3.2, we present an algorithm based on a different structure than the one of Böckenhauer et al. which allows us to prove a better approximation
Chapter 3. The Ordered Traveling Salesman Problem

ratio of
\[ k \beta \log_2(3k-3), \]
that is, of approximately \( k \beta^{1.59 + \log_2(k-1)} \). The proportion between the approximation factor of Böckenhauer et al. and the new algorithm is shown in Figure 3.1. As we can see, an improvement in the ratio is achieved for all pairs \((k, \beta)\).

Furthermore, in Section 3.3, we present a modification of the algorithm of Böckenhauer et al. which allows us to do a more accurate estimation. We decrease the approximation ratio of the modified algorithm to
\[ \frac{3}{2} \cdot \left\lceil \frac{k}{2} \right\rceil \cdot \beta^{2+\log_2(2\left\lceil k/2 \right\rceil+k-3)}. \]

We obtain an improvement for the instances with \( \beta < (3/4)^{\log_2 3/8} \approx 1.2254 \). We also discuss a further enhancement of our algorithm (Section 3.4), that, for cases with \( n \geq \frac{11k+7}{2} \) and \( \beta < \frac{4}{3} \), improves the ratio to
\[ \frac{3}{2} \cdot \left\lceil \frac{k}{2} \right\rceil \cdot \beta^{2+\log_2(\left\lceil k/2 \right\rceil+k-2)}. \]

Since in the case of \( \beta \)-relaxed triangle inequality we are unable to use the "shortening" technique discussed in Subsection 2.8.1, we use a different approach. To avoid duplicate presence of vertices in our structures, we color the vertices of the graph more or less uniformly by some colors. When the structure is built, instead of using all the vertices we would need, we use only vertices of one fixed color. Since the colors in the structure are uniformly distributed, we can estimate the maximum number of vertices that are in the structure but were bypassed by picking just vertices of one color to the structure. Hence, the overall cost of the structure increases by a factor exponential in \( \beta \), but we are able to bound its exponentiation. This bound of course depends on the number of colors we used and also on the distribution of vertices of one color in the graph.

3.2 A New Approximation Algorithm for Near-Metric Ordered Traveling Salesman Problem

This section is devoted to a new approximation algorithm for \( k-\Delta_\beta \text{OTSP} \). The algorithm achieves a \( k \beta^{\log_2(3k-3)} \)-approximation ratio which is improving the ratio of the algorithm of Böckenhauer et al. (the paper was published as [23]) that is discussed in Subsection 2.8.1 in more detail. The plot of the computed proportion between the approximation ratio of the algorithm of Böckenhauer et al. and the new algorithm is shown in Figure 3.1.

Our improved algorithm is presented as Algorithm 6. In the following, we comment it line by line, we discuss its invariants during execution and, in the end, we prove its ratio and correctness.
3.2. A New Approximation Algorithm for $k$-$\Delta_\beta$OTSP

Figure 3.1: The graph shows the quotient of the previously best known approximation ratio $(k + 1) \cdot \min \{4\beta^{2+\log_2(k-1)}, 1.5\beta^{3+\log_2(k-1)}, (\beta + 1)\beta^{2+\log_2(k-1)}\}$ and the ratio $k\beta^{\log_2 3(k-1)}$ achieved by Algorithm 6, i.e., the improvement achieved by the algorithm over the algorithm of Böckenhauer et al.

The general idea of our algorithm is to first connect the special vertices in the required order into a cycle $C$ with no duplicate vertices and then to add the remaining vertices into $C$ to create a Hamiltonian cycle. We use the unique paths in a minimum spanning tree of the graph to connect the special vertices into the cycle $C$. Such paths can guarantee a low cost, but we have to avoid to include the same vertex more than once into $C$. To systematically do this, we color the vertices of the spanning tree uniformly by $k$ colors and we include only vertices of one color in the path connecting two special vertices, hence the vertices are not too far apart and we can still bound the overall cost. The vertices that are not yet merged into $C$ are then connected by two ways. Either a vertex is absorbed by a path into $C$ if it was bypassed by this path. Or there is an entire tree of not yet connected vertices. In this case, we use an approximation algorithm to compute a reasonably cheap Hamiltonian cycle on the tree vertices and we merge the computed Hamiltonian cycles into the cycle $C$.

In the first step of Algorithm 6, we build a minimum spanning tree $T$ and we arbitrarily choose one non-special vertex $r$ of $T$ as its root. Then, starting with $r$, we color all vertices except the special vertices $s_1, \ldots, s_k$, i.e., those vertices that have to occur in prescribed order in the solution, in a breadth-first-search manner by a coloring function $f(v) = ((f($the nearest colored ancestor of $v$) + 1) \mod k) + 1$. From now on, we omit to write the mod $k$ and the addition of 1 from the description of the colors, when it is clear from the context. The values are always in the set $\{1, \ldots, k\}$.

The paths $L_i$ in Algorithm 6 connecting the vertices $s_i$ and $s_{i+1}$ use the coloring for bounding the distance between consecutive vertices of $L_i$ from above.

**Lemma 3.2.1.** In step 3 of Algorithm 6, at most $3k - 4$ edges of $T$ are bypassed between two consecutive vertices of color $i$ while picking vertices for path $L_i$. 
Algorithm 6 (A $k\beta\log_2 3(k-1)$-approximation for $k$-Δ$_\beta$OTSP)

**Input:** A complete graph $G = (V, E)$ with edge cost function $c: E \rightarrow \mathbb{Q}^+$ that satisfies the $\beta$-relaxed triangle inequality ($\beta > 1$), and a sequence of special vertices $(s_1, \ldots, s_k)$ from $V$.

1. Build a minimum spanning tree $T$ of graph $G$ and choose a non-special vertex of $T$ to be a root $r$.
2. Define a coloring function $f: V \setminus \{s_1, \ldots, s_k\} \rightarrow \{1, \ldots, k\}$ as follows. Color $r$ by color 1. Continue with coloring the non-special vertices in breadth-first-search manner: For each vertex $u$, let $v$ be the closest ancestor in $T$ that is not in $\{s_1, \ldots, s_k\}$. Color vertex $u$ by the color $f(u) := ((f(v) + 1) \mod k) + 1$.
3. For $1 \leq i \leq k$, let $L_i$ be the path formed by $P_T(s_i, s_{i+1})$, picking only vertices with color $i$, starting with $s_i$ and ending with $s_{i+1}$.
4. Create new paths $L'_1, \ldots, L'_k$ from $L_1, \ldots, L_k$ by including the vertices bypassed by all $L_i$ as shown in Figure 3.3 and discussed later.
5. Create a cycle $C'$ by connecting paths $L'_1, \ldots, L'_k$.
6. Let $T_1, \ldots, T_q$ with roots $r_1, \ldots, r_q$ be the maximal subtrees of $T$ such that the vertices in $V(T_i) \setminus \{r_i\}$ are not included in $C'$ and $r_i \in V(C')$. Let $e_i^*$ be the cheapest edge incident to $r_i$ in $T_i$. Use the procedure HCT$^3$ (refined) from Subsection 2.8.1 on each pair $(T_i, e_i^*)$, for $1 \leq i \leq q$, to obtain a Hamiltonian cycle $H_i$ on the vertices of $T_i$.
7. For $1 \leq i \leq q$, merge $H_i$ with $C'$ in $r_i$ using $e_i^*$ to construct a Hamiltonian cycle $H$.

**Output:** The Hamiltonian cycle $H$.

**Proof.** Let $u$ and $v$ be two consecutive vertices in the path $L_i$. Let $x$ be the lowest common ancestor of $u$ and $v$ in $T$, i.e., no descendant of $x$ is an ancestor of both $u$ and $v$. Note that $x$ might be $u$ or $v$.

The path $P_T(u, v)$ contains at most $k - 2$ non-special vertices between $x$ and $u$ and also between $x$ and $v$. There are at most $k - 2$ special vertices. Therefore, together with $u$ and $v$, the total number of vertices in $P_T(u, v)$ is at most $(k - 2) + (k - 2) + 1 + (k - 2) + 2 = 3k - 3$. Therefore, the number of edges between the two vertices $u$ and $v$ in $P_T(u, v)$ is at most $3k - 4$. 

If we connect the paths $L_1, \ldots, L_k$ now, we obtain a cycle $C$ containing each vertex at most once and the special vertices respect the given order. But here, we have the problem that $C$ might not contain all vertices of $G$. There are two types of vertices not connected to $C$:

1. vertices in the paths in $T$ between two consecutive special vertices that are bypassed but not used (see the left part of Figure 3.2). Formally, we define the set of these vertices as $W = \{v \in V \mid \forall i \ (1 \leq i \leq k) \ v \notin V(L_i) \land \exists j (1 \leq j \leq k) \ v \in V(P_T(s_j, s_{j+1}))\}$,
2. vertices in subtrees of tree $T$ where no special vertex is located and thus no path is passing this part of the tree (see the right part of Figure 3.2). Formally, $Y = \{v \in V \mid \forall i (1 \leq i \leq k) \ v \notin V(P_T(s_i, s_{i+1}))\}$.

Figure 3.2: The triangle denotes the tree $T$, dashed lines denote the paths $L_1, \ldots, L_k$ and the line between $s_i$ and $s_{i+1}$ is the path $P_T(s_i, s_{i+1})$. The two possible cases where vertex $v$ is not connected to path $L_i$ are shown here. Either the vertex $v$ lies on some path $P_T(s_i, s_{i+1})$ but was not picked to some path $L_j$ ($1 \leq j \leq k$) (left part) or the vertex $v$ is in a subtree where no path $L_j$ is passing (right part).

Now, we show how to merge all vertices of $W$ into the paths $L_i$, thereby constructing the new paths $L'_i$. We process the paths $L_i$ in order, starting with $L_1$. After creating the updated path $L'_i$, let $W_i$ be the set of vertices of $W$ that are still not connected to any of the paths, with $W_0 = W$.

Processing of the path $L_i$ works as follows (see Figure 3.3). Let $xy$ be an arbitrary edge in $L_i$, and let $P_T(x, y) = xv_1v_2 \ldots v_zy$ be the corresponding path in the tree. Let $p_{xy}$ be the maximal index such that, for all $1 \leq j \leq p_{xy}$, $v_j \in W_{i-1}$. Similarly, let $q_{xy}$ be the minimal index such that, for all $j$ such that $p_{xy} < q_{xy} \leq j \leq z$, $v_j \in W_{i-1}$. If $p_{xy}$ and $q_{xy}$ exist, we remove the edge $xy$ from $L_i$ and replace it by the path $xv_1v_2 \ldots v_{p_{xy}}v_{q_{xy}} \ldots v_zy$. In the case where $p_{xy}$ or $q_{xy}$ does not exist, $v_1 \ldots v_{p_{xy}}$ or $v_{q_{xy}} \ldots v_z$ is empty. Then we replace $xy$ by $xv_{q_{xy}}, \ldots, v_zy$ or $xv_1 \ldots v_{p_{xy}}y$, respectively. The new path $L'_i$ is constructed by repeating this process for every edge in $L_i$. We set $W_i = W_{i-1} \setminus V(L'_i)$. Note that vertices from $W_i$ that are located between $p_{xy}$ and $q_{xy}$, if any, are added to some later path $L'_{h}$. This follows from the fact that, since $p_{xy} < q_{xy}$, there exists an edge $(v_{d_1}, v_{d_2})$ in $L_h$ ($i < h \leq k$ and $p_{xy} < d_1 < q_{xy}$) that is processed later.

Using the described approach, we include vertices from $W$ into the new paths $L'_1, \ldots, L'_k$. Then we join these paths to the cycle $C'$. The cycle $C'$ passes the special vertices in the required order and each vertex in $W$ is included there exactly
Chapter 3. The Ordered Traveling Salesman Problem

Once. Therefore, later on we deal with the remaining non-connected vertices (the set \( Y \)).

The vertices in \( Y \) are organized in subtrees of \( T \). We use this fact and we apply the algorithm \( \text{HCT}^3 \) (refined) (see Algorithm 3 from Subsection 2.8.1 for details). This algorithm is an approximation algorithm for the cheapest Hamiltonian cycle in a complete graph with a cost function satisfying the relaxed triangle inequality. The algorithm builds a Hamiltonian cycle on the vertices of a tree and bounds the cost of the created cycle by \( \beta + \beta^2 \) times the cost of the tree. Moreover, the algorithm includes one particular edge of the tree to the created cycle. This edge is then used to merge the created cycle to our cycle \( C' \). In the following paragraphs we will discuss the use of algorithm \( \text{HCT}^3 \) (refined) in more detail.

Let \( T_1, \ldots, T_q \) be the maximal non-empty subtrees with roots \( r_1, \ldots, r_q \) of \( T \) such that \( (V(T_i) \setminus \{r_i\}) \cap V(C') = \emptyset \), i.e., all vertices of the trees except their roots are not in \( C' \). Let \( e_1^*, \ldots, e_q^* \) be the cheapest edges of \( T_1, \ldots, T_q \) that are incident with \( r_1, \ldots, r_q \) (in \( \text{HCT}^3 \) (refined) they are called locally minimal). We use the procedure \( \text{HCT}^3 \) (refined) with initial edges \( e_1^*, \ldots, e_q^* \) to build the Hamiltonian cycles \( H_1, \ldots, H_q \) that contain all vertices from \( T_1, \ldots, T_q \) – see step 6 of Algorithm 6.

The last part of the algorithm is the merge of \( H_1, \ldots, H_q \) with \( C' \). We use the property of the procedure \( \text{HCT}^3 \) (refined) that \( H_1, \ldots, H_q \) contain the initial edges \( e_1^*, \ldots, e_q^* \) respectively. These are edges of the tree \( T \) and therefore we are able to merge the Hamiltonian cycles to \( C' \) in order to create the Hamiltonian cycle \( H \) without duplicating vertices and without significant increase of the total cost as it is shown in Figure 3.4.

The important property of the \( \beta \)-relaxed triangle inequality is that the direct connection between two vertices of a complete graph can be more expensive than a detour. On the other hand, using the \( \beta \)-relaxed triangle inequality, we can bound
3.2. A New Approximation Algorithm for $k$-$\Delta_\beta$OTSP

Figure 3.4: Vertex $r_i$ is the root of tree $T_i$, vertices $x_i$ and $y_i$ are the predecessor and the successor of $r_i$ in $C'$, respectively. Edge $e_i^* = (z_i, r_i)$ is the minimum-cost edge incident to the root $r_i$ in $T_i$. The Hamiltonian cycle $H_i$ is the cycle built by procedure $\text{HCT}^3$ (refined) from Subsection 2.8.1 on the pair $(T_i, e_i^*)$. The merge of $H_i$ to $C'$ is done as follows: We reconnect the predecessor $x_i$ of $r_i$ to the vertex $z_i$. Thus, the Hamiltonian cycle $H$ continues from $x_i$ to $z_i$, then it uses the Hamiltonian cycle $H_i$, and finally, in $r_i$, it is connected back to the edges of $C'$.

Now we are ready to analyze the costs of the paths $L'_i$ and the cost of covering the remaining vertices that are not in any path $L'_i$.

**Lemma 3.2.2.** For $1 \leq i \leq k$, the cost of the path $L'_i$ is bounded from above by

$$\text{cost}(L'_i) \leq \beta^3 \log_2 (3k - 4) \cdot \text{cost}(P_T(s_i, s_{i+1})).$$

**Proof.** According to Lemma 3.2.1, the consecutive vertices of $L_i$ are at most $3k - 4$ edges apart in the corresponding unique path in $T$. Including the additional vertices when creating $L'_i$ from $L_i$ does not increase the distance in $P_T(s_i, s_{i+1})$ between two consecutive vertices. Thus, also in $L'_i$, the distance of consecutive vertices is at most $3k - 4$. Therefore, by Lemma 2.8.3, the statement of this lemma is proved.

**Lemma 3.2.3.** For $1 \leq i \leq q$, the cost of the Hamiltonian cycle $H_i$ is bounded from above by

$$\text{cost}(H_i) \leq (\beta^2 + \beta) \cdot \text{cost}(T_i).$$

**Proof.** Let $T_i$ be a subtree of $T$ from step 6 of Algorithm 6, let $r_i$ be the root of $T_i$, and let $e_i^*$ be the edge of minimal cost that is incident to $r_i$ in $T_i$. Note that $r_i \in C'$ and $(V(T_i) \setminus \{r_i\}) \cap V(C') = \emptyset$. Since $e_i^*$ is a minimal edge of $T_i$ that is incident to the root $r_i$, it is locally minimal (see the definition of $\text{HCT}^3$ (refined)}
in Subsection 2.8.1). Therefore, we can apply the procedure HCT\(^3\) (refined) to the tree \(T_i\) with the initial edge \(e_i^*\).

Note that the procedure HCT\(^3\) (refined) requires that the size of the input tree satisfies \(|V(T_i)| \geq 3\). In the case \(|V(T_i)| = 2\), our claim follows immediately. Otherwise, the output of the procedure HCT\(^3\) (refined) is a Hamiltonian cycle \(H_i\) that contains \(e_i^*\) and its cost is \(\text{cost}(H_i) \leq (\beta^2 + \beta) \cdot \text{cost}(T_i)\) (for \(\beta > 1\)), proving the claim of the theorem.

This implies the main result.

**Theorem 3.2.4.** The approximation ratio of Algorithm 6 solving \(k\Delta_\beta\text{OTSP}\) in time \(O(|V|^2)\) is

\[k\beta\log_2 3(k - 1)\]

**Proof.** It is easy to see that Algorithm 6 computes a Hamiltonian cycle where the special vertices are in the required order. Therefore, from now on we will estimate the approximation ratio. Let us denote by \(\text{OPT}\) an optimal solution of the instance. Since each vertex of \(C'\) is the root of at most one tree \(T_1, \ldots, T_q\), it can be bypassed at most once while connecting the Hamiltonian cycles \(H_1, \ldots, H_q\) to \(C'\). Since edge \(e_i^*\) \((1 \leq i \leq q)\) is an edge of \(T_i\), it is an edge of \(T\) and therefore, applying Lemma 3.2.2, we can estimate the cost factor of the reconnection of \(H_1, \ldots, H_q\) by

\[\gamma = \beta \log_2 3(k - 1)\]

Let us divide the edges of \(T\) into four disjoint groups:

\[
\begin{align*}
S_1 &= \{e_i^* \in E \mid e_i^* \text{ is locally minimal} \land e_i^* \in E(T_i) \land |V(T_i)| = 2 \land 1 \leq i \leq q\} \\
S_2 &= \{e_i^* \in E \mid e_i^* \text{ is locally minimal} \land e_i^* \in E(T_i) \land |V(T_i)| > 2 \land 1 \leq i \leq q\} \\
S_3 &= \{e \in E \mid e \in E(T_i) \setminus \{e_i^*\} \land 1 \leq i \leq q\} \\
S_4 &= E(T) \setminus (S_1 \cup S_2 \cup S_3)
\end{align*}
\]

Now we will prove that for the factors

\[
\begin{align*}
f_1 &= \gamma + 1 \\
f_2 &= \beta^2 + \beta + \gamma \\
f_3 &= \beta^2 + \beta \\
f_4 &= k\gamma
\end{align*}
\]

the inequality

\[
\text{cost}(H) \leq f_1 \cdot \text{cost}(S_1) + f_2 \cdot \text{cost}(S_2) + f_3 \cdot \text{cost}(S_3) + f_4 \cdot \text{cost}(S_4)
\]

holds. Factor \(f_1\) is \(\gamma + 1\), since the cost of reconnecting the edges from \(S_1\) is \(\gamma\) and we have to add the locally minimal edge itself, causing the additional term. Edges from \(S_2\) are inserted into the main cycle with factor \(\gamma\) and, for presence in
3.3. An Improvement of the Algorithm of Böckenhauer et al.

In this section, we continue the study of $k$-$\Delta_\beta$OTSP (recall that $k \geq 4$). As a stepping stone, we use Algorithm 4 from Subsection 2.8.1 which was proposed by Böckenhauer et al. [23]. We modify this algorithm by incorporating three new observations which yields Algorithm 7. This allows us to improve its approximation ratio over the ratio of algorithm studied in Section 3.2 to

$$\frac{3}{2} \cdot \left\lceil \frac{k}{2} \right\rceil \cdot \beta^{2+\log_2 (2\lceil k/2 \rceil +k-3)}.$$ 

This improves over the ratio of the algorithm of Section 3.2 for any pair $(k,\beta)$ with $\beta < (3/4)^{-\log_2 3/8} \approx 1.2254$, except for possibly finitely many (small) values of $k$. The proportion between the approximation ratios of the algorithm presented in this section and the one of Section 3.2 is depicted in Figure 3.5.

First we present our Algorithm 7. In the very end, we discuss how Algorithm 7 can be altered to improve its approximation ratio if $k$ is proportionally small to $n$. 

$H_i$, we gain $\beta^2 + \beta$. Factor $f_3$ follows from Lemma 3.2.3. Using that $\text{cost}(C') \leq \sum_{i=1}^k \text{cost}(L'_i) \leq k \cdot \max_{i=1}^k \text{cost}(L'_i) \leq k \beta^3 \log_2^{3k-4} \text{cost}(S_4)$, we obtain $f_4$. Then we can estimate the cost of $H$:

$$\text{cost}(H) \leq (1 + \gamma) \cdot \text{cost}(S_1) + (\beta^2 + \beta + \gamma) \cdot \text{cost}(S_2) + (\beta^2 + \beta) \cdot \text{cost}(S_3) + k\gamma \cdot \text{cost}(S_4) \leq (\beta^2 + \beta + \gamma) \cdot \text{cost}(S_2) + \max\{1 + \gamma, \beta^2 + \beta, k\gamma\} \cdot (\text{cost}(E(T) \setminus S_2)).$$

From $\beta > 1$ and $k > 2$ ($k \in \mathbb{N}$) we get $\gamma > 1$, $\max\{1 + \gamma, \beta^2 + \beta, k\gamma\} = k\gamma$, $\gamma > \beta^2 > \beta$, and $\beta^2 + \beta + \gamma \leq 3\gamma \leq k\gamma$. Thus, we have

$$\text{cost}(H) \leq k\gamma \cdot \text{cost}(T) \leq k \beta^3 \log_2^{3(k-1)} \cdot \text{cost}(\text{OPT}).$$

We can easily build a minimum spanning tree (see, e.g., [39]) and then search it in breadth-first-search manner in time $O(|V|^2)$. The paths from steps 3 and 4 can be constructed sequentially by a search on the tree starting from a special vertex and ending in the consecutive special vertex. The time complexity of such a simple tree traversal is $O(k \cdot |V|)$. The following step 5 takes $O(k)$ time. The trees $T_1, \ldots, T_q$ with their roots needed in step 6 can be identified by a depth-first or breadth-first search in linear time. Finally, according to Theorem 2.8.10, the step 6 can be performed in quadratic time in respect to the size of each tree. Hence, the overall time-complexity of this step for all the trees is $O(|V|^2)$. The merge in the last step of the algorithm can be also performed in $O(|V|^2)$ which proves the overall time-complexity of our algorithm. \hfill $\square$
Algorithm 7 (An improvement of the algorithm by Böckenhauer et al. [23] for $k$-$\Delta_{\beta}$OTSP)

**Input:** A complete graph $G = (V, E)$ with edge cost function $c: E \rightarrow \mathbb{Q}^+$ that satisfies the $\beta$-relaxed triangle inequality ($\beta > 1$), and a sequence of special vertices $(s_1, \ldots, s_k)$ from $V$.

1: Using some constant-approximation algorithm, construct an approximate TSP solution $C$ on $(G, c)$, disregarding the order on $(s_1, \ldots, s_k)$.

2: Let $P$ be one of the two paths that may be obtained by removing one of the edges incident with $s_1$ in $C$, and let $W = (w_1, \ldots, w_{n-k})$ be the sequence of the non-special vertices in $P$, beginning with the non-special vertex closest to $s_1$ in $P$. Let $f$ be a $[k/2]$-cyclic coloring of non-special vertices of $C$ defined as follows: $f: V \rightarrow \{-1, 0, \ldots, [k/2] - 1\}$ where $f(s_i) = -1$, for all $1 \leq i \leq k$, and $f(w_{i+1}) := i \mod [k/2]$, for all $0 \leq i < n - k$.

3: For $1 \leq i \leq [k/2] - 1$, let $L_{i-1}$ be the subpath in $C$ (either clockwise or counterclockwise) from $s_{2i-1}$ to $s_{2i}$ and then to $s_{2i+1}$, restricted to $s_{2i-1}$, $s_{2i}$, and $s_{2i+1}$ plus all vertices $w$ with $f(w) = i - 1$.

4: If $k$ is odd, let $L_{[k/2]-1}$ be the subpath in $C$ from $s_k$ to $s_1$ that is restricted to $s_k$, $s_1$ and the non-special vertices $w$ that have color $f(w) = [k/2] - 1$.

If $k$ is even, let $L_{[k/2]-1}$ be the subpath in $C$ from $s_{k-1}$ through $s_k$ to $s_1$, restricted to $s_{k-1}$, $s_k$, $s_1$ and the non-special vertices $w$ with $f(w) = [k/2] - 1$.

5: Create new subpaths $L'_0, \ldots, L'_{[k/2]-1}$ from $L_0, \ldots, L_{[k/2]-1}$ by including the non-special vertices bypassed by some $L_i$ as in step 4 of Algorithm 6 (see Figure 3.3 for details).

6: Let $Y$ be the path of $C$ containing all the vertices that are not included in any of the $L'_i$. If such a path does not exist, skip this step. Otherwise, let $x$ and $y$ be the two endpoints of $Y$ and let $s_{2\gamma+3}$ be the special vertex that is neighboring with $x$ in $C$ (see Figure 3.6 for details). Remove the endpoint $s_{2\gamma+3}$ from the subpath $L'_\gamma$ and extend it by including all the vertices $w$ of $Y$ in the direction from $x$ to $y$ colored by $f(w) = \gamma$ and terminate it by the vertex $y$ (if its color differs from $\gamma$).

Extend the subpath $L'_{\gamma+1}$ from $s_{2\gamma+3}$ by including all the vertices of $Y$ from $x$ to $y$ that were not included into $L'_\gamma$ and terminate the modified subpath in vertex $y$.

7: Merge the subpaths $L'_0, \ldots, L'_{[k/2]-1}$ into a cycle $H$.

**Output:** The Hamiltonian cycle $H$.

More precisely, if $n \geq 4k + 2$, we show in the end how to enhance Algorithm 7 to achieve an approximation ratio of

$$\frac{3}{2} \cdot \left\lceil \frac{k}{2} \right\rceil \cdot \beta^{2/\log_2 ([k/2]+k-2)}.$$
3.3. An Improvement of the Algorithm of Böckenhauer et al.

The cycle is then used to construct paths between consecutive special vertices that are then merged together into a cycle where the special vertices are present in the right order. However, during the process of the path construction, some vertices might be omitted and hence are not present anymore in the new cycle. These vertices are in the end somehow added into the cycle and the required Hamiltonian cycle is obtained.

The three observations that are improving Algorithm 4 are as follows.

- We can always connect three vertices present on a cycle in the prescribed order by a single path – the path starts in the first vertex and traverses, either clockwise or counterclockwise, the cycle through the second prescribed vertex to the last vertex.

- The vertices that are bypassed by a subpath, but not included into any subpath, can be just included to the bypassing subpath – the cost of the solution is not increased this way.

- Let \( H \) be a Hamiltonian cycle on a complete weighted graph and let \( L_0, \ldots, L_{p-1} \) be some paths on \( H \) that can be sequentially merged in their endpoints into a walk \( W \), i.e., for all \( 0 \leq i < p \), paths \( L_i \) and \( L_{(i+1) \mod p} \) share one endpoint. Then the vertices of \( H \) that are not covered by \( W \) form a single path \( P \) of \( H \). Moreover, the adjacent vertex of an endpoint of \( P \) in \( H \) is a vertex in which two paths \( L_i \) and \( L_{i+1} \) are merged. The path \( P \) split into two subpaths can be used to extend the paths \( L_i \) and \( L_{i+1} \) such that all vertices of \( H \) are covered.

The first observation allows us to use approximately half of the colors that Algorithm 4 needs. Once we have built the cycle \( C \), we actually do not have to traverse it and connect pairs of consecutive special vertices to obtain the cycle where special vertices are present in the prescribed order. Note that, in Algorithm 4, these connections are denoted as subpaths \( H_i \) (\( 0 \leq i \leq k-1 \)) and are built by
traversing the cycle in the direction so that the special vertex $s_1$ is never bypassed. Instead, we connect an ordered triple of special vertices together in one traversal of the cycle. (If $k$ is odd, the last subpath is connecting just two special vertices – $s_k$ and $s_1$.) In such a traversal, we cannot always avoid the bypasses of the special vertex $s_1$ in the cycle. The advantage of this approach is to reduce the number of traversals each needing a separate color for the set of vertices in the corresponding subpath. Hence, the overall number of colors we need is $\lceil k/2 \rceil$ instead of $k + 1$. The drawback of this approach is that the bypasses might be slightly longer. More precisely, once we are passing the vertex $s_1$ in our traversal, we might have a cut in the coloring: the segment of the cycle between $s_1$ and the next vertex of the same color in the other direction than the one used for the coloring might miss vertices of some colors. This happens if the number of non-special vertices in the cycle is not divisible by the number of colors and thus the cycle does not contain the same number of vertices of each color. Here, the algorithm might be forced to bypass at most two vertices of the same color.

**Lemma 3.3.1.** In step 3 and step 4 of Algorithm 7, at most $2\lceil k/2 \rceil + k - 3$ edges of $C$ are bypassed between two consecutive vertices of subpath $L_i$, for all $0 \leq i \leq \lceil k/2 \rceil - 1$.

Furthermore, a bypass of at most $2\lceil k/2 \rceil + k - 3$ edges can occur only if the vertex $s_1$ is bypassed. All other bypasses jump over at most $\lceil k/2 \rceil + k - 2$ edges.

**Proof.** Let $x$ and $y$ be two consecutive vertices of subpath $L_i$. Since we use a cyclic coloring of the non-special vertices of $C$, the only color cut can be around the vertex $s_1$.

Assume that vertex $s_1$ is bypassed by the edge $xy$ in $L_i$. There are $\lceil k/2 \rceil - 1$ different colors except $i$. For each such color, at most two vertices can be present in between $x$ and $y$ in $C$. In addition, there are at most $k - 2$ special vertices that can be present here as well. Hence, the overall number of vertices between two consecutive vertices on each subpath that is bypassing the color cut is at most $2\lceil k/2 \rceil + k - 4$.

Assume that vertex $s_1$ is not bypassed by the edge $xy$ of $L_i$. Then, since the coloring is cyclic, there is exactly one non-special vertex of every other color between $x$ and $y$. In addition, there are at most $k - 2$ special vertices that can be present here as well. Together, there are at most $\lceil k/2 \rceil + k - 3$ vertices that are bypassed by the edge $xy$, which proves the second part of the claim. \[\square\]

The second observation is used in step 5 of Algorithm 7 in the same way as we used it in step 4 of Algorithm 6 – including bypassed vertices into the subpath that is bypassing them cannot increase the cost of the subpath. We use the same process of the inclusion into subpaths $L_i$ in the construction of the subpaths $L'_i$ as used in Algorithm 6 and hence we do not discuss it further.

At this point, if we sequentially merge the subpaths $L'_0, \ldots, L'_{\lceil k/2 \rceil - 1}$ by their endpoints, we obtain a cycle $Q$ that contains each vertex at most once and that respects the given order on special vertices. However, some vertices of $C$ might not be present in $Q$ as no subpath is bypassing them. This case is covered by
3.3. AN IMPROVEMENT OF THE ALGORITHM OF BÖCKENHAUER ET AL. 67

Figure 3.6: An extension of the paths \( L'_\gamma \) and \( L'_{\gamma+1} \) by the path \( Y \) in step 6 of Algorithm 7. Squares denote special vertices, dots denote non-special vertices. The terminal vertex \( s_{2\gamma+3} \) is removed from \( L_\gamma \), the subpath \( L_\gamma \) is extended by inclusion of all the vertices of color \( \gamma \) in \( Y \) and terminated by the vertex \( y \). The subpath \( L'_{\gamma+1} \), in which the special vertex \( s_{2\gamma+3} \) is preserved, is extended by inclusion of all the remaining vertices of \( Y \) and is terminated by the vertex \( y \).

The third observation – see Figure 3.6 for details. The vertices not included in \( Q \) must form a single path \( Y \) in \( C \) and are included into two consecutive subpaths as follows. Since \( Y \), with the endpoints \( x \) and \( y \), is a path of \( Q \), there must exist subpaths \( L'_\gamma \) and \( L'_{\gamma+1} \) with an endpoint \( s_{2\gamma+3} \) that is adjacent with \( x \). We split \( P \) into two subpaths and extend each of the subpaths \( L'_\gamma \) and \( L'_{\gamma+1} \) by one of these split subpaths of \( P \). The outcome of these extensions is that \( y \) becomes the endpoint of both subpaths and each vertex of \( Y \) is included into exactly one of the extended subpaths. Notice that, in the extension of the subpath \( L'_\gamma \), its special vertex \( s_{2\gamma+3} \) is removed and then it is continued by inclusion of the vertices of the color \( \gamma \) into the subpath up to the vertex \( y \) (inclusively). Since \( P \) was not covered by \( L'_\gamma \), the vertices of color \( \gamma \) are not already included in any subpath and this operation is sound. The other extension can be evaluated in the same way. The subpath \( L'_{\gamma+1} \) (which still contains the special vertex \( s_{2\gamma+3} \)) is extended by including all the remaining vertices of \( P \) into the subpath and is terminated in \( y \). By the same reasoning as above, this operation is also sound. For our convenience of presenting the statements comprehensively, we call the extended subpaths still \( L'_\gamma \) and \( L'_{\gamma+1} \).

After the transformation of the subpaths \( L'_\gamma \) and \( L'_{\gamma+1} \) in step 6, observe that two consecutive vertices in the extended subpaths are bypassing at most \( 2\lceil k/2 \rceil + k - 3 \) edges in \( C \). This is true as (1) the non-extended parts have this property as discussed above; (2) in the extended part of \( L'_\gamma \), only the vertices of color \( \gamma \) are
used and the distance between the last vertex of the color $\gamma$ and $y$, which may be of a different color, cannot be too high as well; (3) the extended part of $L'_{\gamma+1}$ from $s_{2\gamma+3}$ to $y$ contains all the remaining vertices. In particular, this also includes the vertices of the color $\gamma + 1$. Two consecutive vertices of this color are at distance at most $2[k/2] + k - 3$ in the cycle $C$. Note that this is also true for the distance between the last vertex of the extended subpath $x$ and the vertex $s_{2\gamma+3}$. The inclusion of the vertices of the other colors in between the vertices of color $\gamma + 1$ cannot increase the distance.

Lemma 3.3.2. The distance between two consecutive vertices in subpaths $L'_i$ (for $0 \leq i \leq \lceil k/2 \rceil - 1$ and $i \notin \{\gamma, \gamma + 1\}$) and in the extended subpaths $L'_\gamma$ and $L'_{\gamma+1}$ is at most $2[k/2] + k - 3$ on the cycle $C$. Furthermore, a bypass of at most $2[k/2] + k - 3$ edges can occur only if the vertex $s_1$ is bypassed (i.e., if the bypass occurs on the color cut). All other bypasses jump over at most $\lceil k/2 \rceil + k - 2$ edges.

Sketch of the proof. The lemma is proven by the discussion above and a similar reasoning as we used in Lemma 3.3.1.

From the description above, it is clear that the cycle $H$ computed by our algorithm is a feasible solution, i.e., each vertex of the graph is included in $H$ once and the special vertices are present in the given order. From now on, we omit to write colorings and indexing modulo $\lceil k/2 \rceil$, when it is clear from the context.

Now we are ready to estimate the approximation ratio of our algorithm and to calculate in which cases we obtain an improvement over Algorithm 6.

Theorem 3.3.3. For $k \geq 4$, Algorithm 7 computes, in polynomial time, an approximate solution for $k$-$\Delta_\beta$-TSP with a ratio of at most

$$\left\lceil\frac{k}{2}\right\rceil \cdot \min \left\{ 4\beta^1 + \log_2 (2[k/2] + k - 3), 3/2 \beta^2 + \log_2 (2[k/2] + k - 3), (\beta + 1) \beta^1 + \log_2 (2[k/2] + k - 3) \right\}.$$

Proof. Observe that no vertex is present in a subpath $L'_i$ multiple times. Due to the construction of the subpaths, we can always pick a path in $C$ from which we can remove vertices in contiguous blocks of bounded length to obtain the subpath $L'_i$. (This is true also for the extended subpaths.) Therefore, by Lemma 3.3.2 we have

$$\text{cost}(L'_i) \leq \beta^{\log_2 (2[k/2] + k - 3)} \cdot \text{cost}(C).$$

The cost of the merged subpaths forming the cycle $H$ can be estimated as

$$\text{cost}(H) \leq \left\lceil\frac{k}{2}\right\rceil \cdot \beta^{\log_2 (2[k/2] + k - 3)} \cdot \text{cost}(C).$$

In step 1 of the algorithm, we use an approximation algorithm for $\Delta_\beta$-TSP which influences the overall ratio. The currently best approximation ratio for $\Delta_\beta$-TSP is

$$\min \left\{ 4\beta, \frac{3}{2} \beta^2, \beta^2 + \beta \right\}.$$
3.3. AN IMPROVEMENT OF THE ALGORITHM OF BÖCKENHAUER ET AL.

and can be achieved by the best out of the currently known approximation algorithms from [6,17,22] (see Subsection 2.8.1 for details). This gives us the claimed ratio.

In the following paragraphs, we discuss the improvement in the approximation ratio depending on the approximation algorithm for $\Delta_\beta$-TSP plugged in in step 1. Recall that $k \geq 4$, otherwise any Hamiltonian cycle is a feasible solution.

**The improvement obtained if the algorithm of Andreae [6] is used.** In this paragraph, we assume that, in step 1 of our algorithm, we plug in the $(\beta + \beta^2)$-approximation algorithm for $\Delta_\beta$-TSP from [6].

To see for which instances our algorithm is an improvement over Algorithm 6 of Section 3.2, we need to relate the approximation ratios of both the algorithms and solve the inequality

$$\left\lceil \frac{k}{2} \right\rceil \cdot (1 + \beta) \cdot \beta^{1+\log_2(2\lceil k/2 \rceil + k - 3)} \leq k \beta^{\log_2(3k-3)}. \tag{3.1}$$

To show that we do not have any improvement for instances with $k \geq 4$, we first compute a lower bound of the left-hand side of inequality (3.1). In the calculation we use two simple estimations: $(1 + \beta) > 2$ and $\lceil k/2 \rceil \geq k/2$. This implies

$$k \beta^{\log_2(4k-6)} < \left\lceil \frac{k}{2} \right\rceil \cdot (1 + \beta) \cdot \beta^{1+\log_2(2\lceil k/2 \rceil + k - 3)} \leq k \beta^{\log_2(3k-3)}.$$

Our algorithm might admit a better approximation ratio than Algorithm 6 only if the necessary condition

$$k \beta^{\log_2(4k-6)} < k \beta^{\log_2(3k-3)}$$

is satisfied.

Since $\beta > 1$, the necessary condition can be satisfied only if

$$\beta^{\log_2 \frac{3k-3}{4k-6}} > 1,$$

which holds only if $\frac{3k-3}{4k-6} > 1$. Thus, our approximation algorithm might give a better approximation only for instances with $k < 3$.

**The improvement obtained if the algorithm of Bender and Chekuri [17] is used.** We assume that the $4\beta$-approximation algorithm for $\Delta_\beta$-TSP from [17] is applied in step 1 of our algorithm.

We use the same reasoning as in the previous paragraph. Our algorithm has a better approximation ratio than Algorithm 6 of Section 3.2 if inequality (3.2) holds:

$$4 \cdot \left\lceil \frac{k}{2} \right\rceil \cdot \beta^{1+\log_2(2\lceil k/2 \rceil + k - 3)} \leq k \beta^{\log_2(3k-3)}. \tag{3.2}$$
We will show that, for $k \geq 4$, the algorithm does not lead to an improvement in the approximation ratio. First, we bound the approximation ratio of our algorithm from below by using the simple estimation $\lceil k/2 \rceil \geq k/2$. This yields

$$2k\beta^\log_2(4k-6) \leq 4 \cdot \left\lceil \frac{k}{2} \right\rceil \cdot \beta^{1+\log_2(2\lceil k/2 \rceil+k-3)} \leq k\beta^\log_2(3k-3).$$

Hence the necessary condition when our algorithm might give a better ratio is given as

$$\beta^\log_2(4k-6) < 2\beta^\log_2(4k-6) \leq \beta^\log_2(3k-3).$$

Again, we can use the same argument as in the previous paragraph. Since $\beta > 1$, to satisfy the necessary condition, we need

$$\beta^\log_2 \left(\frac{3k-3}{4k-6}\right) > 1$$

and hence $\frac{3k-3}{4k-6} > 1$ which holds only if $k < 3$.

**The improvement obtained if the algorithm of Böckenhauer et al. [22] is used.** Finally, we use the $\frac{3}{2}\beta^2$-approximation algorithm for $\Delta_\beta$-TSP from [22] in step 1 of our algorithm. In the following, we show that, indeed, this gives us an improvement in approximation ratio over Algorithm 6 for infinitely many pairs $(k, \beta)$.

To simplify the calculations, we bound the approximation ratio of our algorithm from above.

$$\frac{3}{2} \cdot \left\lceil \frac{k}{2} \right\rceil \cdot \beta^{2+\log_2(2\lceil k/2 \rceil+k-3)} \leq \frac{3}{2} \cdot \frac{k+1}{2} \cdot \beta^{2+\log_2(k+1+k-3)} \leq \frac{3}{4} \cdot (k+1) \cdot \beta^\log_2(8k-8).$$

Algorithm 7 has a better approximation ratio if inequality (3.3) with the related approximation ratios of the two algorithms, is satisfied:

$$\frac{3}{4} \cdot (k+1) \cdot \beta^\log_2(8k-8) \leq k \cdot \beta^\log_2(3k-3). \quad (3.3)$$

We apply the operation of the binary logarithm to the both sides of the inequality and then we transform the inequality by the standard operations on logarithms as follows. This yields

$$\log_2 \frac{3}{4} + \log_2(k+1) + \log_2 \beta \cdot \log_2(8k-8) \leq \log_2 k + \log_2 \beta \cdot \log_2(3k-3)$$

which can be simplified to

$$\log_2 \beta \cdot (\log_2(8k-8) - \log_2(3k-3)) \leq \log_2 k - \log_2 \frac{3}{4} - \log_2(k+1).$$
This is equivalent to
\[ \log_2 \beta \leq \frac{\log_2 \frac{4k}{3k+3}}{\log_2 \frac{8k-8}{3k-3}}. \]

We further modify the right-hand side of the last inequality to its final form
\[ \log_2 \beta \leq \frac{\log_2 \frac{3k+3}{4k}}{\log_2 \frac{3}{8}}. \] (3.4)

The function on the right-hand side of inequality (3.4) for \( k \geq 4 \) is increasing. Hence the improvement is obtained for all \( \beta \) such that
\[ \log_2 \beta < \lim_{{k \to \infty}} \frac{\log_2 \frac{3k+3}{4k}}{\log_2 \frac{3}{8}} = \log_3/8 \frac{3}{4}. \]

Which allows us to estimate \( \beta \) to be \( \beta < \left( \frac{3}{4} \right)^{\frac{1}{\log_2 3/8}} < 1.225444330894. \)

Note that we have shown that, for arbitrary \( 1 < \beta < \left( \frac{3}{4} \right)^{\frac{1}{\log_2 3/8}} \), there exists an initial value \( k_0 \geq 4 \) such that, for all \( k \geq k_0 \), the approximation ratio of Algorithm 7 is better than the ratio of Algorithm 6 for the pairs \((k, \beta)\) – see Figure 3.7, where these pairs are depicted. For finitely many values of \( k < k_0 \), our algorithm will not achieve an improvement.

We summarize our observation into the final theorem.

Figure 3.7: The graph shows the pairs \((k, \beta)\) for which Algorithm 7 has provably better approximation ratio than Algorithm 6.
Chapter 3. The Ordered Traveling Salesman Problem

Theorem 3.3.4. For any $1 < \beta < (3/4)^{\frac{1}{\log_2 3/8}}$, there exists a fixed $k_0 > 3$ such that, for all $k \geq k_0$, the approximation ratio of Algorithm 7 is better than the ratio achieved by Algorithm 6 for the pairs $(\beta, k)$. In these cases, the approximation ratio of Algorithm 7 is at most

$$\left\lceil \frac{k}{2} \right\rceil \cdot \frac{3}{2} \beta^{2+\log_2(2\lceil k/2 \rceil + k - 3)}.$$

3.4 Further Improvements

All our bypasses in Algorithm 7 that are not bypassing the color cut around $s_1$ contain at most $\lceil k/2 \rceil + k - 3$ vertices. Only the bypasses that occur around this cut are the bottleneck of our estimations and are increasing the estimated length of the bypasses from $\lceil k/2 \rceil + k - 2$ to $2\lceil k/2 \rceil + k - 3$ in all our calculations. For the ease of the presentation, we call a bypass that bypasses also vertex $s_1$ (and hence the color cut) a color-cut bypass.

We estimated that each color-cut bypass contains at most $2 \cdot (\lceil k/2 \rceil - 1) + x$ vertices, where $x$ is the number of special vertices that are bypassed. Hence, the main idea of the improvement is to choose a vertex from which the coloring is started (and which also places the color cut into the cycle) such that each color-cut bypass contains a minimum number of special vertices.

Let $d$ be a positive integer whose precise value is specified later. Let us divide the cycle $C$ into $d$ parts of roughly the same size. From the pigeon-hole principle, one of these parts contains at most $\lfloor k/d \rfloor$ special vertices and at least $\lfloor n/d \rfloor$ of all the vertices of $C$. If this part of the cycle contains at least $3 \cdot \lceil k/2 \rceil - 1$ non-special vertices, we can place the color cut into this part such that starting and ending vertices of all color-cut bypasses are present in this part. Then, the number of bypassed vertices in each color-cut bypass is at most $2 \cdot (\lceil k/2 \rceil - 1) + \lceil k/d \rceil$. Hence, if we choose $d$ to be large (but still $d \leq n$), the number of vertices bypassed by each color-cut bypass converges to $k + \frac{k}{n}$.

The last part that we have to show is that, indeed, the chosen part contains at least $3 \cdot \lceil k/2 \rceil - 1$ vertices with respect to $d$ and $n$. We can estimate the number of non-special vertices in the chosen part of $C$ which is at least

$$\left\lceil \frac{n}{d} \right\rceil - \left\lfloor \frac{k}{d} \right\rfloor \geq \frac{n - d + 1}{d} - \frac{k}{d} = \frac{n - d + 1 - k}{d}.$$

Hence, if

$$\frac{n - d + 1 - k}{d} \geq 3 \cdot \left\lceil \frac{k}{2} \right\rceil - 1,$$

we can place the color cut into the chosen part. The inequality above is true if

$$\frac{n - d + 1 - k}{d} \geq 3 \cdot \frac{k + 1}{2} - 1,$$
and hence,
\[ 2 \cdot (n - d + 1 - k) \geq 3d \cdot (k + 1) - 2d. \]

If we further modify the inequality, we are able to bound \( d \) by
\[ d \leq \frac{2(n + 1 - k)}{3k + 3}. \] (3.5)

Any positive integer value \( d \) satisfying the inequality (3.5) can be used to divide the cycle into \( d \) parts, one of these parts contains only \( \lceil k/d \rceil \) special vertices which allows each color-cut bypass to contain at most \( 2 \cdot (\lceil k/2 \rceil - 1) + \lfloor k/d \rfloor \) vertices. To minimize the length of the color-cut bypass, we need to minimize \( \lfloor k/d \rfloor \) and hence \( d \) has to be maximal.

If the number of the special vertices \( k \) is rather small in proportion to \( n \), then \( d \) is rather large and thus the number of special vertices in the chosen part is small, just \( \lfloor k/d \rfloor \).

Since our coloring achieves bypasses of \( \lceil k/2 \rceil + k - 3 \) vertices for each segment of the cycle without the color cut, the partition of the cycle \( C \) into many parts does not help in some point to decrease the length of bypasses.

If \( d = 2 \) (and \( n \geq 4k+2 \)), the color-cut bypasses contain at most \( 2(\lceil k/2 \rceil - 1) + \lceil k/2 \rceil = \lceil k/2 \rceil + k - 2 \) vertices which is by 1 vertex more than the number of vertices bypassed in the remainder of the cycle. Therefore, if \( n \geq 4k + 2 \), Algorithm 7 enhanced by the special placement of the color cut has an approximation ratio of
\[ \left\lceil \frac{k}{2} \right\rceil \cdot \min \left\{ 4\beta^{1+\log_2(\lceil k/2 \rceil + k-1)}, \frac{3}{2}\beta^{2+\log_2(\lceil k/2 \rceil + k-1)}, (\beta + 1)\beta^{1+\log_2(\lceil k/2 \rceil + k-1)} \right\}. \]

If \( k \geq 4 \), then \( \lfloor k/3 \rfloor \leq \lceil k/2 \rceil - 1 \). Thus, if in addition \( d = 3 \) (and hence \( n \geq \frac{11k+7}{2} \)), the color-cut bypasses become at most as long as the non-color-cut bypasses and the enhanced Algorithm 7 is achieving ratio
\[ \left\lceil \frac{k}{2} \right\rceil \cdot \min \left\{ 4\beta^{1+\log_2(\lceil k/2 \rceil + k-2)}, \frac{3}{2}\beta^{2+\log_2(\lceil k/2 \rceil + k-2)}, (\beta + 1)\beta^{1+\log_2(\lceil k/2 \rceil + k-2)} \right\}. \]

We can make a similar calculation as in the analysis of the original Algorithm 7 to see which of the three possible approximation algorithms for the Hamiltonian cycle computed in step 1 of the algorithm are improving the ratio over Algorithm 6.

For the latter approximation ratio with bypasses of length \( \lceil k/2 \rceil + k - 2 \) edges, we can do similar calculations as we did if the \( 3/2 \beta^2 \)-approximation algorithm for \( \Delta_\beta \)-TSP from Böckenhauer et al. [22] is used.

The inequality (3.3) is changed to
\[ \frac{3}{4} \cdot (k + 1) \cdot \beta^{\log_2(6k-6)} \leq k \cdot \beta^{\log_2(3k-3)}, \]
and the inequality (3.4) changes to

\[ \log_2 \beta \leq \frac{\log_2 \frac{4k}{3k+3}}{\log_2 \frac{6k-6}{3k-3}} \]

Again, the function on the right-hand side is increasing for \( k \geq 4 \) and, hence, an improvement is obtained for all \( \beta \) (and all values of \( k \) from some point) such that

\[ \log_2 \beta < \lim_{k \to \infty} \log_2 \frac{\frac{4k}{3k+3}}{\frac{6k-6}{3k-3}} = \log_2 \frac{4}{3}. \]

This allows us to estimate \( \beta \) to satisfy

\[ \beta < \frac{4}{3}. \]

Furthermore, note that we achieved an improvement only for values \( \beta < 2 \). In these cases, the \( \frac{3}{2} \beta^2 \)-approximation of Böckenhauer et al. is achieving the best approximation ratio out of all three approximation algorithms. Thus, the usage of the other two approximation algorithms for \( \Delta_{\beta}-\text{TSP} \) in step 1 of the algorithm cannot bring further improvements.

### 3.5 Summary

In this chapter, we investigated approximation algorithms for \( k-\Delta_{\beta}\text{OTSP} \). We presented gradual improvements in approximability over the algorithm of \([23]\) which was the previously best known approximation algorithm for the problem. In Section 3.2, we studied an algorithm that is based on a minimum spanning tree and, in Section 3.3 and Section 3.4, we improved the algorithm of \([23]\) based on a Hamiltonian cycle to achieve a better approximation ratio for instances with \( \beta \) close to one.

We would like to mention here that the coloring in Algorithm 6 that is discussed in Section 3.2 was improved and is submitted to a journal. The improvement in the coloring decreases the distance between two consecutive vertices in the paths. The approximation ratio of the submitted algorithm is decreased to

\[ k \beta \log_2 \left( \left\lfloor \frac{3k}{2} \right\rfloor + 1 \right). \]

In the paper, it is proved that the coloring is optimal (up to a small constant) and hence, it seems, that to further improve the approximation ratio one has to employ a new idea.

Nevertheless, the approximation algorithm presented in Section 3.3 achieves still a better approximation ratio than the submitted algorithm for values of \( \beta \)
that are close to one. More precisely, Algorithm 7 of Section 3.3 achieves a better ratio for pairs \((k, \beta)\) with

\[
\beta < \left(\frac{3}{4}\right)^{\frac{1}{\log_2(3/16)}} \approx 1.12650639945
\]

and \(k \geq 4\) except for a finitely many (small) values of \(k\). If \(n \geq (11k + 7)/2\), our enhancement of Algorithm 7 from Section 3.4 is achieving an improvement for \(\beta < 2/\sqrt{3} \approx 1.1547\) (again, possibly, except for finitely many small values of \(k\)).

There are still several open questions related to \(k-\Delta_3\) OTSP and \(k\)-OTSP. First of all, we still do not have hard instances for our algorithms which could suggest whether our estimations are tight or there is still space for improvements. Furthermore, both our algorithms are using a “static” coloring of vertices that does not reflect the actual situation in the initial Hamiltonian cycle. We believe that the approximation ratio, especially the part that is dependent on the length of bypasses, could be improved by a more sophisticated coloring. Our final enhancement of Algorithm 7 can be applied only if \(n \geq 4k + 2\), i.e., if the number of special vertices is not too high. However, it seems that the cases where \(n\) and \(k\) are almost the same are the hardest ones. For these cases, one may consider an approximation algorithm that is very different in its used structures from all our approaches. Another very interesting open problem is the question whether the approximation ratio of an algorithm for \(k-\Delta_3\) OTSP has to be dependent on \(k\) or not. One possible approach to decrease the approximation ratio significantly in proportion to \(k\) can consider approximation algorithms with running time exponential in \(k\) but possibly with a constant approximation ratio independent of \(k\).
Chapter 4

The Steiner Cycle Problem

Many practical problems in operations research can be formulated as variants of the famous traveling salesman problem (TSP). For a complete graph and an edge cost function, the TSP asks to build a minimum-cost cycle that contains all the vertices, each exactly once. Another prominent problem is the well-known minimum Steiner tree (STP) problem, where, given a weighted complete graph and a subset of vertices, called terminals, the goal is to find a minimum-cost connected subgraph containing all terminals. An overview on known facts about both problems is presented in Section 2.8.

In this chapter, we consider a new modification of TSP to an STP-like problem that we refer to as the minimum Steiner cycle problem (SCP). Here, we deal with its first characterization and give some hardness and approximation results. In Section 4.1, we define SCP formally and we discuss its related problems. Afterwards, in two sections, we discuss the hardness and approximability results of various restricted subclasses of SCP. We conclude the chapter by discussing open problems in Section 4.4.

4.1 Definitions and Overview

The problem investigated in this chapter is formally defined as follows.

**Problem 4.1.1.** The Steiner cycle problem (SCP) is the following optimization problem:

**Input:** A complete graph $G = (V, E)$ with an edge cost function $c: E \to \mathbb{Q}^+$ and a set $K \subseteq V$ of $k$ special vertices.

**Feasible solutions:** Any cycle $C$ of $G$ that contains all the vertices of $K$.

**Costs:** The cost of the edges in the cycle $C$, i.e., $\text{cost}(C) = \sum_{e \in E(C)} c(e)$.

**Goal:** Minimization.
For a graph $G = (V, E)$, a subset $K \subseteq V$ and a cost function $c: E \rightarrow \mathbb{Q}^+$, we refer to the subgraph of $G$ on the vertices from $K$ by $G(K, c)$. Recall that the complete distance network $G_D(S)$ of a graph $G$ is the complete graph defined on the vertices of $S$ and with edge costs that correspond to the cost of the cheapest path (with respect to $c$) connecting its endpoints in $G$.

We are characterizing several variations of this problem. The SCP problem with a constant number $k$ of special vertices is denoted as $k$-SCP. When the cost function of the graph satisfies the triangle inequality, we denote the problem variants as $k$-$\Delta$SCP and $\Delta$SCP, respectively. When the cost function satisfies the $\beta$-relaxed triangle inequality, we use the notation $k$-$\Delta_\beta$SCP and $\Delta_\beta$SCP, respectively.

The paper [59] gives an alternate, more general definition of the SCP. The definition in [59] includes penalties for unvisited vertices. It can easily be shown that, in the general case, the version with vertex penalties can be reduced to a version without them – however, this reduction may break other properties such as the non-negativity of edge costs and triangle inequalities between edge costs. Hence, in order to investigate the problem in more detail, we need to distinguish between these two definitions. As our definition matches the original Steiner tree problem more closely, we suggest to use the name “SCP with vertex penalties” for the general definition from [59].

In [59], the author considers the SCP with vertex penalties as a 0-1 integer linear problem, examines the polytope defined by the set of SCP solutions and develops two lifting procedures to extend facet-defining inequalities from the traveling salesman polytope. These results are not related to our line of research.

In the case where the number of special vertices in SCP is constant, we discuss two results here: The main result is that, unless $P = NP$, there is no approximation algorithm for SCP on directed graphs with a constant number of special vertices with an approximation ratio polynomial in the input size. This even holds for only four special vertices. For undirected graphs with $\beta$-relaxed triangle inequality, we give a $(\beta^2 + \beta)$-approximation algorithm. Note that the general SCP on undirected graphs without restriction on the edge cost is not approximable by any polynomial factor as the same holds for TSP [98]. This also carries over to SCP on both directed and undirected graphs.

### 4.2 Hardness of $k$-SCP on Directed Graphs

In this section, we prove that 4-SCP on directed graphs is not approximable with a ratio polynomial in the input size. We show this by reducing the following $NP$-complete problem to the 4-SCP.
4.2. Hardness of $k$-SCP on Directed Graphs

**Problem 4.2.1.** The directed 2-vertex-disjoint path problem (Dir-2-VDP) is the following decision problem:

**Input:** A directed graph $G = (V, E)$ and four distinct vertices $s_0, s_1, t_0, t_1 \in V$.

**Decision:** Does $G$ contain a path from $s_0$ to $t_0$ and a path from $s_1$ to $t_1$ that are vertex-disjoint?

**Lemma 4.2.2** (Fortune et al., [55]). The directed 2-vertex-disjoint path problem is $\mathcal{NP}$-complete.

**Theorem 4.2.3.** For any $k > 3$, there is no approximation algorithm for $k$-SCP on directed graphs with an approximation ratio polynomial in the input size, unless $\mathcal{P} = \mathcal{NP}$.

**Proof.** Assume that the directed 4-SCP is approximable with ratio $p(n) \geq 1$, where $p$ is a polynomial and $n$ is the number of vertices in the graph. We show how to decide Dir-2-VDP by using the directed 4-SCP approximation algorithm with ratio $p(n)$. Since Dir-2-VDP is $\mathcal{NP}$-complete, and under the assumption that $\mathcal{P} \neq \mathcal{NP}$, this leads to a contradiction, hence, such approximation algorithm cannot exist.

Let $G = (V, E)$ and $s_0, s_1, t_0, t_1$ be an instance of Dir-2-VDP and let $W := p(|V|) \cdot |V| + 1$. We build a complete directed graph $G' = (V, V \times V \setminus \{v \overleftarrow{v} \mid v \in V\})$ where the cost function $c$ is defined as follows (see Figure 4.1).

- $c(\overrightarrow{t_i s_{1-i}}) = 1, i \in \{0, 1\},$
- $c(\overrightarrow{t_i x}) = W, x \in V \setminus \{s_{1-i} \mid i \in \{0, 1\}\},$
- $c(\overrightarrow{x s_i}) = W, x \in V \setminus \{t_{1-i} \mid i \in \{0, 1\}\},$
- the costs of all other edges from $E$ are set to 1,
- the costs of all remaining edges that are not in $E$ are set to $W$.

![Figure 4.1: The edge cost function for graph $G'$](image-url)
Let the Steiner cycle $S$ be the outcome of the $p(|V|)$-approximation algorithm for 4-SCP on graph $G'$ and special vertices $s_0, s_1, t_0, t_1$. We prove in the following that, the graph $G$ contains two vertex-disjoint paths $s_0 \sim t_0$ and $s_1 \sim t_1$ if and only if $\text{cost}(S) < W$.

Assume that $G$ contains two vertex-disjoint paths $s_0 \sim t_0$ and $s_1 \sim t_1$. Then the sequence $s_0 \sim t_0 s_1 \sim t_1 s_0$ forms a Steiner cycle of cost at most $|V| < W$ proving the first implication.

We prove the other direction by contradiction: Assume that $G$ does not contain vertex-disjoint paths $s_0 \sim t_0$ and $s_1 \sim t_1$ and $\text{cost}(S) < W$. Since $\text{cost}(S) < W$, cycle $S$ contains edges of cost 1 only. Therefore, except for the edges $t_0 s_1$ and $t_1 s_0$, all other edges of $S$ are present also in the graph $G$. We show that the only order in which the special vertices can appear in $S$ is $s_0, t_0, s_1, t_1$. The paths between these special vertices in the Steiner cycle $S$ identify the paths $s_0 \sim t_0$ and $s_1 \sim t_1$ which will lead to a contradiction.

First start from the special vertex $t_1$. Since all other of its outgoing edges are expensive, the following vertex on $S$ is the special vertex $s_0$. Then the following special vertex on the cycle $S$ has to be $t_0$. Otherwise, the special vertex $s_1$ would be connected in the cycle $S$ after the special vertex $s_0$ by some too expensive edge. Then, since all other outgoing edges from the special vertex $t_0$ are too expensive, the special vertex $t_0$ is connected in $S$ directly to the special vertex $s_1$ by the directed edge $t_0 s_1$. Since all vertices in $S$ occur exactly once and the directed edges present in $S$, except the two connecting edges $t_0 s_1$ and $t_1 s_0$, are present in graph $G$, we found two vertex-disjoint paths $s_0 \sim t_0$ and $s_1 \sim t_1$. This is a contradiction.

Corollary 4.2.4. The problem of finding a minimum Steiner cycle on $k > 3$ special vertices in a directed graph is NP-hard.

We also have the following positive result on solvability of $k$-SCP. For $k = 1$ the problem is not much of interest.

Theorem 4.2.5. The problem of finding a minimum Steiner cycle with $k = 2$ special vertices in both directed and undirected graphs is in $\mathcal{P}$.

Proof. The problem can be solved by employing an algorithm for minimum-cost maximum flow either in directed or undirected graphs. For details about the minimum-cost maximum flow problem, see e.g., [3].

The flow network is built from the graph $G$ by duplicating vertices – for each vertex in $G$, we have an incoming and an outgoing vertex. For each edge $\overrightarrow{uv}$ of $G$ we add an edge connecting the outgoing vertex for $u$ with the incoming vertex for $v$. The cost of such an edge corresponds to its original cost in $G$, the capacity is set to one. The incoming vertex is connected with its corresponding outgoing vertex with an edge of cost zero and a capacity one. The source is connected to the outgoing vertex of one of the two special vertices, the incoming vertex of the other special vertex is connected to the sink. Both connecting edges have cost zero and capacity two.
Observe that $G$ contains the two disjoint paths if and only if the maximum flow in the network is two, each unit flow corresponds to one path. The vertex-disjointness requirement is enforced by the edges connecting incoming and outgoing vertices with capacity one not permitting two paths passing through a single vertex of $G$.

The minimality of the cost allows us to find the edges of the minimal overall cost.

The undirected version of $k$-SCP seems to be different from the directed one, similarly as there are differences between directed TSP and undirected TSP. The undirected version of the decision problem of finding $k$ vertex-disjoint paths in a graph ($k$ is constant) is in $\mathcal{P}$, as it was shown by Robertson and Seymour [93]. The problem of deciding whether an undirected graph contains edge-/vertex-disjoint paths between two distinct pairs of vertices is known to be in $\mathcal{P}$ due to [50]. On the other hand, in the same settings for vertex-disjoint paths, where one requires the path between the first pair of vertices to be the shortest and allows the path between the second pair to be an arbitrary one, is $\mathcal{NP}$-complete even for undirected graphs with unit edge lengths due to [50].

To conclude this discussion, the hardness of $k$-SCP on undirected graphs is open. We expect that the problem is $\mathcal{NP}$-hard for some high constant $k$, but the previous results mentioned above might be an evidence that the problem is still in $\mathcal{P}$ for a small number of special vertices.

### 4.3 Approximating the Minimum Steiner Cycle Problem

In this section, we discuss several approximation algorithms for various settings of SCP, in particular, undirected versions of $k$-\(\Delta\beta\)SCP and $\Delta\beta$SCP, and both directed and undirected versions of $\Delta$SCP.

We use the following notation. Let $G = (V,E)$ be a directed/undirected complete graph, let $K \subseteq V$ be a set of special vertices and let $c$ be an edge cost function of $G$ satisfying the ($\beta$-relaxed) triangle inequality. Let $OPT$ be an optimal solution of the SCP in the setting that is particularly used.

### 4.3.1 Approximation of Directed and Undirected $k$-$\Delta$SCP and $\Delta$SCP

In both the directed and undirected case, we observe that, due to the triangle inequality, the cost of the direct edge between two vertices is not more expensive than any bypass. Therefore, in this setting, the Hamiltonian cycle on the subgraph of $G$ induced by the vertices from $K$ is the minimum Steiner cycle of $G$ with special vertices $K$ (with respect to the edge cost function $c$).

In other words, the application of an $\alpha$-approximation algorithm for metric TSP on the induced subgraph gives us an $\alpha$-approximation of directed/undirected
\( \Delta \text{SCP} \). On the other hand, \( \Delta \text{SCP} \) is a generalization of the metric TSP which allows us to approximate metric TSP as good as \( \Delta \text{SCP} \).

**Theorem 4.3.1.** \( \Delta \text{SCP} \) is approximable with factor \( \frac{3}{2} \) on undirected graphs and with factor \( \frac{2}{3} \log_2 k \) on directed graphs with \( k \) special vertices.

**Proof.** The result for undirected graphs follows from the Christofides’ algorithm (see Theorem 2.8.5 or [37] for details). The result for directed graph follows from result of Feige and Singh (see Theorem 2.8.6 or [51] for details).

In the setting where the number of special vertices is constant, we simply try all possible cycles on the special vertices and we pick the one that is the cheapest. This is an optimal solution of our problem as the triangle inequality guarantees that direct edges between vertices are cheaper than bypasses. Thus we can conclude:

**Theorem 4.3.2.** For any constant \( k \in \mathbb{N} \), \( k \Delta \text{SCP} \) on directed and undirected graph is in \( \mathcal{P} \).

### 4.3.2 Approximation of Undirected \( \Delta_\beta \text{SCP} \) and \( k \Delta_\beta \text{SCP} \)

In this subsection we discuss an approximation algorithm for \( \Delta_\beta \text{SCP} \) that can be used for \( k \Delta_\beta \text{SCP} \) as well. The algorithm is presented as Algorithm 8 and is achieving a ratio of \((\beta + \beta^2)\).

**Algorithm 8** (A \((\beta + \beta^2)\)-approximation for \( \Delta_\beta \text{SCP} \) and \( k \Delta_\beta \text{SCP} \))

**Input:** A complete graph \( G = (V, E) \) with edge cost function \( c : E \to \mathbb{Q}^+ \) that satisfies the \( \beta \)-relaxed triangle inequality \((\beta > 1)\), and a set \( K \) of \( k \) special vertices of \( V \).

1. Let \( G_D(K) \) be a complete distance network of \( G \) on the vertices from \( K \).
2. Compute a minimum spanning tree \( T_D \) on \( G_D(K) \).
3. Let \( G^* \) be an expansion of \( T_D \) in \( G \), i.e., the edges of \( G^* \) correspond to the paths used as edges in \( E(T_D) \) and the vertices of \( G^* \) are precisely the vertices used in these paths.
4. Compute a minimum spanning tree \( T \) in \( G^* \).
5. Compute a cycle \( C \) by application of the HCT\(^3\) (refined) function of Andreae on \( T \) and \( G \) (see Algorithm 3 of Subsection 2.8.1 or [6] for more details).

**Output:** The Hamiltonian cycle \( C \).

The presented approximation algorithm first builds a tree \( T \) that spans all the special vertices and whose cost is not higher than the cost of the optimal solution. Then we apply the \((\beta + \beta^2)\)-approximation algorithm for building a Hamiltonian cycle (for details see Algorithm 3 from Subsection 2.8.1 or [6]) to obtain a Steiner cycle whose cost is a constant factor away from the cost of the optimal solution.

The tree \( T \) is constructed as follows. We build the distance network \( G_D(K) \) and compute a minimum-cost spanning tree \( T_D \) for it. Then we transform \( T_D \) back to the graph \( G \) by replacing the edges by the corresponding cheapest paths.
(we only use those vertices that are involved in these paths). In this graph, we might have some extra edges that form cycles. Therefore, we build a minimum spanning tree on this graph and we denote it as $T$. Assume now that we remove the most expensive edge from the optimal solution $OPT$ to obtain a path $P$. This path sequentially connects the special vertices from $K$. We can replace each such path with the cheapest path between the two special vertices obtaining some new graph $G' = (V', E')$, $V' \subseteq V$, $E' \subseteq E$. Our tree $T$ is the cheapest subgraph of $G$ connecting all special vertices and therefore

$$
cost(T) \leq cost(G') \leq cost(P) \leq cost(OPT).
$$

The work of Andreae (Subsection 2.8.1 or [6]) presents one of the approximation algorithms for dealing with $\Delta_\beta$-TSP. The algorithm is very general: After plugging in a tree, it recursively builds a Hamiltonian cycle that contains all the vertices of the given tree and has bounded cost. In our case, we use the algorithm to solve $k$-$\Delta_\beta$SCP and $\Delta_\beta$SCP. According to Theorem 2.8.10, we plug in our tree $T$ and we obtain a Hamiltonian cycle $C$ that contains all the vertices of the tree $T$ (but may contain other edges of the complete graph $G$). Since $T$ contains all the special vertices, $C$ is a Steiner cycle of graph $G$. Theorem 2.8.10 gives the bound

$$
cost(C) \leq (\beta^2 + \beta)cost(T)
$$

on the cost of $C$. In both $k$-$\Delta_\beta$SCP and $\Delta_\beta$SCP, we have $cost(T) \leq cost(OPT)$, giving us the bound

$$
cost(C) \leq (\beta^2 + \beta)cost(T) \leq (\beta^2 + \beta)cost(OPT).
$$

**Theorem 4.3.3.** The undirected versions of $k$-$\Delta_\beta$SCP and $\Delta_\beta$SCP are approximable with a factor of $(\beta^2 + \beta)$ using the $HCT^3$ (refined) algorithm from Subsection 2.8.1 or [6].

### 4.4 Summary

In this chapter, we have presented hardness results and approximation algorithms for various settings of the minimum Steiner cycle problem that is a hybrid between TSP and STP. Our results are summarized in Table 4.1 and Table 4.2.

In Table 4.1 the results for the cases with non-constant number of special vertices are presented. In this setting, SCP is at least as hard as TSP since all vertices of the graph can be picked to the special set of vertices. Therefore, improving any approximation ratio of our cases beyond the best known ratio of corresponding TSP would directly imply the improvement of the TSP approximation. The directed case seems to be harder. It is still open how good the directed TSP with $\beta$-relaxed triangle inequality can be approximated. This carries over to our $\Delta_\beta$SCP. Note that, in all the cases except the latter mentioned setting, we have matched the best known approximation of TSP.
Chapter 4. The Steiner Cycle Problem

Table 4.1: An overview of the approximation ratios of the algorithms analyzed in this chapter for the setting with a non-constant number of special vertices. By $\Delta$, $\Delta_{\beta}$ and “general” we denote the different restrictions on the cost function, $k$ denotes the number of special vertices in the graph and “hard” expresses that the problem is not approximable with a polynomial that is dependent on the size of the input.

<table>
<thead>
<tr>
<th>$c$ function</th>
<th>directed $G$</th>
<th>undirected $G$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta$</td>
<td>$\frac{2}{3} \log_2 k$</td>
<td>$\frac{3}{2}$</td>
</tr>
<tr>
<td>$\Delta_{\beta}, \beta &gt; 1$</td>
<td>$\beta^2 + \beta$</td>
<td>hard</td>
</tr>
<tr>
<td>general</td>
<td>hard</td>
<td>hard</td>
</tr>
</tbody>
</table>

Table 4.2: An overview of the approximation ratios of the algorithms analyzed in this chapter for the instances with a constant number of special vertices. By $\Delta$, $\Delta_{\beta}$ and “general” we denote the different restrictions on the cost function, and “hard” expresses that the problem is not approximable with a polynomial that is dependent on the size of the input.

<table>
<thead>
<tr>
<th>$c$ function</th>
<th>constant # of spec. vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta$</td>
<td>$P$</td>
</tr>
<tr>
<td>$\Delta_{\beta}, \beta &gt; 1$</td>
<td>$\beta^2 + \beta$</td>
</tr>
<tr>
<td>general</td>
<td>hard</td>
</tr>
</tbody>
</table>

As a final note, we would like to mention that $k$-SCP in undirected graphs is very similar to the problem of finding the cheapest vertex-disjoint paths between given $k$ pairs of vertices in a graph. (One just modifies the given graph by adding
zero-cost edges between special vertices. Any algorithm solving $k$-SCP on the modified graph computes the cheapest vertex-disjoint paths in the original graph.) The decision version of this $k$-vertex-disjoint-path problem for unweighted graphs is in $\mathcal{P}$ due to Robertson and Seymour [93]. The results was obtained during the almost 20 years of the development of the graph minor theory by the authors. In many cases, the results of the theory are obtained only for unweighted instances and it is open whether similar claims hold for weighted cases. If $k$-SCP were solved in polynomial time, the weighted version of Robertson and Seymour’s problem would be in $\mathcal{FPT}$. Such result would be probably important for the parameterized complexity theory community and could suggest approaches for transforming the results of Robertson and Seymour to the generalized weighted version. This all suggests that our problem is likely to be very involved.
Chapter 4. The Steiner Cycle Problem
Chapter 5

The Minimum Topic-Connected Overlay

Nowadays, many internet applications already support many-to-many communication based on sharing content: publishers publish information through a logical channel that is consumed by subscribed users. The situation is often modeled by publish/subscribe (pub/sub) systems. These systems can be classified into two categories. In the content-based pub/sub systems, the channels are associated with a collection of attributes and the messages are delivered to a subscriber only if their attributes match user-defined constraints. Each channel in the topic-based pub/sub systems is associated with a single topic. Here, the messages are distributed to the users via channels by his/her topic selection. There are numerous implementations of pub/sub systems, for details see [5, 13, 30, 31, 92, 99, 107].

In this chapter, we focus on the topic-based peer-to-peer pub/sub systems. In such a system, subscribers interested in a particular topic have to be connected without the use of intermediate agents (such as servers). Many aspects of such a system can be studied (see [36, 89]). One may be interested in achieving a small diameter of the overlay network to minimize the total time in which a message is distributed to all the subscribers. If an (average) degree of nodes in the network is minimized, the subscribers need to maintain (in average) a smaller number of connections. Here, we consider the minimization of the number of all the connections in the system, i.e., the minimum number of edges the overlay network needs to have to satisfy the demands of the subscribers. A small number of edges may decrease the maintenance requirements as the network needs only a small number of connections. All messages sent via one edge of the network may be aggregated to a single message and thus amortize the head count of otherwise small messages.

5.1 Preliminaries and Definitions

The most general problem that we study in this chapter is defined as follows.
Problem 5.1.1. Minimum topic-connected overlay (Min-TCO):

Input: A set of users $U$, a set of topics $T$ and a user interest function $F: U \rightarrow 2^T$.

Feasible solutions: Any set of edges $E \subseteq \{uv \mid u, v \in U \land u \neq v\}$ such that, for all $t \in T$, the graph on $E$ induced by the users interested in the topic $t$ is connected.

Costs: The number of edges in $E$, i.e., $|E|$.

Goal: Minimization.

For a user interest function $F$ we define the “inverse interest function” $F'(t) = \{v \in U \mid t \in F(v)\}$. Informally, it denotes the set of users that are interested in the topic $t$.

MIN-TCO was studied in different scenarios in [7, 36, 77, 78]. The hardness of the problem was studied in [36] where the inapproximability by a constant was proven and a logarithmic-factor approximation algorithm was presented.

Theorem 5.1.2 (Chockler et al., [36]). There exists a polynomial-time $O(\log |T|)$-approximation algorithm for MIN-TCO.

In [7], the authors study the Network Inference problem that is essentially identical with MIN-TCO. The authors here improve the inapproximability result of [36] to a logarithmic function.

Theorem 5.1.3 (Angluin et al., [7]). The approximation ratio of MIN-TCO problem is $\Omega(\log |U|)$, unless $P = NP$.

We also focus on the special cases of MIN-TCO (called MIN-$d$-TCO) where, for each topic, there are at most $d \geq 2$ users interested in it. We are not aware that the restriction of MIN-TCO to these special instances was studied before. We believe that MIN-$d$-TCO has wide practical applications such as cases where a publisher has a limited number of slots for users. For the sake of the integrity of this thesis, we formally define the MIN-$d$-TCO.

Problem 5.1.4. MIN-$d$-TCO is the following optimization problem:

Input: A set of users $U$, a set of topics $T$ and a user interest function $F: U \rightarrow 2^T$, such that there are at most $d$ users interested in a topic $t$, i.e., $|F'(t)| \leq d$, for all $t \in T$.

Feasible solutions: Any set of edges $E \subseteq \{uv \mid u, v \in U \land u \neq v\}$ such that, for all $t \in T$, the graph on $E$ induced by the users interested in the topic $t$ is connected.

Costs: The number of edges in $E$, i.e., $|E|$.

Goal: Minimization.
5.2. Hardness of Min-TCO and Min-$d$-TCO

In the inapproximability part (Section 5.2), we extend the method presented in [36] and design an AP-reduction from instances of the minimum hitting set problem (see Subsection 2.8.4 for details) to instances of Min-TCO proving the \textit{LOGAPX}-completeness of the problem. Hence, concerning approximability, the Min-TCO is equivalent with such a famous problem as the minimum set cover problem. As our reduction is not blowing up the number of users interested in a common topic, the reduction is also an AP-reduction of an \textit{APX}-complete subproblem of the minimum hitting set problem \textsf{Min-}$d$-HS (see Subsection 2.8.4 for details) to \textsf{Min-}$d$-TCO. Therefore, we immediately obtain \textit{APX}-hardness of \textsf{Min-}$d$-TCO.

In Section 5.3, we continue the study of \textsf{Min-}$d$-TCO. We design a one-to-one reduction of the instances of \textsf{Min-}$d$-TCO to \textsf{Min-}$d$-HS. As these special instances of the hitting set problem are approximable with a constant ratio, we obtain the first constant approximation algorithm for \textsf{Min-}$d$-TCO. Thus we conclude that \textsf{Min-}$d$-TCO is \textit{APX}-complete, even for $d = 3$.

As a consequence, our one-to-one reduction, together with already known \textit{FPT}-algorithm for \textsf{Min-}$d$-HS parametrized by the solution size, yields the first non-trivial parametrized algorithm for \textsf{Min-}$d$-TCO that is parametrized by the solution size. The speed up is by a factor of $2^k$ over the trivial brute force algorithm. We shortly discuss this result in Section 5.4.

5.2 Hardness of Min-TCO and Min-$d$-TCO

In this section, we discuss lower bounds for both Min-TCO and Min-$d$-TCO. We first present a simple observation.

\textbf{Theorem 5.2.1.} \textsf{Min-}$2$-TCO can be solved in linear time.

\textit{Proof.} In \textsf{Min-}$2$-TCO, there are at most two users interested in a common topic and these users must be connected together by a direct edge. Hence, the unique optimal solution of the problem contains all the edges between pairs of such users. \hfill \qed

Observe that, if $v, u \in U$ and $F(v) \subseteq F(u)$, we can remove user $v$ from the graph, solve the problem on the smaller instance and then connect $v$ to $u$ by a direct edge. This is true since $v$ has to be incident with at least one edge, and the edge $uv$ ensures that $v$ is connected to all the topics it is interested in. Furthermore, in any feasible solution, the other users connected to $v$ by an edge can be reconnected to user $u$ without disconnecting any topic and, hence, by the removal of the user $v$ we cannot eliminate all optimal solutions. Thus, in the following, we assume that, for any two users $u$ and $v$, the sets $F(u)$ and $F(v)$ are incomparable.

Moreover, we assume that, for all $t \in T$, we have $|F'(t)| > 2$. Otherwise the two users interested in a topic $t$ have to be connected by a direct edge in every overlay. Therefore, the topics that both users are interested in can be removed from the interests of the two users and the problem can be solved on the smaller instance.
In what follows, we extend methods from [36] and design an AP-reduction from \textsc{Min-}d-\textsc{HS} to \textsc{Min-(d+1)-TCO}, for \(d \geq 2\). Since \textsc{Min-}d-\textsc{HS} is \textsc{APX}\textsc{-}complete, the \textsc{APX}\textsc{-}hardness of \textsc{Min-}d-\textsc{TCO} immediately follows, for \(d \geq 3\). As this reduction holds for any \(d \geq 2\), we conclude the \textsc{LOGAPX}\textsc{-}hardness of \textsc{Min-TCO}.

**Theorem 5.2.2.** For arbitrary \(d \geq 2\), there exists an approximation preserving (AP) reduction from \textsc{Min-}d-\textsc{HS} to \textsc{Min-(d+1)-TCO}.

**Proof.** Let \(I_{\text{HS}} = (X, S)\) be an instance of \textsc{Min-}d-\textsc{HS} and let \(\varepsilon > 0\) be arbitrary. We omit the subscript in the functions \(\cost_{d-\text{HS}}\) and \(\cost_{(d+1)-\text{TCO}}\) as they are unambiguous. For the instance \(I_{\text{HS}}\), we create an instance \(I_{\text{TCO}} = (U, T, F)\) of \textsc{Min-}d-\textsc{TCO} with \(|X| + k\) users, where \(k := |X|^2 \cdot \lceil \frac{1 + \varepsilon}{\varepsilon} \rceil\), as follows (the function \(f\) in Definition 2.5.6).

\[
\begin{align*}
U &= X \cup \{p_i \mid p_i \notin X \land 1 \leq i \leq k\}, \\
T &= \{t^i_{S_j} \mid S_j \in S \land 1 \leq i \leq k\}, \\
F(x) &= \begin{cases} \\
\{t^i_{S_j} \mid x \in S_j \land S_j \in S \land 1 \leq i \leq k\} & \text{for } x \in X \\
\{t^i_{S_j} \mid S_j \in S\} & \text{for } x = p_i
\end{cases}
\end{align*}
\]

Observe that the instance contains \(k \cdot |S|\) topics and its size is polynomial in the size of \(I_{\text{HS}}\). The users interested in a topic \(t^i_{S_j}\) (\(S_j \in S\)) are exactly the elements that are members of set \(S_j\) in \textsc{Min-}d-\textsc{HS} plus a special user \(p_i\) (\(1 \leq i \leq k\)).

Let \(\text{Sol}_{\text{TCO}}\) be a feasible solution of \textsc{Min-}d-\textsc{TCO} on instance \(I_{\text{TCO}}\). We partition the solution into levels. Level \(i\) is a set \(L_i\) of the edges of \(\text{Sol}_{\text{TCO}}\) that are incident with the special user \(p_i\). In addition, we denote by \(L_0\) the set of edges of \(\text{Sol}_{\text{TCO}}\) that are not incident with any special user. Therefore, \(\text{Sol}_{\text{TCO}} = \bigcup_{i=0}^{k} L_i\) and \(L_i \cap L_j = \emptyset\) (\(0 \leq i < j \leq k\)).

We claim that, for any \(L_i\) (\(1 \leq i \leq k\)), the set of the non-special users incident with edges of \(L_i\) is a feasible solution of the instance \(I_{\text{HS}}\) of \textsc{Min-}d-\textsc{HS}. This is true since, if a set \(S_j \in S\) is not hit, none of the edges \(xp_i\) (\(x \in S_j\)) is in \(L_i\). But then the users interested in topic \(t^i_{S_j}\) are not interconnected as user \(p_i\) is disconnected.

Let \(j\) be chosen such that \(L_j\) is the smallest of all sets \(L_i\), for \(1 \leq i \leq k\). We construct \(\text{Sol}_{\text{HS}}\) by picking all the non-special users that are incident to some edge from \(L_j\) (the function \(g\) in Definition 2.5.6). Denote an optimal solution of \textsc{Min-}d-\textsc{HS} and \textsc{Min-}d-\textsc{TCO} for \(I_{\text{HS}}\) and \(I_{\text{TCO}}\) by \(\text{Opt}_{\text{HS}}\) and \(\text{Opt}_{\text{TCO}}\), respectively.

If we knew the solution \(\text{Opt}_{\text{HS}}\), we would be able to construct a feasible solution of \textsc{Min-}d-\textsc{TCO} on \(I_{\text{TCO}}\) as follows. First, we pick the edges \(xp_i\), \(x \in \text{Opt}_{\text{HS}}\), for all special users \(p_i\), and include them in the solution. This way, for any topic \(t \in F(p_i)\), we connect \(p_i\) to some element of \(X\) that is interested in \(t\), too. To have a feasible solution, we could miss some edges between some elements of \(X\). So, we pick all the edges between elements from \(X\). The feasible solution of \textsc{Min-}d-\textsc{TCO} on \(I_{\text{TCO}}\) that we obtain has roughly cost

\[
k \cdot \cost(\text{Opt}_{\text{HS}}) + |X|^2 \geq \cost(\text{Opt}_{\text{TCO}}).
\]
On the other hand, if we replace all levels $L_i$ ($1 \leq i \leq k$) by level $L_j$ in $Sol_{TCO}$, we still have a feasible solution of Min-TCO on $I_{TCO}$, with cost possibly smaller. Thus

$$k \cdot cost(Sol_{HS}) \leq cost(Sol_{TCO}).$$

We use these two inequalities to bound the cost of $Sol_{HS}$:

$$k \cdot cost(Sol_{HS}) \leq \frac{cost(Sol_{TCO})}{cost(Opt_{TCO})} \cdot (k \cdot cost(Opt_{HS}) + |X|^2)$$

and thus

$$\frac{cost(Sol_{HS})}{cost(Opt_{HS})} \leq \frac{cost(Sol_{TCO})}{cost(Opt_{TCO})} \cdot \left(1 + \frac{|X|^2}{k}\right).$$

If $cost(Sol_{TCO})/cost(Opt_{TCO}) \leq 1 + \varepsilon$ and $\alpha := 2$, then we have

$$\frac{cost(Sol_{HS})}{cost(Opt_{HS})} \leq (1 + \varepsilon) \cdot \left(1 + \frac{|X|^2}{k}\right) \leq (1 + \varepsilon) \cdot \left(1 + \frac{\varepsilon}{1 + \varepsilon}\right) = 1 + 2\varepsilon.$$

It is easy to see that the five conditions of Definition 2.5.6 are satisfied and thus we have an AP-reduction.

Corollary 5.2.3. For any $\delta > 0$ and polynomial-time $\alpha$-approximation algorithm for Min-$(d+1)$-TCO, there exists a polynomial-time $(\alpha + \delta)$-approximation algorithm for Min-$d$-HS.

Proof. The approximation algorithm for Min-$d$-HS uses Theorem 5.2.2 with

$$k := |X|^2 \cdot \left\lceil \frac{\alpha}{\delta} \right\rceil.$$

Our theorem also implies the following negative results on the approximability. One of them only holds if unique games conjecture is true. This conjecture is discussed in Subsection 2.8.4.

Corollary 5.2.4. Min-$d$-TCO

- is $NP$-hard to approximate in polynomial time within a factor of $(d - 2 - \varepsilon)$, for $d \geq 4$ and any $\varepsilon > 0$.
- admits no polynomial-time $(d - 1 - \varepsilon)$-approximation algorithm, for $d \geq 3$ and any $\varepsilon > 0$, unless the unique games conjecture fails.

Proof. Otherwise, the reduction described in the proof of Theorem 5.2.2 would imply an approximation algorithm for Min-$d$-HS with a ratio better than $d-1$ and $d$ respectively. This would directly contradict Theorem 2.8.31 and Theorem 2.8.32, respectively.

Corollary 5.2.5. Min-TCO cannot be approximated in polynomial time within a factor of $0.2267 \cdot \ln |S|$, unless $P = NP$. 
Chapter 5. The Minimum Topic-Connected Overlay

Proof. The corollary follows from Theorem 5.2.2 and Theorem 2.8.29.

The following corollary is an improvement of the already known results of Theorem 5.1.2 where an $O(\log |T|)$-approximation algorithm is presented, and of Theorem 5.1.3 where a lower bound of $\Omega(\log(|U|))$ on the approximability is shown. We closed the gap by designing a reduction that can reduce any problem from class $\text{LOGAPX}$ to Min-TCO preserving the approximation ratio up to a constant.

Corollary 5.2.6. Min-TCO is $\text{LOGAPX}$-complete.

Proof. Min-TCO is in the class $\text{LOGAPX}$ since it admits a logarithmic approximation algorithm as presented in Theorem 5.1.2. Our reduction from the proof of Theorem 5.2.2 is independent of $d$ and thus an AP-reduction from the $\text{LOGAPX}$-complete Min-HS (Theorem 2.8.28) to Min-TCO.

Corollary 5.2.7. For arbitrary $d \geq 3$, Min-$d$-TCO is $\text{APX}$-hard.

Proof. Our AP-reduction from the proof of Theorem 5.2.2 reduces, for $d \geq 2$, the $\text{APX}$-complete Min-$d$-HS (Theorem 2.8.30) to Min-$(d+1)$-TCO.

5.3 A Constant Approximation Algorithm for Min-$d$-TCO

In this section, we present a reduction from Min-$d$-TCO to Min-$O(d^2)$-HS which shows that there exists a constant approximation algorithm for Min-$d$-TCO as Min-$d$-HS is constantly approximable (Theorem 2.8.34). Moreover, the constant approximation algorithm classifies this problem to be a member of the class $\text{APX}$ and thus, since the $\text{APX}$-hardness was proven in Section 5.2, we conclude that Min-$d$-TCO is $\text{APX}$-complete.

Definition 5.3.1. Let $V = \{v_1, \ldots, v_n\}$ be a set of vertices and, for every partition $(A_i, B_i)$ of $V$, let $E_i = \{uv \mid u \in A_i \land v \in B_i\}$. Then we call the family $S = \{E_1, \ldots, E_m\}$ of all sets of edges between vertices of all the partitions of $V$ a characteristic family of edges on $V$. In other words, $S$ contains all sets of edges that form maximum bipartite graphs on $V$.

In the following lemma, we show the basic properties of a characteristic family of edges.

Lemma 5.3.2. Let $S = \{E_1, \ldots, E_m\}$ be a characteristic family of edges on the set $V$ of $n$ vertices. Then the following holds:

1. The size of the family $S$ is $m = 2^{n-1} - 1$.
2. For all $j$, $1 \leq j \leq m$,

$$|E_j| \leq \left\lceil \frac{n}{2} \right\rceil \cdot \left\lfloor \frac{n}{2} \right\rfloor.$$
3. For all $1 \leq i < j \leq m$, any two sets $E_i$ and $E_j$ differ in at least $n - 1$ elements.

4. Let $H \subseteq \{uv \mid u, v \in V \land u \neq v\}$. The graph $(V, H)$ is connected if and only if $H$ is a hitting set of

$$(\{uv \mid u, v \in V \land u \neq v\}, S).$$

5. The size of $S$ is minimal such that part 4 holds.

Proof. Observe that the complementary graph $(V, F_j)$, where $F_j = \{uv \mid u, v \in V \land u \neq v\} \setminus E_j$, contains two complete graphs – one on the vertices of $A_j$ and another on the vertices of $B_j$, and it is a maximal graph (in the number of edges) that is not connected. We use this observation to prove the last two parts of our lemma.

Part 1: We count the different partitions $(A_j, B_j)$ of the vertices $V$ as each such partition determines a different set $E_j$ of edges. There are $2^n$ ways how to distribute vertices from $V$ into partitions. We have to subtract 2 possibilities for the cases where one of $A_j$ or $B_j$ is empty. Each of the other possibilities is counted twice – once when the vertices are present in $A_j$ and once when they are present in $B_j$.

Part 2: Let the two sets of vertices $A_j$ and $B_j$ of a partition contain $k > 0$ and $n - k$ vertices. Then the size of $E_j$ is $k \cdot (n - k)$. This function reaches its maximum for $k = n/2$ and thus we can conclude that, for all $j, 1 \leq j \leq m$, we have $|E_j| \leq \lfloor n/2 \rfloor \cdot (n - \lfloor n/2 \rfloor) = \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$.

Part 3: Let us consider two different partitions $(A_i, B_i)$ and $(A_j, B_j)$ of the vertices $V$. The sets $A_i$ and $A_j$ must differ by at least one vertex. W.l.o.g., let the vertex $v \in A_i$ and $v \notin A_j$. Then, due to the transition of the vertex $v$ from $A_i$ to $B_j$, there are $|B_j|$ edges that are in $E_i$ but cannot be in $E_j$, and there are $|A_i| - 1$ edges that are not in $E_j$, but are in $E_j$. Thus, the overall difference in the number of elements between the sets $E_i$ and $E_j$ is at least $|A_i| + |B_j| - 1 = n - 1$.

Part 4: First, we prove the only-if case. Suppose that $H$ is a hitting set, but $(V, H)$ is not connected. Since $S$ contains complements of all maximal sets of edges that induce a disconnected graph, there exists $j (1 \leq j \leq m)$ such that $H \subseteq F_j$. But then, since $E_j$ is complementary to $F_j$, it follows that $E_j \cap H = \emptyset$. Thus, $H$ cannot be a hitting set as $E_j$ is not hit.

For the if case, suppose that $(V, H)$ is connected, but $H$ is not a hitting set of $(\{uv \mid u, v \in V \land u \neq v\}, S)$. Then there exists $j$ such that $E_j$ is not hit by $H$ and thus $H \subseteq F_j$. Yet in such a case, by our assumption, $(V, F_j)$ is not connected and thus $(V, H)$ cannot be connected as well.

Part 5: Let $S' = S \setminus E_j$, and let $(\{uv \mid u, v \in V \land u \neq v\}, S')$ be an instance of minimum hitting set problem. Then we claim that $F_j$ is a hitting set of $(\{uv \mid u, v \in V \land u \neq v\}, S')$. First, observe that $F_j \neq \emptyset$ since $E_j$ cannot contain all the edges. Moreover, there exists $e \in E_j$ ($E_j \in S'$) such that $e \notin E_j$. Then $e \in (\{uv \mid u, v \in V \land u \neq v\} \setminus E_j) = F_j$ and thus $F_j$ is a hitting set. However, by the definition of $F_j$, the graph $(V, F_j)$ cannot be connected and thus, the only-if case of part 4 does not hold.
Now we are ready to present a simple one-to-one reduction of \textsc{Min-}d-TCO to \textsc{Min-}\(O(d^2)\)-HS. The core concept is to construct a family of sets that has to be hit in \textsc{Min-}\(O(d^2)\)-HS as a union over all the topics of the characteristic family of edges on the users interested in the topic.

**Theorem 5.3.3.** There exists a one-to-one reduction of instances of \textsc{Min-}d-TCO to instance of \textsc{Min-}\(O(d^2)\)-HS.

**Proof.** Let \(I_{TCO} = (U, T, F)\) be an instance of \textsc{Min-}d-TCO. For each topic \(t \in T\) we define \(S_t\) to be the characteristic family of edges on users in \(F'(t)\). Note that Lemma 5.3.2 holds for each \(S_t\) with \(n := d\). We construct an \textsc{Min-}\(O(d^2)\)-HS instance \(I_{HS} = (X, S)\) as follows:

\[
X = \{uv \mid u, v \in U \land u \neq v\}
\]
\[
S = \bigcup_{t \in T} S_t.
\]

The family contains \((|U|)^2\) elements and at most \(|T| \cdot (2^{d-1} - 1)\) sets in \(S\) and thus has a size polynomial in \(|I_{TCO}|\). Obviously, the construction of \(I_{HS}\) takes time polynomial in \(|I_{TCO}|\), too. We now show that a feasible solution of \(I_{TCO}\) corresponds to a feasible solution of \(I_{HS}\) and vice versa.

First, consider a feasible solution \(Sol_{HS}\) of \(I_{HS}\) and a topic \(t \in T\). Due to our construction, the family \(S\) contains the characteristic family \(S_t\) on users \(F'(t)\). Therefore, by Lemma 5.3.2 part 4 and the fact that \(Sol_{HS}\) is a hitting set, we know that the graph induced by the edges in \(Sol_{HS}\) on users \(F'(t)\) is connected.

Now, consider a feasible solution \(Sol_{TCO}\) of \(I_{TCO}\). By the following argument, we can easily see that \(Sol_{TCO}\) hits all the sets in \(S\). Let \(P \in S\) be a set that is not hit by \(Sol_{TCO}\). Then there exists \(t\) such that \(P \in S_t\) and thus a set of the characteristic family was not hit and \(Sol_{TCO}\) is not a hitting set of \(S_t\). Yet in such a case, considering Lemma 5.3.2 part 4, the subgraph induced on users \(F'(t)\) by edges from \(Sol_{TCO}\) cannot be connected and that is in contradiction with the definition of \textsc{Min-}d-TCO.

**Theorem 5.3.4.** For arbitrary \(d \geq 3\), there exists a polynomial-time approximation algorithm for \textsc{Min-}d-TCO with ratio

\[
\left\lceil \frac{d}{2} \right\rceil \cdot \left\lfloor \frac{d}{2} \right\rfloor.
\]

**Proof.** We employ the reduction from Theorem 5.3.3 together with the well-known \(d\)-approximation algorithm for \textsc{Min-}d-HS (see Theorem 2.8.35). Since the size of each set in \(S\) is at most \([d/2] \cdot [d/2]\) (Lemma 5.3.2 part 2), by application of this approximation algorithm on the \textsc{Min-}\(O(d^2)\)-HS instance \((X, S)\), we obtain a \([d/2] \cdot [d/2]\) approximate solution for our \textsc{Min-}d-TCO instance.

Note that our reduction is tight in the size of \(S\) as it is minimal (Lemma 5.3.2 part 5), thus to achieve an improvement in the approximation algorithm, a different method has to be developed.
Corollary 5.3.5. Min-3-TCO inherits the approximation hardness of the minimum vertex cover problem.

Corollary 5.3.6. For arbitrary \( d \geq 3 \), Min-\( d \)-TCO is APX-complete.

Proof. Due to our reduction, the problem belongs to the class APX, the APX-hardness is proven by Corollary 5.2.7.

\[ \square \]

5.4 A Parameterized Algorithm for Min-\( d \)-TCO

We shortly summarize the consequences of our reduction from Theorem 5.3.3 leading to a nontrivial parameterized algorithm for Min-\( d \)-TCO. (For a basic introduction to parameterized complexity theory, see Section 2.7.)

Problem 5.4.1. Min-\( d \)-TCO-sol(\( k \)) is a parameterization of Min-\( d \)-TCO by the output size.

Input: An instance of Min-\( d \)-TCO and a parameter \( k \).

Decision: Is there a feasible solution of the Min-\( d \)-TCO instance of size at most \( k \)?

Consider an instance of Min-\( d \)-TCO with \( n \) vertices. A straightforward parameterized algorithm for Min-\( d \)-TCO-sol(\( k \)) has to consider all sets of at most \( k \) edges out of possible \( \binom{n}{2} \) edges of the instance. This number of sets can be roughly estimated as

\[
\sum_{i=0}^{k} \binom{n^2}{i} \leq k \cdot \binom{n^2}{k} \leq k \cdot n^{2k}.
\]

Hence, the time complexity of such an approach is \( O^*(n^{2k}) \).

However, if we apply first our transformation from Theorem 5.3.3 and then the FPT-algorithm from Theorem 2.8.38 of Subsection 2.8.4, we obtain a better algorithm for Min-\( d \)-TCO-sol(\( k \)).

Theorem 5.4.2. Min-\( d \)-TCO-sol(\( k \)) with \( n \) users can be solved in time \( O(\alpha^k + n^2) \) with

\[
\alpha = \frac{1}{2} \left( \left\lfloor \frac{n}{2} \right\rfloor \cdot \left\lfloor \frac{n}{2} \right\rfloor - 1 \right) + \frac{1}{2} \left( \left\lfloor \frac{n}{2} \right\rfloor \cdot \left\lfloor \frac{n}{2} \right\rfloor - 1 \right) \cdot \sqrt{1 + \frac{4}{(\left\lfloor n/2 \right\rfloor \cdot \left\lfloor n/2 \right\rfloor - 1)^2}}
\]

which is approximately

\[
\alpha = \left\lfloor \frac{n}{2} \right\rfloor \cdot \left\lfloor \frac{n}{2} \right\rfloor - 1 + O(n^{-2}).
\]
Proof. Let \((x, k)\) be an instance of Min-\(d\)-TCO-sol\((k)\). We apply our reduction from Theorem 5.3.3 to transform \(x\) to an instance \(y\) of Min-\(O(d^2)\)-HS with dimension \(d := \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil\). Our transformation matches feasible solutions of both instances in a one-to-one manner, hence, \((x, k)\) is a yes-instance of Min-\(d\)-TCO-sol\((k)\) if and only if \((y, k)\) is a yes-instance of Min-\(d\)-HS-sol\((k)\).

Hence, the FPT-algorithm of Niedermeier and Rossmanith for Min-\(d\)-HS problem (see Theorem 2.8.38) can be applied. The algorithm yields the claimed time complexity.

If we compute a rough estimation of the time complexity of this algorithm, we have

\[
O(\alpha^k + n^2) = O^*(4^{-k}n^{2k}).
\]

Hence, we can conclude that our algorithm is exponentially better than a trivial algorithm that exhaustively searches through all possibilities.

5.5 Summary

In this chapter, we have closed the gap in the approximation hardness of Min-TCO by showing its LOGAPX-completeness. We studied a subproblem of Min-TCO called Min-\(d\)-TCO, we designed the first constant approximation algorithm and we showed that Min-\(d\)-TCO is APX-complete. From our reductions, we concluded the existence of the first non-trivial parametrized algorithm for Min-\(d\)-TCO. We also gave various inapproximability results for Min-TCO and Min-\(d\)-TCO that are corollaries of our theorems and the already known results.

The open questions for a further study of Min-TCO can be divided into several groups. First of all, one can define a similar problem to Min-TCO that minimizes other parameters of the instance. Secondly, one can study the behavior of Min-TCO depending on some other parameters such as the instances where the number of connections of a user is bounded by a constant. Finally, there are still remaining open issues in parameterized complexity: Can we use our reductions to design a fixed-parameter algorithm for Min-\(d\)-TCO? Is Min-TCO \(W[2]\)-hard as the minimum hitting set problem? Can we apply the properties of characteristic sets to obtain a kernelization and a non-trivial exact algorithm for Min-\(d\)-TCO?
Chapter 6

Splitting Problems

In this chapter, we consider a new operation, the so-called splitting of variables that we apply on several problems. In Section 6.1, we discuss the motivation behind the operation. Our results are presented in Sections 6.2 and 6.3, where the satisfiability of a Boolean formula and the Hamiltonicity of a graph with respect to the operation of splitting are investigated. In the last section we summarize our experiences and we discuss possibilities for further research.

We would like to note that the presented results are from a still undergoing research. Hence, they are far from being complete or from covering all the interesting questions that the reader might have.

6.1 Motivation

Nowadays, an important problem in biology is to reconstruct the evolutionary relations between living beings. In general, an evolutionary relation is often built based on either species, their properties, or individual genes. The most commonly used approach to reconstruct such data is to build a so-called phylogenetic tree. The leaves of this tree correspond to the species or other considered data and the inner nodes represent the hypothetical ancestors that are extinct. For more information about different phylogenetic trees and various algorithms for their computation see [20, 100].

The reconstruction of the phylogenetic trees from samples is a computational problem on its own. However, another issue is the quality of the sampled data, that can be false such that a valid phylogenetic tree might not exist. If the data is false, one straightforward and very natural assumption is that two distinct species were recognized as the same one, hence their ancestors should be distinct which can lead to a constructable phylogenetic tree. Moreover, assuming that the data is not completely corrupted, one should ask for the minimum number of corrections to be made so that the data becomes consistent, which corresponds to the minimum number of incorrectly recognized species. These assumptions are not artificial and should closely model the real expectations.
The study in this chapter is inspired by the corrections of corrupted phylogenetic data as discussed above. The operation of splitting, that we consider, aims to generalize the data consistency correction for other problems. The general motivation is alike, the splitting operation models a single correction on the data and the overall goal is the minimum possible alteration of the data.

### 6.2 Satisfiability of Formulas

There are many natural problems that can be modeled by Boolean formulas. The set of feasible solutions of such a problem often consists of assignments for the variables of the corresponding formula such that the formula is satisfied by the assignment and, possibly, some extra conditions hold.

However, once the formula is not satisfiable, something in the process of generating the formula went wrong and the formula does not match the real problem instance. Similarly as in the example with phylogenetic trees from the previous section, we anticipate that the formula is mainly correct and there were just minor problems that caused the situation.

In this section, we start the study of such corrupted formulas and we discuss possibilities how to extract from them the information that is correct. For more details related to the satisfiability problem, see Subsection 2.8.3.

First, we consider the problem of variable splitting of a formula such that it becomes satisfiable. As we will see from the further discussion and the following definition, the splitting of a variable in a Boolean formula has similar effects as erasing clauses containing corrupted variables.

Next, we consider a dual problem to splitting where even though the given formula is not satisfiable, we are interested in the maximum number of variables that we can assign at once without causing a straight contradiction in the formula.

**Definition 6.2.1.** Let $F$ be a Boolean formula over $X = \{x_1, \ldots, x_n\}$ and let $y$ and $z$ be two new variables. A variable $x_i$ is split in $F$ if each occurrence of $x_i$ in $F$ is replaced by either $y$ or $z$ and each occurrence of $\overline{x_i}$ is replaced by either $\overline{y}$ or $\overline{z}$. Such operation of replacement of $x_i$ and $\overline{x_i}$ is called a splitting of $x_i$.

Note that, if multiple variables has to be split, the operation of splitting is independent on the order of the split variables. We define the splitting problem on SAT in general, but we will be interested mainly in 2-SAT and 3-SAT.

**Problem 6.2.2.** **Min-Split-$k$-SAT** is the following optimization problem:

**Input:** A Boolean formula $F$ over $X = \{x_1, \ldots, x_n\}$ in $k$-CNF.

**Feasible solutions:** A set $S \subseteq X$ such that there exists a splitting of variables of $S$ in $F$ such that the split formula is satisfiable.

**Costs:** One plus the size of $S$, i.e., $1 + |S|$.

**Goal:** Minimization.
Perhaps, one would expect that the cost of a Min-Split-$k$-SAT solution should be just $|S|$ not increased by a one. The addition of one to the cost is just a technical detail that is needed if one considers the approximability of the Min-Split-$k$-SAT. If the cost of an instance would be defined just as $|S|$, the cost of an optimal solution of a satisfiable instance would be zero. However, for Min-Split-$k$-SAT ($k \geq 3$), we are unable to test this fact in polynomial time. Due to our (and the standard) definition of the approximation ratio, any $f(n)$-approximation algorithm would have to deliver a solution of the cost $0 \cdot f(n)$ which would have to be also an optimal solution. This is certainly not what we aim for in our study of the problem. Our current definition of the cost function avoids such artificial situations and allows us to study the approximability of the problem.

Observe that, if our goal is to split the minimum number of variables such that the entire new formula becomes satisfiable, the optimal way of splitting a variable $x$ is to replace its positive occurrences by a new variable $y$ and the negative occurrences by the other new variable $\bar{z}$. Hence, in such a newly created formula the variable $y$ can be set to be 1 and $z$ to be zero, so that all the clauses that contained $x$ and now contain either $y$ or $\bar{z}$ become true and do not have to be considered in the formula anymore. Note that literals $\bar{y}$ and $z$ are not present in the new formula and hence the operation of split could be viewed as erasing of all clauses containing the variable $x$ which is accounted into the cost of the solution by one. Setting a variable to a Boolean value so that some clauses are satisfied instead of erasing all the clauses containing the variable by splitting it has a contribution of zero to the overall cost of the solution. Intuitively, the splitting of a variable removes all the clauses that contain a “faulty” variable which should not be present in the formula. We formulate this fact into an observation.

**Observation 6.2.3.** A splitting of a variable in an instance of Min-Split-$k$-SAT is equivalent to erasing the variable from the instance, together with all the clauses in which it is present.

Note that deciding whether a given instance of Min-Split-3-SAT admits a solution without splits or not is identical with 3-SAT which is $\mathcal{NP}$-complete (Corollary 2.8.18). Hence, the general Min-Split-$k$-SAT, which can be transform by Lemma 2.8.17 into Min-Split-3-SAT, is a hard problem. In what follows, we present our hardness results for Min-Split-$k$-SAT for $k \leq 2$ and we shortly discuss the approximability of the problem.

**Observation 6.2.4.** Min-Split-1-SAT can be solved in polynomial time.

*Proof.* An instance of Min-Split-1-SAT is a conjunction of literals. Hence, if for a variable $x$ both literals $x$ and $\bar{x}$ are present in the given formula, both literals cannot be satisfied by an assignment of $x$ and we have to split $x$. Hence, each variable that is present both as positive and as negative in a formula have to be split in an optimal solution. □

**Theorem 6.2.5.** There exists an AP-reduction from the minimum vertex cover problem to Min-Split-2-SAT.
Proof. Let $I_{VC} = (V, E)$ be an instance of the minimum vertex cover problem (MIN-VC). In the corresponding instance $I_{MS2S} = (X, F)$ of the MIN-SPLIT-2-SAT we identify each vertex with a single variable, i.e.,

$$X = \{x_v \mid v \in V\}.$$  

For each edge of $(u, v) \in E$, we add to $F$ all four possible clauses on variables $x_u$ and $x_v$, i.e.,

$$F = \bigwedge_{(u, v) \in E} \left( (x_u \lor x_v) \land (\overline{x_u} \lor x_v) \land (x_u \lor \overline{x_v}) \land (\overline{x_u} \lor \overline{x_v}) \right).$$

Observe that, since for each edge $(u, v)$ all four clauses $(x_u \lor x_v)$, $(\overline{x_u} \lor x_v)$, $(x_u \lor \overline{x_v})$ and $(\overline{x_u} \lor \overline{x_v})$ are in $F$, there is no assignment to either $x_u$ or $x_v$ that would satisfy all of them. Hence, $F$ is not satisfiable. However, once we split one of the variables $x_u$ or $x_v$, by the above discussion, all four clauses become satisfiable. Since such four clauses causing unsatisfiability of $F$ are present in $F$ for each edge of $I_{VC}$, at least one variable corresponding to an endpoint of the edge has to be split. Once a variable $x$ is split, there exists an assignment to the two newly used variables so that each clause where $x$ occurred is satisfied. Hence, a feasible solution of MIN-SPLIT-2-SAT on $I_{MS2S}$ indeed corresponds to a feasible solution of MIN-VC on $I_{VC}$.

On the other hand, any feasible solution $Sol_{VC}$ of $I_{VC}$ can be transformed in one-to-one correspondence to a feasible solution $Sol_{MS2S}$ of $I_{MS2S}$: the variables corresponding to the vertex cover are those that are split. As all edges of $E$ are covered in $Sol_{VC}$, the four clauses corresponding to each edge are satisfied due to the split variable.

Note that the same transformations can be performed also for the optimal solutions. Hence, for all corresponding $Sol_{VC}$ and $Sol_{MS2S}$ we have

$$cost(Sol_{VC}) = cost(Sol_{MS2S}),$$

which concludes the proof. \qed

**Corollary 6.2.6.** MIN-SPLIT-2-SAT is APX-hard.

**Corollary 6.2.7.** For an arbitrary $\varepsilon > 0$, MIN-SPLIT-2-SAT cannot be approximated by a polynomial-time algorithm within a factor of $2 - \varepsilon$, unless the unique games conjecture fails.

Proof. Otherwise, our reduction from Theorem 6.2.5 would allow us to approximate MIN-VC within a factor of $2 - \varepsilon$ which would be in contradiction with Theorem 2.8.32. \qed

**Corollary 6.2.8.** It is NP-hard to approximate MIN-SPLIT-2-SAT with a factor of 1.3606 by a polynomial-time algorithm.
6.2. Satisfiability of Formulas

Proof. Otherwise, our reduction from Theorem 6.2.5 would allow us to approximate Min-VC within the factor 1.3606 which would be in contradiction with Theorem 2.8.33.

From the approximability perspective, Min-Split-2-SAT is trivially approximable by a factor of \((n + 1)/2\): First we check by a polynomial-time algorithm (Theorem 2.8.19) whether the given instance is satisfiable (and hence the cost of the solution is one). If not, it is clear that we have to split at least one variable and hence the cost of an optimal solution is at least 2. But, if we split all the variables, we obtain a feasible solution of cost \(n + 1\).

We are also aware of an \(O(n \log n)\)-approximation algorithm for the Min-Split-2-SAT which we do not discuss here in detail [85].

By the same argument as for Min-Split-2-SAT, the Min-Split-3-SAT is trivially approximable by a factor of \(n + 1\) if all the variables are split. The lower bound for the approximability of Min-Split-3-SAT is very high, which is not so surprising. In [79], it was shown that, for an arbitrary \(\varepsilon > 0\), the Min-Split-3-SAT with \(n\) variables is not approximable within a factor of \(n^{1-\varepsilon}\).

In the remainder of this section, we study a dual problem to Min-Split-\(k\)-SAT. Instead of minimizing the number of variables that are “faulty” and should be removed from a formula, we focus on the satisfiable part of the formula. Our goal is to assign as many variables at once as possible such that they do not evaluate any clause to a contradiction, but the remainder of the formula (with unassigned variables) persists unsatisfiable.

Problem 6.2.9. Max-Assign-\(k\)-SAT is the following optimization problem:

Input: A Boolean formula \(F\) over \(X = \{x_1, \ldots, x_n\}\) in \(k\)-CNF.

Feasible solutions: A set \(S \subseteq X\) such that there exists an assignment \(\alpha\) of \(S\) that does not evaluate any clause of \(F\) to false, i.e., for each clause \((l_1 \lor \cdots \lor l_k)\) of \(F\), either at least one literal of \(l_i\) is true by \(\alpha\) or at least one literal \(l_i\) is unassigned by \(\alpha\) (\(1 \leq i \leq k\)).

Costs: The size of \(S\), i.e., \(|S|\).

Goal: Maximization.

Note that the artificial problem that we experienced in the Min-Split-\(k\)-SAT and due to which the plus one in the cost function appeared there does not occur here. The reason is that no instance of the Max-Assign-2-SAT which is not an instance of the Max-Assign-1-SAT can cost zero. This follows from the fact, that there is always at least a single variable that can be assigned without causing a contradiction in any clause – it is sufficient to pick any of the two variables in a 2-CNF clause of the Max-Assign-2-SAT and assign it an arbitrary value.

Now, we present our results concerning the approximability of Max-Assign-\(k\)-SAT.
Theorem 6.2.10. Max-Assign-1-SAT can be solved in polynomial time.

Proof. An instance of Max-Assign-1-SAT is a conjunction of literals. Hence, only if a given formula contains both variable $x$ and its negation, $x$ cannot be assigned any Boolean value and has to stay unassigned. □

Theorem 6.2.11. There exists an AP-reduction from the maximum independent set problem on undirected hypergraphs to the Max-Assign-2-SAT.

Proof. Let a $d$-dimensional undirected hypergraph $I_{IS} = (V, E)$ be an instance of the maximum independent set problem (Max-IS). We reduce it to an instance $I_{MA2S} = (X, F)$ of the Max-Assign-2-SAT by a reduction that generalizes the one from Theorem 6.2.5.

Each vertex of $V$ corresponds to a single variable, i.e.,

$$X = \{x_v \mid v \in V\}.$$ 

For each hyperedge $(u_1, \ldots, u_d) \in E$ we add to $F$ all possible 2-CNF clauses on literals of negated and unnegated variables that correspond to different vertices in the edge. More precisely,

$$F = \bigwedge_{(u_1, \ldots, u_d) \in E} \left( \bigwedge_{1 \leq i < j \leq d} (x_{u_i} \lor x_{u_j}) \land (\overline{x_{u_i}} \lor \overline{x_{u_j}}) \land (x_{u_i} \lor \overline{x_{u_j}}) \land (\overline{x_{u_i}} \lor x_{u_j}) \right).$$

Furthermore, we refer to the 2-CNF clauses that correspond to a single hyperedge $e$ (the part without the outer wedge of the above expression) as to a clique of the edge $e$.

Note that, for each edge of $E$, we add to $F$ at most $4^{\binom{d}{2}} = O(d^2) = O(n^2)$ clauses. Hence, our reduction is indeed a polynomial-time reduction, that moreover does not depend on the dimension of the hypergraph.

The crucial property we use in this proof is that we cannot assign any values to two or more variables from a single clique. This is true since each clique contains four clauses $(x \lor y)$, $(x \lor \overline{y})$, $(\overline{x} \lor y)$ and $(\overline{x} \lor \overline{y})$ and hence assigning any values to both $x$ and $y$ leads to a contradiction in one of these four clauses.

First, consider a feasible solution $Sol_{IS}$ of the Max-IS of $I_{IS}$. We set all the variables of $X$ corresponding to the elements of $Sol_{IS}$ to be true. Since $Sol_{IS}$ is an independent set and $F$ is the union of the cliques for each edge in $I_{IS}$, there is at most one variable $x$ in each clique that is assigned. This variable cannot cause a contradiction as each clause that contains either of the literals $x$ or $\overline{x}$ contains also a second unassigned literal. Hence, each feasible solution $Sol_{IS}$ of Max-IS can be matched to a feasible solution $Sol_{MA2S}$ of Max-Assign-2-SAT.

Consider a feasible solution $Sol_{MA2S} \subseteq X$ of Max-Assign-2-SAT with an assignment $\alpha$, i.e., if $\alpha$ is assigned to the variables of $Sol_{MA2S}$ such that there is no contradiction in any clause of the formula $F$. Let $Sol_{IS}$ be the set of vertices that correspond to the variables of $Sol_{MA2S}$. Observe that, from each clique, at most one variable can be assigned without creating a contradiction. Since each clique
corresponds to an edge in \( E \), each edge is covered by at most one vertex from \( \text{Sol}_{IS} \) and hence, indeed, \( \text{Sol}_{IS} \) is an independent set of \( I_{IS} \). We have just proved that any feasible solution \( \text{Sol}_{MA2S} \) of Max-Assign-2-SAT can be matched to a feasible solution \( \text{Sol}_{IS} \) of Max-IS.

Note that the transformation between feasible solutions also holds for the optimal solutions. Hence, for all corresponding \( \text{Sol}_{IS} \) and \( \text{Sol}_{MA2S} \), we have

\[
\text{cost}(\text{Sol}_{IS}) = \text{cost}(\text{Sol}_{MA2S}),
\]

which concludes the proof.

By the following corollary showing that already Max-Assign-2-SAT is a very hard problem, we conclude the study of the hardness of Max-Assign-\( k \)-SAT as this generalized version is going to be extremely hard.

**Corollary 6.2.12.** Max-Assign-2-SAT with \( n \) variables cannot be approximated by a polynomial-time algorithm within a factor of \( \frac{n}{2^{O(\sqrt{\log n})}} \), unless \( P = NP \).

**Proof.** Otherwise, by our reduction, we would be able to approximate the maximum independent set problem on hypergraph by a better factor than \( \frac{n}{2^{O(\sqrt{\log n})}} \) which would be in contradiction with Theorem 2.8.43.

---

### 6.3 Hamiltonicity of Graphs

In this section, we consider a splitting operation for another well-known decision problem – the problem of deciding whether a given graph contains a Hamiltonian cycle.

The motivation is similar as in the previous section. We are given a graph instance of a problem that should be Hamiltonian, but is not. The experimental data from which the instance was built was corrupted. We need to repair the data and the graph, so it becomes Hamiltonian in order to continue our further computation that is valid only on Hamiltonian graphs. Similar situations might arise, for instance, in bioinformatics during the genome sequencing (see Section 8.1 from [20] where a directed version of TSP is used to model the shortest common superstring problem).

There are two simple and natural problems that could cause such a situation. Either several relations were not transferred into the instance and hence some edges are not present in the graph. Or several different vertices were identified into a single vertex. We consider the latter case where the splitting operation on vertices of the graph is seen as a method of repairing the Hamiltonicity property.

There are two general ways how such a reparation of a vertex can be performed with respect to its incident edges.

- Either the vertex is replaced by two new vertices which have the identical incident edges as the replaced vertex. Hence, this process just creates another copy of the vertex in the graph (see Figure 6.1). We refer to this approach as a **cloning of a vertex**.
• Or the vertex is replaced by two new vertices and its incident edges are split in some way between the two new vertices (see Figure 6.2). We refer to this approach as a splitting of a vertex.

Similarly as in the previous sections, we anticipate that our instance is essentially correct and the most likely way to obtain the original instance is to use the operation a minimum number of times.

First, we consider the operation of cloning on a graph \( G \). Observe that any tour \( T \) on \( G \) such that \( V(G) = V(T) \) can be transformed into a Hamiltonian cycle in \( G \) with cloned vertices that correspond to the multiply visited vertices in \( T \) as follows. Traverse the graph \( G \) in the tour \( T \) and whenever a vertex should be visited a second time, clone it and continue the traversal in this cloned vertex. Let \( c \) be the number of vertices that were cloned in the graph \( G \) during the traversal. Then the length of \( T \) is precisely \( |V| + c \). Hence the problem of minimizing the number of cloned vertices in \( G \) such that it becomes Hamiltonian is identical to the problem of minimizing the length of the tour \( T \) that visits every vertex in \( G \) at least once. The latter problem is known as a graph TSP (GRAPH-TSP) in the literature and it is an extensively studied problem. Formally, GRAPH-TSP is defined on an unweighted graph and asks to build a shortest tour that contains each vertex at least once. Equivalently, one can ask to compute the TSP in a complete graph with edge costs corresponding to the shortest paths in the unweighted initial graph. In [61], the GRAPH-TSP is claimed to be \( \text{APX} \)-hard due to a modification of a proof from [90]. For a long time, it was known to be approximable by a modified Christofides’ algorithm (see Subsection 2.8.1 for the description of the Christofides’ algorithm) with a factor of 1.5. Recently, the barrier of the ratio 1.5 was broken by two teams of researchers, independently. Gharan et al. [58] proved that the GRAPH-TSP is approximable by a factor of \( 1.5 - \varepsilon \), where \( \varepsilon \) is in the order of \( 10^{-12} \), Mömke and Svensson [86] showed a 1.461-approximation algorithm. The latter paper also shortly outlines the state of the art for the GRAPH-TSP.

We are not aware that the problem of splitting of vertices, as it is going to be defined, was considered before in another context. Hence, in the remainder of this section, we aim to present the first observations related to this problem.
Definition 6.3.1. Let $G = (V, E)$ be an undirected graph, let $u \in V$, let $v$ and $w$ be two new vertices and let $(P, Q)$ be a partition of neighboring vertices of $u$, i.e., $P \cup Q = N(u)$ and $P \cap Q = \emptyset$. A vertex $u$ is split in $G$ if it is replaced by two new vertices $v$ and $w$ and all the incident edges of $u$ are replaced by bidirectional edges $vp$ and $wq$, where $p \in P$ and $q \in Q$ (see Figure 6.2). We also refer to the operation as being a splitting of the vertex $u$ in the graph $G$ and the graph on which the splitting was applied we call a split graph.

![Figure 6.2: The splitting operation at vertex $u$. The neighbors of $u$ are partitioned into two sets $P$ and $Q$, vertex $u$ is replaced by two new vertices $v$ and $w$ and the vertices of the two sets are connected to the two new vertices.](image)

Since edges in a graph are not duplicated by the splitting of vertices, graphs that contain a bridge (i.e., an edge whose removal disconnects the graph) cannot be split to become Hamiltonian. More generally, graphs with the following property can be modified by the vertex splitting to admit a Hamiltonian cycle.

Property 6.3.2. An undirected connected graph $G$ contains a tour that visits every vertex at least once and uses every edge at most once.

For instance, if a graph contains an Eulerian tour as a subgraph, it satisfies Property 6.3.2.

Note that Property 6.3.2 is both a necessary and sufficient condition for a graph that can be modified by the vertex splitting to become Hamiltonian – any Hamiltonian cycle transferred from the split graph into the original graph forms a tour with Property 6.3.2. On the other hand, one can traverse a tour $T$ that visits each vertex at least once and each edge at most once and build a split graph containing a Hamiltonian cycle as follows: Traverse $T$, whenever a vertex should be visited for the second time, split it and continue the traversal in the split vertex.

Now we are ready to formally define our splitting problem.

Problem 6.3.3. **MIN-SPLIT-HC** is the following optimization problem:

**Input:** A graph $G = (V, E)$ satisfying Property 6.3.2.

**Feasible solutions:** A sequence $S$ of splits of vertices from $V$ such that the graph that is created by sequentially applying all splits of $S$ contains a Hamiltonian cycle.
Costs: One plus the number of splits of vertices, i.e., 1 + |S|.

Goal: Minimization.

We cannot define the cost of a solution of Min-Split-HC as the number of splits |S| only, as similar artificial problems to those discussed in Section 6.2 would arise. Observe that deciding whether an Eulerian graph contains a Hamiltonian cycle is an \( \mathcal{NP} \)-complete problem. As all Eulerian graphs satisfy our property, the Min-Split-HC is well defined for such graphs. Hence, if the cost in the Min-Split-HC would be defined as |S|, any algorithm with finite approximation ratio would be able to decide whether a given Eulerian graph \( G \) contains a Hamiltonian cycle (each approximation algorithm would have to deliver a solution of cost 0 if and only if a Hamiltonian cycle is present in \( G \)).

The crucial difference that makes Min-Split-HC very different from a corresponding problem for cloning (i.e., graph-TSP) is that each edge of a graph can be used at most once in a feasible solution of the Min-Split-HC, whereas the other problem allows to use a single edge more times. Hence, the following result is very different to what we know about the graph-TSP.

**Theorem 6.3.4.** For any \( \varepsilon > 0 \), there is no polynomial-time \( (n^{1-\varepsilon}) \)-approximation algorithm for Min-Split-HC for a graph with \( n \) vertices, unless \( \mathcal{P} = \mathcal{NP} \).

**Proof.** For a contradiction, assume that, for a constant \( \varepsilon \), there is a polynomial-time \( (n^{1-\varepsilon}) \)-approximation algorithm \( A \) for the Min-Split-HC, let \( G \) be a graph satisfying Property 6.3.2 and let \( \alpha := (1 - \varepsilon)/\varepsilon \).

We present a Turing reduction in which our algorithm \( A \) is employed at most \( n - 1 \) times and that decides whether \( G \) contains a Hamiltonian cycle or not. Since deciding whether an Eulerian graph (which has Property 6.3.2) contains a Hamiltonian cycle is a \( \mathcal{NP} \)-complete problem, our algorithm would decide this problem in polynomial time which leads to a contradiction.

Let us fix a single vertex \( v_0 \in V(G) \). We perform the following transformation sequentially for each edge \( v_0v \in E(G) \) and, from all (at most \( n - 1 \)) solutions we obtain, we pick the best one, i.e., the one with the smallest cost.

Let \( R_v \) be a graph \( G \) with removed edge \( v_0v \), i.e., \( V(R_v) = V(G) \) and \( E(R_v) = E(G) \setminus \{v_0v\} \). We construct a new graph \( G'_v \) from \( n^\alpha \) copies \( C_1, \ldots, C_{n^\alpha} \) of \( R_v \). The copies are connected together by edges between the vertex \( v_0 \) of \( C_i \) and the vertex \( v \) in \( C_{i+1} \), for all \( 1 \leq i \leq n^\alpha - 1 \), and also by an edge between \( v_0 \) of \( C_{n^\alpha} \) and \( v \) of \( C_0 \). The graph \( G'_v \) is depicted in Figure 6.3. Note that the size of the new graph is polynomial in the number of vertices in \( G \).

By the following simple argument, it follows that \( G'_v \) also satisfies Property 6.3.2 and hence deciding whether \( G'_v \) contains a Hamiltonian cycle is still \( \mathcal{NP} \)-complete. Let \( T \) be a tour proving that Property 6.3.2 is satisfied in \( G \). If \( v_0v \in E(T) \), the tour in each copy \( C_i \) contains the walk between vertices \( v_0 \) and \( v \) and the outgoing edges to the previous and the next copy. Otherwise, the tour in each copy \( C_i \) is identical to \( T \), and in addition it contains the edge \( v_0v \) and the outgoing edges to the previous and the next copy.
We first show how the Hamiltonicity property is transformed in between the graphs.

The graph $G$ contains a Hamiltonian cycle if and only if there exists a vertex $v \in V$ (and an edge $v_0v \in E$) such that $G_v$ contains a Hamiltonian cycle.

Assume that $G$ contains a Hamiltonian cycle $H$. One of the incident edges $v_0x \in E$ has to be present in $H$. The graph $G_x$ contains a Hamiltonian cycle that is constructed as follows. Start in $x$ of $C_1$, traverse the entire $H$ to the vertex $v_0$, continue with an edge to $x$ of $C_2$, again traverse the entire $H$ to the vertex $v_0$ and continue in this manner the traversal until using the last edge from $v_0$ of $C_n$ to $x$ of $C_1$ and the traversal is closed. It is not hard to see that the constructed cycle is a Hamiltonian cycle. Note that, if $G$ is Hamiltonian, there is no need to perform any split in $G_v$ to build a Hamiltonian cycle.

On the other hand, assume that $G_v$ contains a Hamiltonian cycle $C$ for some $v_0x \in E$. Then we can construct a Hamiltonian cycle in $G$ from $C$ as $C$ has to traverse all vertices of each $C_i$ ($1 \leq i \leq n^\alpha$) at once, without leaving the copy $C_i$. This is true since each copy is connected by the remainder of $G_x$ by only two edges that have to be used for entering and leaving the copy $C_i$. Then the traversed edges of $C$ in any $C_i$ together with the edge $v_0x$ form a Hamiltonian cycle.

Observe that splitting a vertex $u$ from the copy $C_i$ in $G_v$ can effect only vertices present in $C_i$ – by the splitting we cannot duplicate edges and hence, however vertices in $C_i$ are split, the modified graph of $C_i$ is always connected to the remainder of $G_v$ only by the two edges leaving to the previous and the next copy. Hence, if $\gamma$ splits are necessary in some graph $G_x$ to create a Hamiltonian graph, at least $\lfloor \gamma/n^\alpha \rfloor$ splits are necessary in $G$ to create a Hamiltonian graph.

If $G$ contains a Hamiltonian cycle, for one of neighboring vertices $v$ of vertex $v_0$, the graph $G_v$ contains a Hamiltonian cycle. For $G_v$, the cost of the optimal solution is hence 1, but our $(n^{1-\varepsilon})$-approximation algorithm can deliver a solution of cost at most $(n^{\alpha+1})(1-\varepsilon)$ and hence, the cost-minimum solution over all graphs $G_x$, for $v_0x \in E$, costs at most $n^{(\alpha+1)(1-\varepsilon)} = n^\alpha$. 

Figure 6.3: The graph $G_v$ that consists of $n^\alpha$ copies of a smaller graph $G$ in which an identical single edge $v_0v$ was removed.
On the other hand, if $G$ does not contain a Hamiltonian cycle, at least one split is necessary in $G$ which transforms into $n^\alpha$ necessary splits in any assumed graph $G'_v$. Hence each feasible solution of Min-Split-HC on $G'_v$ costs at least $n^\alpha + 1$.

To conclude our proof, if $A$ would exist, we would be able to solve the $NP$-complete problem of deciding whether $G$ contains a Hamiltonian cycle in polynomial time.

We would like to note that our proof and the algorithm can be extended to other classes of graphs on which decidability of the Hamiltonicity property is $NP$-hard, the class is closed under the operation of splitting and, furthermore, the graphs constructed by the addition (and the removal) of edges $v_0v$ between the copies $C_i$ ($v_0v$ in each copy) keeps the membership in the class.

As a final remark, note that, in graph-TSP, the cost of an optimal solution is $n$ and any constant approximation algorithm computes a feasible solution of cost $\Omega(n)$. In Min-Split-HC is the situation similar – the cost one of an optimal solution and our inapproximability result only implies that the cost of any approximation solution is $\Omega(n)$, too.

6.4 Summary

In this chapter, we have studied several optimization problems that model one possible way of repairing of corrupted instances of hard decision problems. All the studied problems are $NP$-hard.

We are not aware that any of the studied problems was previously considered in any context.

Our results suggest that, in the study of such optimization problems, one should consider the following concepts that arose in our problems and that seem to be rather general.

- The problem should be carefully defined such that, for each feasible problem instance, there has to exist a feasible solution (i.e., the problem is in $NP\text{-}O$). Moreover, the cost of an optimal solution should be positive such that, even though the underlying problem is $NP$-hard, it is still reasonable to consider approximation algorithms – any approximation algorithm for a problem with an optimal solution of zero cost decides whether the given instance satisfies the underlying decision problem or not. (We have experienced similar concerns for both Min-Split-$k$-SAT and Min-Split-HC.)

- If there is an additive constant in the cost function and the underlying decision problem is $NP$-hard, one could try to blow up the size of given instances polynomially. Then one may obtain an inapproximability result through a possible gap (which could emerge due to the additive constant) between the cost of a feasible solution if the instance satisfies the underlying decision problem and the cost of any optimal solution if the instance does not satisfy the underlying decision problem. (This concept we successfully applied in the Min-Split-HC.)
• In general, the optimization problems are going to be hard and we expect that they will be not very well approximable as the underlying problems are hard as well. (Quite high inapproximability results were achieved for all three problems.)

However, we find the idea of repairing corrupted instances appealing. A very interesting open question for our further research is the behavior of similar optimization problems that are based on problems that can be decided in polynomial time. It is an interesting question whether, generally, a polynomial-time decidable problem converted into an optimization problem which minimizes operations to repair its instances will turn out to be hard or not.
Chapter 7

Exact Algorithms for Uniform Min-Ones-2-SAT

In this chapter, we present our gradual development of exact algorithms for Uniform Min-Ones-2-SAT. In the first section, we define the problem and give some basic notions we will be using throughout this chapter. The previously known results are presented at the end of this section. In the subsequent two sections, we present our two exact algorithms, the last section is devoted to a biological application of Uniform Min-Ones-2-SAT.

7.1 Definitions and Overview

We consider Boolean formulas in 2-CNF over a set of variables $X = \{x_1, \ldots, x_n\}$.

The definitions regarding Boolean logic are summarized in Section 2.3 and the overview of known results for SAT is presented in Subsection 2.8.3. Recall that a 2-CNF clause is called positive, if both literals in it are unnegated variables, it is called negative, if both literals are negated variables, and mixed, if it contains one negated and one unnegated variable.

Problem 7.1.1. **Min-Ones-2-SAT** is the following optimization problem:

**Input:** A satisfiable 2-CNF formula $\varphi = C_1 \land C_2 \land \cdots \land C_m$ over the set of variables $X = \{x_1, \ldots, x_n\}$.

**Feasible solutions:** All satisfying assignments for $\varphi$.

**Costs:** The cost of a satisfying assignment $\beta$ is the number of variables set to 1 by $\beta$.

**Goal:** Minimization.

Uniform Min-Ones-2-SAT is the restriction of Min-Ones-2-SAT to input instances without mixed clauses, and Positive Min-Ones-2-SAT is the restriction of Uniform Min-Ones-2-SAT to inputs containing only positive clauses.
For our algorithms, we are going to use a graph representation for the \textsc{Uniform Min-Ones-2-SAT}, where the vertices of the graph correspond to the variables and the edges correspond to the clauses.

**Definition 7.1.2.** For a formula $\varphi$ in 2-CNF without mixed clauses, we define the satisfiability graph $G_{\varphi} = (X, E_+ \cup E_-)$, where $E_+ = \{xy \mid (x \lor y) \text{ is a positive clause in } \varphi\}$ and $E_- = \{xy \mid (\overline{x} \lor \overline{y}) \text{ is a negative clause in } \varphi\}$. We call the edges from $E_+$ positive edges and those from $E_-$ negative edges. By $G_{\varphi}^+$ and $G_{\varphi}^-$, we denote the graph induced by the edges $E_+ \text{ and } E_-$, respectively.

Note that the satisfiability graph is actually a multigraph, since two vertices may be connected by both a positive and a negative edge. Recall that a \textit{vertex cover} of a graph $G = (V, E)$ is a set $C \subseteq V$ of vertices such that every edge is incident to at least one vertex from $C$. An \textit{independent set} of a graph $G = (V, E)$ is a set $I \subseteq V$ of vertices that are pairwise non-adjacent. The \textit{minimum vertex cover problem} (MIN-VC) is the problem of finding a vertex cover of minimum cardinality in a given graph, whereas the \textit{maximum independent set problem} is the problem of finding an independent set of maximum cardinality in a given graph. For more details regarding the two problems see Subsection 2.8.5.

**Observation 7.1.3.** \textsc{Positive Min-Ones-2-SAT} is equivalent to the minimum vertex cover problem.

**Proof.** Observe that the variables set to be 1 in \textsc{Positive Min-Ones-2-SAT} correspond to the vertices picked for a vertex cover in the satisfiability graph of the instance. The claim follows immediately from this observation.

Note that, in every positive clause, at least one variable has to be set to 1 and in every negative clause at most one variable may be set to 1 in order to satisfy the formula. Considering our graph representation, this leads to the following observation.

**Observation 7.1.4.** \textsc{Uniform Min-Ones-2-SAT} is equivalent to the problem of finding a minimum vertex cover in $G_{\varphi}^+$ that is an independent set in $G_{\varphi}^-$.  

**Proof.** The equivalence of \textsc{Uniform Min-Ones-2-SAT} with the minimum vertex cover problem follows from Observation 7.1.3 as \textsc{Uniform Min-Ones-2-SAT} on $G_{\varphi}^+$ is \textsc{Positive Min-Ones-2-SAT}.

Each edge $uv$ of $G_{\varphi}^-$ corresponds to a formula $(\pi \lor \overline{\pi})$ that cannot have both variables set to 1. Thus the solution corresponds to a set of vertices in the graph representation that is independent.

The \textsc{Min-Ones-2-SAT} was first considered by Gusfield and Pitt [62] who presented a 2-approximation algorithm for it. Although it is possible to decide the satisfiability of a 2-CNF formula in linear time (see Theorem 2.8.19), the \textsc{Min-Ones-2-SAT} problem is $\mathcal{NP}$-hard and even $\mathcal{APX}$-hard since it is a generalization of the $\mathcal{APX}$-complete minimum vertex cover problem (Theorem 2.8.30 with $d = 2$).
7.2. Our First Exact Algorithm

A couple of months after we achieved the results presented in this chapter, Misra et al. [84] presented a polynomial-time reduction from MIN-ONES-2-SAT to MIN-VC that preserves the size of the instance. Thus, by Observation 7.1.3, the MIN-ONES-2-SAT and MIN-VC are equal. Besides the result of Misra et al., our algorithm from Section 7.3 matches the time complexity of the currently best known exact algorithm for MIN-VC and thus was not yet overcome by this result.

Both presented exact algorithms are based on the standard approach – we design data reduction rules and branching rules and we employ them in a branch-and-bound algorithm to reduce the size of the given instance of our problem. In Section 7.2, we present the first nontrivial exact algorithm for UNIFORM MIN-ONES-2-SAT with $n$ variables. Its time complexity is moderate: $O^*(1.25993^n)$. The structure of the instance is analyzed in more detail and the data reduction rules are improved in Section 7.3. Here, we improve our exact algorithm to the time complexity $O^*(1.2127^n)$ to match the best known exact algorithm for MIN-VC.

Additionally, we show that UNIFORM MIN-ONES-2-SAT is fixed-parameter tractable by showing that our algorithm can be adapted to verify in $O^*(2^k)$ time whether an assignment with at most $k$ variables set to 1 exists.

7.2 Our First Exact Algorithm

The algorithm for UNIFORM MIN-ONES-2-SAT presented here is based on the well known branch-and-bound technique. We pick a small subgraph of the satisfiability graph that has a simple structure and we branch on one of its vertices – we either take the vertex into the vertex cover or not. Due to the properties of the selected subgraph, we are able to deduce the presence of other vertices in the cover and thus to prune some branches of the exhaustive search.

Our approach is based on the following observation about alternating trees in the satisfiability graph. Here, an alternating tree is a subtree $T$ of the satisfiability graph, rooted in some vertex $r$, such that all paths from the root alternate between positive and negative edges. An alternating tree is called positive, if all edges incident to its root $r$ are positive, and it is called negative, if all edges incident to $r$ are negative.

**Observation 7.2.1.** Let $G_\varphi = (X, E_+ \cup E_-)$ be the satisfiability graph of a uniform 2-CNF formula $\varphi$, and let $x$ be some vertex of $G_\varphi$. If $x$ gets assigned the value 1, this induces a value for every vertex of any negative alternating tree rooted in $x$. Moreover, if $x$ gets assigned the value 0, this induces a value for every vertex of any positive alternating tree rooted in $x$.

**Proof.** Let $v_0, v_1, \ldots, v_p$ be a negative alternating path in $G_\varphi$, i.e., $(\overline{v_{2i}} \lor \overline{v_{2i+1}})$ and $(v_{2i+1} \lor v_{2i+2})$ are clauses of $\varphi$, for all $0 \leq i \leq \lfloor (p - 1)/2 \rfloor$. Observe that, once we assign $x = v_0$ value 1, the rest of the path is determined – value 1 is assigned to all vertices with even index and value 0 is assigned to all vertices with odd index. Thus assigning the value 1 to the root of a negative alternating tree uniquely determines the values of the remaining vertices of the tree. The proof for the positive alternating path is analogous. \qed
The overall strategy of our algorithm is to use some data reduction rules based on Observation 7.2.1 and branching rules to eliminate all negative edges from the satisfiability graph and to subsequently apply a fast exact algorithm for the minimum vertex cover problem (Min-VC) on the remaining graph (see Theorem 2.8.44).

More precisely, the application of each rule can be divided into two steps. First, we guess the membership in the vertex cover for a constant number of vertices, and then we deduce information about other vertices in the neighborhood. Whenever this deduction process shows an inconsistency, the current branch of the search fails and is terminated immediately.

We will prove that, once none of our rules can be applied, all negative edges have been removed. Then the branching terminates and we employ an exact algorithm for the minimum vertex cover problem in order to obtain one of the final solutions. Out of all final solutions, we then pick the best one.

The data reduction and branching rules have to be applied sequentially in the order described below. With each application of a rule, some subset of the vertices is chosen to be included either in the vertex cover $C$ or in the non-vertex-cover set $N$. Whenever a vertex is assigned to be in $C$ or in $N$, this is propagated along the corresponding maximal alternating tree according to Observation 7.2.1, which can be found easily by a depth- or breadth-first search algorithm. These rules are shown graphically in Figure 7.1.

1. Insert all vertices from $V(G^-_\varphi) \setminus V(G^+_{\varphi})$ into $N$.
2. Use exhaustive search on all connected components containing at most four vertices to assign these vertices to $C$ and $N$.
3. If there exists a triangle of unassigned vertices consisting of two positive edges $uv$ and $vw$ and one negative edge $uw$, then $v$ has to be inserted into $C$.
4. If there exists an unassigned vertex $v \in V(G^+_{\varphi}) \cap V(G^-_{\varphi})$ that has at least two incident edges in $E_+$ and also two incident edges in $E_-$, consider two branches according to whether $v$ is inserted into $C$ or $N$.
5. If the graph contains an alternating cycle of length 4, branch over the two possible assignments for an arbitrary vertex on this cycle.
6. If the graph contains an alternating path on unassigned vertices that has length at least 4, branch over the two possible assignments for the third vertex of the path (if the length of path is exactly 4, this is the middle vertex).
7. If there are two unassigned vertices $v$ and $u$ connected by both a positive and a negative edge, exactly one of them has to belong to $C$. Consider two sub-cases:
   
   (a) If all other edges incident to $u$ are positive and all other edges incident to $v$ are negative, insert $u$ into $C$. 

7.2. Our First Exact Algorithm

(b) Otherwise, branch over the two possible choices of inserting \( u \) or \( v \) into \( C \).

8. If there exists an unassigned vertex \( v \) that is incident with two positive edges and one negative edge, branch over the two possibilities of inserting \( v \) into \( C \) or \( N \).

9. If there exists a triangle of unassigned vertices consisting of one positive edge \( uv \) and two negative edges \( uw \) and \( vw \), branch over the choice of inserting \( u \) or \( v \) into \( C \).

10. Insert all unassigned vertices incident to negative edges into \( N \) and their neighbors via positive edges into \( C \) and compute a minimum vertex cover on the graph induced by the remaining unassigned vertices.

We now prove the correctness of this algorithm and analyze its running time. We start by describing the situation when none of the first nine rules can be applied any more.

**Lemma 7.2.2.** Suppose that the first nine of the above data reduction and branching rules cannot be applied any further. Let \( G' = (V', E') \) be the graph induced by the still unassigned vertices, and let \( E'_+ = E' \cap E_+ \) and \( E'_- = E' \cap E_- \) denote the set of positive and negative edges of \( G' \), respectively. Let \( V'_+ \subseteq V' \) and \( V'_- \subseteq V' \) be the set of vertices incident to some positive and negative edge in \( G' \), respectively.

Then every vertex \( x \in V'_- \) is incident to exactly one positive edge whose other endpoint is in \( V'_+ \).

**Proof.** We first show that, under the preconditions of the lemma, no two vertices in \( V'_- \) can be connected via a positive edge. Towards contradiction, suppose that there exist \( x, y \in V'_- \) such that \( xy \in E'_+ \). Note that this edge is included in a connected component of \( G' \) of size at least five due to rule 2. Since \( x, y \in V'_- \), there exist \( u, v \in V'_- \) such that \( xu, yv \in E'_- \). Moreover, since rule 9 cannot be applied, we have \( u \neq v \) and since also rule 7 cannot be applied, \( x \neq v \) and \( y \neq u \). Thus, all four vertices \( x, y, u, v \) have to be different.

We now consider the positive edge \( up \) for \( p \neq x \) that has to be present in the graph \( G' \) since otherwise rule 1 would be applicable. If vertex \( p \) would lie outside the set \( \{x, y, u, v\} \), rule 6 would be applicable. Thus, \( \{x, y, u, v, p\} = \{x, y, u, v\} \). If \( p = v \), we would have an alternating cycle \( x, y, v, u, x \) of length 4, which would make rule 5 applicable. Thus, the only remaining possibility is \( p = y \). But in this case, rule 3 would be applicable on the triangle \( x, y, u, x \). Thus, the existence of the positive edge \( xy \) leads to a contradiction and we have proved that no two vertices in \( V'_- \) can be connected via a positive edge.

Now we prove the second claim of the lemma. As we have shown above, all positive edges incident to vertices from \( V'_- \) have to have their second endpoints in \( V'_+ \). The existence of a vertex \( v \) in \( V'_- \) that is an endpoint of two different positive edges would make rule 8 applicable, since \( v \) has to have a neighbor \( u \) via a negative edge, and \( u \) has to have a neighbor in \( V'_+ \) as well, due to rule 1.

\[ \square \]
Figure 7.1: An illustration of the data reduction and branching rules of the presented exact algorithm for Uniform Min-Ones-2-SAT. Solid lines denote positive edges, dashed lines denote negative edges. Vertices in $C$ are drawn black, vertices in $N$ are drawn white; the black-and-white vertices are still unassigned.
Lemma 7.2.3. **Rules 1 to 10 are correct and thus our algorithm is correct.**

*Proof.** We start with the non-branching rules. It is clear that a vertex that is incident to negative edges only, cannot appear in any minimum vertex cover on the positive edges. Thus, the application of rule 1 is correct. The correctness of rule 2 is obvious, since all connected components can be treated independently. For rule 3, assume that \( v \in N \). Due to the two positive edges in the triangle, both \( u \) and \( w \) have to be in \( C \), contradicting the negative edge between them. Thus, \( v \in C \), and rule 3 is correct. Now consider rule 7 (a). The double edge between \( u \) and \( v \) implies that exactly one of these two vertices has to be included in \( C \). Choosing \( u \in C \) and \( v \in N \) implies that all edges incident to \( u \) and \( v \) are satisfied. This leaves the highest possible degree of freedom for choosing the assignment for the neighboring vertices and thus leads to a minimum vertex cover solution.

Considering the branching rules 4 to 6, 7 (b), and 8, we observe that the algorithm always branches over the two possible assignments for a single vertex and then applies Observation 7.2.1 for inferring the assignment for some further vertices. Therefore, the correctness of these rules follows directly from Observation 7.2.1. For the branching rule 9, it suffices to observe that at least one of the vertices \( u \) and \( v \) has to be included in \( C \).

It remains to prove the correctness of rule 10. From Lemma 7.2.2, we know that every vertex \( x \) that is incident to a negative edge, is incident to exactly one positive edge, and furthermore, that the other endpoint \( y \) of this positive edge is incident to positive edges only. This means, instead of inserting \( x \) into \( C \), we can insert \( y \) into \( C \) without changing the cost of the solution. Thus, rule 10 and therefore, the whole algorithm are correct. \( \square \)

Now we analyze the time complexity of our algorithm. The main point here is that, in every branching step, we significantly reduce the number of unassigned vertices.

**Lemma 7.2.4.** Any application of one of the branching rules 4 to 6, 7 (b), 8, and 9 reduces the number of unassigned vertices in every branch by at least 3.

*Proof.** In the following, we prove this separately for every branching rule (see Figure 7.1).

For rule 4, assigning \( v \in C \) in the first branch implies that its at least two neighbors via negative edges have to be included into \( N \), giving a total of at least 3 newly assigned vertices. For assigning \( v \in N \) in the second branch, analogously at least two neighbors via positive edges have to be included into \( C \).

For an alternating cycle of even length, it is easy to see that fixing the assignment for some arbitrary vertex triggers the assignment for all other vertices in the cycle. Thus, the application of rule 5 reduces the number of unassigned vertices by at least 4.

For an alternating path containing at least 4 edges, the third vertex \( v \) of this path is the root of two different alternating paths of length at least 2, one starting with a positive edge, the other starting with a negative edge. Due to Observation 7.2.1, this implies that any assignment for \( v \) triggers the assignment for one
of these alternating paths, and thus for at least two additional vertices. Thus, also
the branching according to rule 6 reduces the number of unassigned vertices by at
least 3.

When applying rule 7 (b), we consider the case of a double edge $uv$ and branch
over the two possible assignments for it. Additionally, we know that rule 7 (a) is
not applicable. Thus, we distinguish three cases for the analysis (see Figure 7.1).
Assume first that $u$ and $v$ are incident to some additional positive edge each. Let $x$
and $y$ be their neighbors via these edges, respectively. Then, in the branch where
$u$ is included in $N$, its neighbor $x$ has to be inserted into $C$. Analogously, in the
other branch, $y$ has to be inserted into $C$.\footnote{Note that the case where $x = y$
is already covered by rule 3.} Now assume that $u$ has a neighbor $x$ via a negative edge and $v$ has a neighbor $y$ via a negative edge. Analogously
as in the preceding case, if $u$ is included in $C$, then $x$ has to be in $N$, and if $v$
is included in $C$, then $y$ has to be in $N$.\footnote{Note that, in the case $x = y$, no branching is necessary since this vertex has to be in $N$, anyway.} The remaining case is that $v$ (or $u$) is
incident to a positive and a negative edge with endpoints $x$ and $y$, respectively. In
this case, the assignment for $v$ triggers the assignment of either $x$ or $y$ according
to Observation 7.2.1. Thus, applying rule 7 (b) assigns values to at least three
vertices in any case.

In the situation of rule 8, denote by $u$ the neighbor of $v$ via the negative edge
and by $w_1$ and $w_2$ the two neighbors of $v$ via the positive edges. Due to rule 1,
there exists a further vertex $x$ which is adjacent to $u$ via a positive edge. Since
rule 3 is not applicable, $x \neq w_1$ and $x \neq w_2$. According to Observation 7.2.1, $w_1$
and $w_2$ have to be included in $C$ in the branch where $v$ is inserted into $N$. On the
other hand, letting $v \in C$ triggers the assignment for $u$ and $x$.

Finally, we consider rule 9. Due to rule 1, there exists another vertex $x$ which
is adjacent to $w$ via a positive edge. Since rule 3 is not applicable, we know that
$x \neq u$ and $x \neq v$. Independent of whether $u$ or $v$ is inserted into $C$, we know that
$w \in N$ and thus $x \in C$. \hfill\Box

We have now proved that every branching step reduces the number of unas-
signed vertices by at least 3. This leads us to the following theorem.

**Theorem 7.2.5.** The branch-and-bound algorithm based on the rules 1 to 10 solves
**Uniform Min-Ones-2-SAT** on $n$ variables in $O^*(1.25993^n)$ time and polynomial
space.

**Proof.** Following from Lemma 7.2.4, we have the recurrence

$$T(n) = 2 \cdot T(n - 3)$$

for the time complexity of our approach for inputs with $n$ vertices. Applying
standard techniques for solving recurrence equations (see, e.g., [60]) yields an upper
bound of

$$T(n) \leq \left(\frac{3}{\sqrt{2}}\right)^n \leq 1.25993^n.$$
7.3. A Faster Exact Algorithm

In addition, we have to consider the time needed for the minimum vertex cover computation after the application of rule 10. For this, we can use any exact minimum vertex cover algorithm. Since every minimum vertex cover in a graph is a maximum independent set in the complement graph (i.e., in the graph resulting from swapping edges and non-edges), we can also use any exact algorithm for the maximum independent set problem on the complement graph. One algorithm for this is due to Bourgeois et al. (see Theorem 2.8.44) and runs in $O^*(1.2127^n)$ time. Thus, the overall running time of our algorithm can be bounded by $O^*(1.25993^n)$.

The space complexity of our algorithm is polynomial as all our structures and the computation of the algorithm of Bourgeois et al. fit well into a polynomial space.

7.3 A Faster Exact Algorithm

In this section, we improve over the algorithm presented in Section 7.2. We study the structure of instances in more detail and we improve our data reduction rules, such that the branching factor is decreased. We again reduce our instance to an instance of minimum vertex cover problem (Min-VC) without increasing the size of the graph. Afterwards, we plug in the best known exact algorithm solving Min-VC in $O^*(1.2127^n)$ time and polynomial space (see Theorem 2.8.44). We design our rules such that the overall branching factor of our part is smaller than 1.2127 and thus the time complexity of the entire algorithm is limited by its slowest part – the exact algorithm for the minimum vertex cover problem.

We call a branching an $(a, b)$-branching if in one branch we can deduce the state of $a$ vertices and in the other branch the state of $b$ vertices (including the one we branch on). In order to achieve a better branching factor than 1.2127, instead of the (3, 3)-branching used in Section 7.2, we use either a (6, 2)-, a (5, 3)-, or a (4, 4)-branching which gives us a better factor (see Table 7.1).

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.00000</td>
<td>1.61804</td>
<td>1.46558</td>
<td>1.38028</td>
<td>1.32472</td>
<td>1.28520</td>
<td>1.25543</td>
</tr>
<tr>
<td>2</td>
<td>1.61804</td>
<td>1.41422</td>
<td>1.32472</td>
<td>1.27202</td>
<td>1.23651</td>
<td>1.21061</td>
<td>1.19072</td>
</tr>
<tr>
<td>3</td>
<td>1.46558</td>
<td>1.32472</td>
<td>1.25993</td>
<td>1.22075</td>
<td>1.19386</td>
<td>1.17399</td>
<td>1.15856</td>
</tr>
<tr>
<td>4</td>
<td>1.38028</td>
<td>1.27202</td>
<td>1.22075</td>
<td>1.18921</td>
<td>1.16731</td>
<td>1.15097</td>
<td>1.13819</td>
</tr>
<tr>
<td>5</td>
<td>1.32472</td>
<td>1.23651</td>
<td>1.19386</td>
<td>1.16731</td>
<td>1.14870</td>
<td>1.13473</td>
<td>1.12374</td>
</tr>
<tr>
<td>6</td>
<td>1.28520</td>
<td>1.21061</td>
<td>1.17399</td>
<td>1.15097</td>
<td>1.13473</td>
<td>1.12247</td>
<td>1.11278</td>
</tr>
<tr>
<td>7</td>
<td>1.25543</td>
<td>1.19072</td>
<td>1.15856</td>
<td>1.13819</td>
<td>1.12374</td>
<td>1.11278</td>
<td>1.10409</td>
</tr>
</tbody>
</table>

Table 7.1: The branching factors of an $(a, b)$-branching rounded up to the 5th place after the decimal point.
7.3.1 Reduction Rules

Below, we present a list of reduction rules that will be applied during the branch-and-bound algorithms whenever their premises apply.

Dnegative: If \( v \) is incident only to negative edges, we can deduce that \( v \in N \).

Dspread: If we already decided that \( v \in C \), and we have \( uv \in E_− \), we can deduce that \( u \in N \). Similarly, from \( v \in N \) and \( uv \in E_+ \), we have \( u \in C \). This rule is based on Observation 7.2.1.

Dtriangle: If we have positive edges \( xy \) and \( xz \) and a negative edge \( yz \), we can deduce that \( x \in C \). If we have negative edges \( xy \) and \( xz \) and a positive edge \( yz \), we can deduce that \( x \in N \). The rule is in accordance with rules 3 and 9 of Section 7.2.

Dgreedy: If \( v_1, \ldots, v_k \) (for some \( k \geq 1 \)) and \( u \) are vertices such that:

- for each \( i \) there is a positive edge \( v_iu \)
- no \( v_i \) has any other positive edges
- there are no negative edges between \( u \) and vertices other than \( v_i \)

then we can just consider the case where \( u \in C \) and \( v_i \in N \), for all \( 1 \leq i \leq k \).

Lemma 7.3.1. All presented reduction rules are correct. That is, applying any of the rules will never cause our search to skip all optimal valid assignments.

Proof. We will consider the reduction rules one at a time.

If a variable \( v \) only appears in negative clauses, we take any valid assignment where \( v \) is set to 1 and change it to be 0, thereby obtaining a better valid assignment. Hence, we can just consider the case where \( v \) is set to 0 and so Dnegative is correct.

By Observation 7.2.1, the Dspread rule is correct: Let \( v \in C \) and \( uv \in E_− \). Then the only remaining way to satisfy the clause \((\overline{v} \lor \overline{u})\) is to set \( u \) to 0. If \( v \in N \) and \( uv \in E_+ \), then the only way to satisfy the clause \((u \lor v)\) is to set \( u \) to 1.

Consider now the situation when rule Dtriangle can be applied. At least one of \( y \) and \( z \) must be set to 0 in any valid assignment. As there is a positive edge connecting that variable to \( x \), it follows that \( x \) must be set to 1 in any valid assignment. The other case is proved similarly. At least one of \( y \) and \( z \) must be 1 in any valid assignment. As there is a negative edge connecting that variable to \( x \), it follows that \( x \) must be 0 in any valid assignment.

Consider now any valid assignment of a subgraph where rule Dgreedy can be applied. Assume that, for some \( i \), we have \( v_i \in C \), that is, the variable \( v_i \) is set to 1. This assignment can be changed as follows. Set all \( v_i \) to 0. If \( u \) is 0, set it to 1. This change did not increase the number of variables set to 1 and it preserved the validity of the assignment. To see why the latter holds, note that all clauses that only contain \( u \) and one of the \( v_i \) are satisfied by our new assignment,
all other clauses containing \( u \) are positive and therefore satisfied, and all other clauses containing the \( v_i \) are negative and therefore satisfied.

Hence, when looking for an assignment that minimizes the number of variables set to 1, it is sufficient to consider the one chosen by this rule.

\[ \square \]

### 7.3.2 The Algorithm and its Branchings

Our algorithm consists of several phases, as described below.

**Phase 1:** We use `Dnegative` to ensure that each vertex is incident to a positive edge. This corresponds to rule 1 of the algorithm from Section 7.2.

**Phase 2:** While there is some vertex incident to at least 5 negative edges, we pick any such vertex \( v \) and branch on it (with applying `Dspread` afterwards).

**Phase 3:** For any vertex \( v \) incident to some negative edges, we analyze all possible cases how its neighborhood looks like. In each case, we either show that we can deduce the state of some vertex, or we find a vertex such that branching on it produces two sufficiently smaller instances.

**Phase 4:** Once only positive edges are left, we apply a general algorithm to find a minimum vertex cover.

In some situations, our algorithm will branch on a given vertex \( v \), meaning that we sequentially try first the case \( v \in C \) and then the case \( v \in N \). In each case, we will be applying the reduction rules (usually the rule `Dspread`) to deduce as much information as possible.

Now we give a detailed description of Phases 2 and 3 of our branch-and-bound algorithm. To achieve the desired running time, any branching must be at least a \((6,2)\)-, a \((5,3)\)-, or a \((4,4)\)-branching. We will call all such branchings acceptable.

We introduce some additional notation we will use in the proof. The phrase positive neighbor is shorthand for “neighbor in the graph \((X,E_+)\)”, analogously for negative neighbor. The degree of \( v \) in \( G_+^v \) is denoted \( \Delta_+(v) \), \( \Delta_-(v) \) is defined analogously. We also define \( \Delta_\pm(v) = (\Delta_+(v), \Delta_-(v)) \) and \( \Delta(v) = \Delta_+(v) + \Delta_-(v) \).

In each of the cases presented below, \( v \) will be some vertex incident to negative edges, \( n_{1}, \ldots, n_{s_v} \), for some \( s_v \in \mathbb{N} \), will be its negative neighbors and \( p_1, \ldots, p_{t_v} \), for some \( t_v \in \mathbb{N} \), will be its positive neighbors. If \( \ell \) of the \( n_i \) coincide with the \( p_j \), we will label them such that, for \( 1 \leq k \leq \ell \), we have \( n_{s_v-\ell+k} = p_k \), i.e., the last few \( n_i \) are equal to the first few \( p_j \).

In all figures, the vertices shown as squares are already final, i.e., there are no other edges leaving these vertices. Positive edges are shown as full lines, negative edges are shown as dashed lines, the edges where we do not need to know whether they are positive or negative are dotted.

In the following, we discuss the invariants and possible branchings of the algorithm during its four phases.
Lemma 7.3.2. After Phase 1, each unassigned vertex $v$ has $\Delta_+(v) > 0$.

Proof. All vertices $v$ with $\Delta_+(v) = 0$ are assigned into $N$ by rule $D_{\text{negative}}$ during the Phase 1.

Take any vertex $v$ and let $\Delta_\pm(v) = (a,b)$. Regardless of the rest of the graph, if we branch on $v$, this will be at least an $(a+1, b+1)$-branching. If $b > 4$, this branching is guaranteed to be acceptable and we can apply it immediately.

Lemma 7.3.3. The branching during Phase 2 is acceptable.

Proof. Each vertex $v$ in Phase 2 has $\Delta_+(v) \geq 1$. If also $\Delta_-(v) \geq 5$, then the branching at $v$ is at least a $(6,2)$-branching (the vertex $v$ is included in both branchings) which is acceptable.

Lemma 7.3.4. Each possible branching that may occur during Phase 3 is acceptable.

Proof. We sequentially analyze all possible cases how a small neighborhood around a vertex $v$ incident to at least one negative edge may look like.

Vertex with a large degree:

Let $v$ be an arbitrary vertex with $\Delta(v) > 5$ and $\Delta_-(v) > 0$. Then the branching at $v$ is clearly acceptable and we can apply it immediately.

After this process terminates, we get a graph in which each vertex is incident to at least one positive edge and at most four negative edges. Additionally, a vertex adjacent to some negative edges has degree at most 5. We will now show how to process the remaining vertices incident to negative edges.

Vertices with four negative edges:

Suppose that $\Delta_-(v) = 4$. Then we must have $\Delta_\pm(v) = (1,4)$. There has to be a positive edge incident to $n_1$. If it leads into a new vertex, by $D_{\text{spread}}$ we get at least a $(6,2)$-branching at $v$, otherwise we can apply the rule $D_{\text{triangle}}$.

Vertices with three negative edges:

The case $\Delta_\pm(v) = (2,3)$ is handled in the same way as $\Delta_\pm(v) = (1,4)$, except this time the branching is at least $(5,3)$ instead of at least $(6,2)$.

Now let $\Delta_\pm(v) = (1,3)$. There is at least one positive edge incident to each of $n_1$ and $n_2$. If there is a positive edge $xy$ with $x \in \{n_1, n_2\}$ and $y \in \{n_1, n_2, n_3, p_1\}$, we can apply $D_{\text{triangle}}$. If two positive edges lead from $\{n_1, n_2, n_3\}$ into two distinct new vertices, we get at least a $(6,2)$-branching at $v$.

Finally, we are left with one of the cases shown in Figure 7.2. If there is a negative edge from $a$ into a new vertex, we get a $(6,2)$-branching at $v$. If there is a negative edge $ap_1$, we get at least a $(5,3)$-branching at $v$. In all other cases we can apply $D_{\text{greedy}}$ at $a$. 
7.3. A Faster Exact Algorithm

From this moment on, we may assume that each vertex in our graph is incident to at most two negative edges.

Vertices with two negative edges:

*Three positive edges:* We have $\Delta_\pm(v) = (3, 2)$.

If there is no positive edge $vn_1$, there must be a positive edge leaving $n_1$. If its second endpoint is a neighbor of $v$, we apply $D_{\text{triangle}}$, otherwise we have at least a $(4, 4)$-branching at $v$.

Two positive edges: We have $\Delta_\pm(v) = (2, 2)$.

If there is a positive edge $n_1n_2$, a negative edge $p_1p_2$, or any edge $xy$ with $x \in \{n_1, n_2\}$ and $y \in \{p_1, p_2\}$, we apply $D_{\text{triangle}}$.

We will now consider three sub-cases depending on the number of $n_i$ and $p_j$ that coincide. These cases are shown in Figure 7.4.

In the left case, we must have positive edges leaving $n_1$ and $n_2$ into new vertices. If these are two distinct vertices, we get a $(5, 3)$-branching at $v$. Otherwise label their common vertex $a$. If there is a negative edge from $a$ to a new vertex or to one of the $p_i$, we get an acceptable branching at $v$. In the other case, we can apply $D_{\text{greedy}}$ at $a$. 

Figure 7.2: Two cases while handling a vertex $v$ with three negative edges.

Figure 7.3: The last considered case for three positive and two negative edges.
In the middle case, we must have a positive edge $n_1a$ where $a$ is a new vertex. We apply the same reasoning at $a$ as in the previous case.

We are left with the case on the right. If one of $n_1$ and $n_2$ has degree 2, we can remove this vertex and $v$ from the graph, find the optimal solution for the smaller graph, and then reattach the two vertices and determine their values. Otherwise, we have at least one more edge leaving each of the vertices. If there is an edge $n_1n_2$, we apply $D_{\text{triangle}}$. If there are edges into two new vertices, we get an acceptable branching at $v$.

The last remaining case is shown in Figure 7.5. If the edges $n_1a$ and $n_2a$ are of different types, we have an acceptable branching at $v$. If they are both negative, we apply $D_{\text{greedy}}$ at $v$. If they are both positive, we can remove the vertices $v$ and $n_1$, add a negative edge $n_2a$, solve the smaller problem and use the solution to reconstruct the optimal solution. To see the correctness of this construction, note that there are two possible assignments for $v, n_1, n_2$, namely $v \in N, n_1, n_2 \in C$ or $v \in C, n_1, n_2 \in N$. The latter, more desirable, assignment is only feasible if $a \in C$. Thus, $n_2$ has a different value than $a$ in any optimal solution which can be guaranteed by the additional negative edge.

**One positive edge:** We have $\Delta_{\pm}(v) = (1, 2)$.

If there is a positive edge $n_1n_2$ or any edge $n_ip_1$, we apply $D_{\text{triangle}}$.

Again, we now have two separate cases: either $n_2 = p_1$ or not.

Case $n_2 \neq p_1$. If there is no negative edge at $p_1$, we apply $D_{\text{greedy}}$ at $p_1$. If there are positive edges from $n_1$ and $n_2$ into two distinct new vertices, we get at least a $(5, 3)$-branching at $v$. Otherwise, there are exactly two positive
edges \( n_1a \) and \( n_2a \) (see Figure 7.6). If there are no negative edges incident to \( a \) or the only negative edges incident to \( a \) are \( n_1a \) or \( n_2a \), we apply \( D\text{greedy} \) at \( a \), otherwise we get an acceptable branching at \( a \).

![Figure 7.6: The remaining case for one positive and two negative edges with \( n_2 \neq p_1 \).](image)

Case \( n_2 = p_1 \). Note that there is no edge between \( n_1 \) and \( n_2 \). If there is no other negative edge at \( n_2 \), we apply \( D\text{greedy} \) at \( n_2 \). Otherwise we have a negative edge \( n_2a \). Both \( n_1 \) and \( a \) must be incident to positive edges. If there is a positive edge \( n_1a \), we get at least a \((4, 4)\)-branching at \( v \). If there are positive edges from \( n_1 \) and \( a \) to new vertices (not necessarily distinct ones), we get at least a \((4, 4)\)-branching at \( v \). The last remaining possibility are positive edges \( n_1b \) and \( n_2a \), which gives us a \((5, 3)\)-branching at \( v \).

Vertices with a single negative edge:

We are left with a graph in which each vertex is only incident to at most one negative edge, and each vertex incident to a negative edge has degree at most 5.

We skip the analysis for the cases \( \Delta_\pm(v) = (4, 1) \) and \( \Delta_\pm(v) = (3, 1) \). The analysis for these cases is just a simpler case of the analysis for \( \Delta_\pm(v) = (2, 1) \).

**Two positive edges:** We have \( \Delta_\pm(v) = (2, 1) \).

If there is any edge \( n_1p_i \), or a negative edge \( p_1p_2 \), we apply \( D\text{triangle} \).

We will first consider the case \( n_1 \neq p_1 \). There are either one or two positive edges incident to \( n_1 \). Both cases are shown in Figure 7.7.

In the case on the left side of Figure 7.7, consider the vertex \( a \). If there is no negative edge leaving \( a \), we can apply \( D\text{greedy} \) at \( a \). If there is one into some \( p_i \), we get an acceptable branching at \( v \). Finally, let there be a negative edge \( ab \). The new vertex \( b \) must have a positive edge. If it leads into \( a \), we get at least a \((6, 2)\)-branching at \( a \), otherwise we get at least a \((5, 3)\)-branching at \( v \).

The case in Figure 7.7 on the right can be solved as follows. If \( v \) and \( n_1 \) are both set to 0, then \( p_1, p_2, a_1, a_2 \) must all be set to 1 and we find an optimal solution for the rest of the graph. Otherwise exactly one of \( v \) and \( n_1 \) is set
to be 1. We remove vertices $v$ and $n_1$, add all four positive edges $p_i a_j$, find the optimal solution for the new graph and then deduce the values for $v$ and $n_1$. Note that this is a special $(6,2)$-branching and the added positive edges force the vertices from $\{a_1, a_2\}$ or from $\{p_1, p_2\}$ to be set to 1 (and hence one of $\{n_1, v\}$ can be set to 0).

Now for the case $n_1 = p_1$. If $\Delta(n_1) = 2$, we apply $D_{\text{greedy}}$ at $v$. Otherwise we get a positive edge $n_1 a$, as all other cases were already covered by our analysis before (see Figure 7.8). Now note that exactly one of $v$ and $n_1$ is set to be 1. Also, as a consequence, $p_2$ and $a$ may not be 0 at the same time. We now can add a positive edge $p_2 a$, remove the vertices $v$ and $n_1$, find the optimal solution for the smaller graph and afterwards deduce the truth values for $v$ and $n_1$.

A single positive edge:

Each vertex is either incident to positive edges only, or it is incident to a positive and a negative edge. Let $v$ be such that $\Delta_{\pm}(v) = (1,1)$.

If $p_1 = n_1$, we apply $D_{\text{greedy}}$ at $n_1$. If the positive edge from $n_1$ goes to $p_1$, we apply $D_{\text{triangle}}$, otherwise let its endpoint be $a$. If there are no negative edges from $a$ or $p_1$, we can apply $D_{\text{greedy}}$ at that vertex. If there is a negative edge $a p_1$, we have a $(4,4)$-branching at $v$. Finally, we might have negative edges $ab$ and $p_1 c$. Together with a positive edge that must be incident to $c$ we again have a $(4,4)$-branching at $v$.

This concludes Phase 3 of our algorithm.
**Corollary 7.3.5.** Whenever our algorithm branches, the branching vertex will be chosen to achieve at least a $(6, 2)$-, a $(5, 3)$-, or a $(4, 4)$-branching.

### 7.3.3 Time Complexity and Parameterization

In this subsection, we conclude the presentation of our algorithm by giving the estimation of its time complexity and remark its possible usage as a parameterized algorithm.

**Theorem 7.3.6.** Our algorithm correctly solves the **Uniform Min-Ones-2-SAT** on $n$ variables and its time complexity is $O^*(1.2127^n)$.

**Proof.** The correctness immediately follows from Lemma 7.3.1 and the detailed analysis of the algorithm above.

Using the algorithm of Bourgeois et al. (Theorem 2.8.44), we can find the minimum vertex cover in any graph with $n$ vertices in $O^*(1.2127^n)$ time.

By Corollary 7.3.5, each branching of our algorithm is at least a $(6, 2)$-, a $(5, 3)$-, or a $(4, 4)$-branching. Consider the recurrences that describe the time complexity of a branch-and-bound algorithm in each of these cases separately:

$$
T(n) = 2T(n - 4) + O(1) \\
T(n) = T(n - 3) + T(n - 5) + O(1) \\
T(n) = T(n - 2) + T(n - 6) + O(1).
$$

The first recurrence solves to $T(n) = O^*(1.18921^n)$, the following recurrence solves to $T(n) = O^*(1.19386^n)$ and the last one solves to $T(n) = O^*(1.21061^n)$.

Hence, even if the worst case occurs and our algorithm is forced to perform $(6, 2)$-branchings until it runs out of vertices and then uses the algorithm for the minimum vertex cover problem, its time complexity will be $O^*(1.2127^n)$. 

**Corollary 7.3.7.** Our algorithm can be modified into an exponential space algorithm that correctly solves the **Uniform Min-Ones-2-SAT** on $n$ variables in time $O^*(1.2109^n)$.

**Proof.** In Phase 4 of our algorithm, instead of employing the polynomial space exact algorithm for **Min-VC**, we apply the exponential time and space exact algorithm of Robson (Theorem 2.8.45) that solves the **Min-VC** on $n$ vertices in time $O^*(1.2109^n)$. This is actually the worst case as all the three of our recurrences have coefficients smaller than 1.2109.

We would like to note that our algorithm is just a wrapping around the exact algorithm for **Min-VC** that handles the independent-set constraint on the negative edges of the graph. If the given instance does not possess such edges (i.e., it is a **Positive Min-Ones-2-SAT** instance), our algorithm skips the first three phases and just calls the subroutine to compute the minimum vertex cover. Hence, any
ambition to speed-up our algorithm would have to also improve the exact algorithm for MIN-VC.

Additionally, our algorithm can be easily modified to obtain a FPT-algorithm for the problem parameterized by the size of the solution:

**Problem 7.3.8.** **Uniform Min-Ones-2-SAT-sol(\(k\))** is a parameterization of Uniform Min-Ones-2-SAT by the output size.

**Input:** An instance of Uniform Min-Ones-2-SAT and a parameter \(k\).

**Decision:** Is there a feasible solution of the Uniform Min-Ones-2-SAT instance of size at most \(k\)?

**Theorem 7.3.9.** Our algorithm is a FPT-algorithm solving Uniform Min-Ones-2-SAT-sol(\(k\)) in time \(O^*(2^k)\).

**Proof.** Phase 1 runs in polynomial time. In Phase 2 and Phase 3, we can easily verify that each branching is done on a vertex incident to a positive and a negative edge. Hence, in each branch we set at least one new variable to 1. This implies that we can limit the search-tree depth to \(k\). If a solution with at most \(k\) variables set to 1 exists, we will still find it.

For Phase 4, we can use the algorithm by Chen, Kanj and Jia (Theorem 2.8.39), which is an \(O^*(kn + 1.2852^k)\) algorithm for the parameterized version of the minimum vertex cover problem.

### 7.4 Application to a Haplotype Classification Problem

With the enhancement of sequencing methods, more and more genetic data becomes available for research. One of the grand challenges in molecular biology is to interpret this genomic data and to make use of it, for example to detect genetic diseases. In this chapter, we investigated a challenging combinatorial problem which arises in this context.

Humans and many other living organisms (including all vertebrates) have a diploid genome, i.e., they have two copies for each chromosome. For an individual organism, these two copies usually differ from each other as a result of their inheritance from two separate parent entities. The most common differences are the so-called single nucleotide polymorphisms (SNPs), where a single nucleotide is replaced by another one. For a given chromosome, the complete sequence information of one copy is called a haplotype.

Some of these haplotypes might show genetic variations that are responsible for a predisposition for some kind of disease. It is of great importance to find these variations for a better understanding of such diseases and for the development of more successful treatments. Roughly speaking, our goal is to find these defective haplotypes in our data. The first problem is that the genomic data generated by many sequencing experiments does not produce the haplotypes of an individual,
7.4. Application to a Haplotype Classification Problem

Table 7.2: Types of individuals according to haplotype and phenotype data

<table>
<thead>
<tr>
<th>type</th>
<th>phenotype</th>
<th>haplotypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>healthy</td>
<td>both normal</td>
</tr>
<tr>
<td>2</td>
<td>healthy</td>
<td>1 normal, 1 defective</td>
</tr>
<tr>
<td>3</td>
<td>symptoms of disease</td>
<td>1 normal, 1 defective</td>
</tr>
<tr>
<td>4</td>
<td>symptoms of disease</td>
<td>both defective</td>
</tr>
<tr>
<td>5</td>
<td>healthy</td>
<td>both defective</td>
</tr>
<tr>
<td>6</td>
<td>symptoms of disease</td>
<td>both normal</td>
</tr>
</tbody>
</table>

but rather a mixture of both haplotypes, the so-called *genotype*. It is a challenging problem by itself to compute the haplotypes from the genotype, but some successful algorithms have been presented in the literature, for an overview see, for example, [20, 27, 82].

We do not want to consider this problem here, rather we assume that we have access to some reliable haplotype data. Together with this haplotype data for some population of individuals, we are given some phenotypical data describing for each individual whether it shows some symptoms of the disease or not.

It is assumed that the predisposition for the disease is connected to some genetic defect on one or both haplotypes of the individual. The goal is to learn how to distinguish haplotypes with this defect from normal ones.

We now want to classify the individuals from the population into six different types according to the relationship between their haplotypes and the observed symptoms of the disease as shown in Table 7.2. Note that the information from the second column of the table is known to us whereas the information from the third column is what we would like to deduce.

For a first attempt of classifying the haplotypes and finding the defective ones, we want to simplify the model by restricting ourselves to the first four rows of Table 7.2. This means, we assume that the population does neither include individuals which show symptoms of the disease but have two normal haplotypes nor healthy individuals with two defective haplotypes. Intuitively, this means, we assume that an individual with two defective haplotypes will almost surely show symptoms of the disease, whereas it is rare for an individual to show symptoms of the disease without genetic predisposition.

This problem can now be modelled by **Uniform Min-Ones-2-SAT**: Consider the haplotypes as the variables of the formula, where an assignment of 1 to a variable means to classify the respective haplotype as defective. Then every individual of the population describes one clause of the formula. For a healthy individual, at least one of its two haplotypes has to be normal, this corresponds to a negative clause. On the other hand, if an individual shows symptoms of the disease, this means that at least one of its haplotypes has to be defective, corresponding to a positive clause. Therefore, finding an assignment with as few as possible ones, represents an explanation for the disease with the smallest possible number of
defective haplotypes. This mirrors our expectation that the population should contain more normal haplotypes than defective ones.
Bibliography


of Lecture Notes in Computer Science, pages 1194–1204. Springer-Verlag, 2005. 87


[85] Tobias Mömke, 2010. Personal communication. 101


