Foundations of computing Volume 14

Matthias Bentert

Elements of Dynamic and 2-SAT Programming: Paths, Trees, and Cuts

Universitätsverlag der TU Berlin

Matthias Bentert

Elements of Dynamic and 2-SAT Programming: Paths, Trees, and Cuts

Foundations of computing | 14

Matthias Bentert

Elements of Dynamic and 2-SAT Programming: Paths, Trees, and Cuts

Universitätsverlag der TU Berlin

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the
Deutsche Nationalbibliografie; detailed bibliographic data
are available on the internet at http://dnb.dnb.de.

Universitätsverlag der TU Berlin, 2021
http://www.verlag.tu-berlin.de

Fasanenstr. 88, 10623 Berlin
Tel.: +49 (0)30 314 76131 / Fax: -76133
E-Mail: publikationen@ub.tu-berlin.de

Zugl.: Berlin, Techn. Univ., Diss., 2020
Gutachter: Prof. Dr. Rolf Niedermeier (TU Berlin)
Gutachter: Prof. Dr. Cristina Bazgan (Université Paris Dauphine)
Gutachter: Prof. Dr. Thore Husfeldt (Lund University & ITU Copenhagen)
Die Arbeit wurde am 17.12.2020 an der Fakultät IV unter
Vorsitz von Prof. Dr. Stephan Kreutzer erfolgreich verteidigt.

Print: docupoint GmbH
Layout/Typesetting: Matthias Bentert

ISBN 978-3-7983-3209-6 (print)
ISBN 978-3-7983-3210-2 (online)

ISSN 2199-5249 (print)
ISSN 2199-5257 (online)

Published online on the institutional repository of the
Technische Universität Berlin:
DOI 10.14279/depositonce-11462
http://dx.doi.org/10.14279/depositonce-11462

Zusammenfassung

In dieser Arbeit entwickeln wir schnellere exakte Algorithmen (schneller bezüglich der Worst-Case-Laufzeit) für Spezialfälle von Graphproblemen. Diese Algorithmen beruhen größtenteils auf *dynamischem Programmieren* und auf *2-SAT-Programmierung*. Dynamisches Programmieren beschreibt den Vorgang, ein Problem rekursiv in Unterprobleme zu zerteilen, sodass diese Unterprobleme gemeinsame Unterunterprobleme haben. Wenn diese Unterprobleme optimal gelöst wurden, dann kombiniert das dynamische Programm diese Lösungen zu einer optimalen Lösung des Ursprungsproblems. 2-SAT-Programmierung bezeichnet den Prozess, ein Problem durch eine Menge von 2-SAT-Formeln (aussagenlogische Formeln in konjunktiver Normalform, wobei jede Klausel aus maximal zwei Literalen besteht) auszudrücken. Dabei müssen erfüllende Wahrheitswertbelegungen für eine Teilmenge der 2-SAT-Formeln zu einer Lösung des Ursprungsproblems korrespondieren. Wenn eine 2-SAT-Formel erfüllbar ist, dann kann eine erfüllende Wahrheitswertbelegung in Linearzeit in der Länge der Formel berechnet werden. Wenn entsprechende 2-SAT-Formeln also in polynomieller Zeit in der Eingabegröße des Ursprungsproblems erstellt werden können, dann kann das Ursprungsproblem in polynomieller Zeit gelöst werden. Im folgenden beschreiben wir die Hauptresultate der Arbeit.

Bei dem Diameter-Problem wird die größte Distanz zwischen zwei beliebigen Knoten in einem gegebenen ungerichteten Graphen gesucht. Das Ergebnis (der Durchmesser des Eingabegraphen) gehört zu den wichtigsten Parametern der Graphanalyse. In dieser Arbeit erzielen wir sowohl positive als auch negative Ergebnisse für Diameter. Wir konzentrieren uns dabei auf parametrisierte Algorithmen für Parameterkombinationen, die in vielen praktischen Anwendungen klein sind, und auf Parameter, die eine *Distanz zur Trivialität* messen.

Bei dem Problem Length-Bounded Cut geht es darum, ob es eine Kantenmenge begrenzter Größe in einem Eingabegraphen gibt, sodass das Entfernen dieser Kanten die Distanz zwischen zwei gegebenen Knoten auf ein gegebenes Minimum erhöht. Wir bestätigen in dieser Arbeit eine Vermutung aus der wissenschaftlichen Literatur, dass Length-Bounded Cut in polynomieller Zeit in der Eingabegröße auf Einheitsintervallgraphen (Intervallgraphen, in denen jedes Intervall die gleiche Länge hat) gelöst werden kann. Der Algorithmus basiert auf dynamischem Programmieren.

k-Disjoint Shortest Paths beschreibt das Problem, knotendisjunkte Pfade zwischen k gegebenen Knotenpaaren zu suchen, sodass jeder der k Pfade ein kürzester Pfad zwischen den jeweiligen Endknoten ist. Wir beschreiben ein

dynamisches Programm mit einer Laufzeit $n^{O((k+1)!)}$ für dieses Problem, wobei n die Anzahl der Knoten im Eingabegraphen ist. Dies zeigt, dass k-Disjoint Shortest Paths in polynomieller Zeit für jedes konstante k gelöst werden kann, was für über 20 Jahre ein ungelöstes Problem der algorithmischen Graphentheorie war.

Das Problem Tree Containment fragt, ob ein gegebener phylogenetischer Baum T in einem gegebenen phylogenetischen Netzwerk N enthalten ist. Ein phylogenetisches Netzwerk (bzw. ein phylogenetischer Baum) ist ein gerichteter azyklischer Graph (bzw. ein gerichteter Baum) mit genau einer Quelle, in dem jeder Knoten höchstens eine ausgehende oder höchstens eine eingehende Kante hat und jedes Blatt eine Beschriftung trägt. Das Problem stammt aus der Bioinformatik aus dem Bereich der *Suche nach dem Baums des Lebens* (der Geschichte der Artenbildung). Wir führen eine neue Variante des Problems ein, die wir Soft Tree Containment nennen und die bestimmte Unsicherheitsfaktoren berücksichtigt. Wir zeigen mit Hilfe von 2-SAT-Programmierung, dass Soft Tree Containment in polynomieller Zeit gelöst werden kann, wenn N ein phylogenetischer Baum ist, in dem jeweils maximal zwei Blätter die gleiche Beschriftung tragen. Wir ergänzen dieses Ergebnis mit dem Beweis, dass Soft Tree Containment *NP*-schwer ist, selbst wenn N auf phylogenetische Bäume beschränkt ist, in denen jeweils maximal drei Blätter die gleiche Beschriftung tragen.

Abschließend betrachten wir das Problem Reachable Object. Hierbei wird nach einer Sequenz von rationalen Tauschoperationen zwischen Agentinnen gesucht, sodass eine bestimmte Agentin ein bestimmtes Objekt erhält. Eine Tauschoperation ist rational, wenn beide an dem Tausch beteiligten Agentinnen ihr neues Objekt gegenüber dem jeweiligen alten Objekt bevorzugen. Reachable Object ist eine Verallgemeinerung des bekannten und viel untersuchten Problems Housing Market. Hierbei sind die Agentinnen in einem Graphen angeordnet und nur benachbarte Agentinnen können Objekte miteinander tauschen. Wir zeigen, dass Reachable Object *NP*-schwer ist, selbst wenn jede Agentin maximal drei Objekte gegenüber ihrem Startobjekt bevorzugt und dass Reachable Object polynomzeitlösbar ist, wenn jede Agentin maximal zwei Objekte gegenüber ihrem Startobjekt bevorzugt. Wir geben außerdem einen Polynomzeitalgorithmus für den Spezialfall an, in dem der Graph der Agentinnen ein Kreis ist. Dieser Polynomzeitalgorithmus basiert auf 2-SAT-Programmierung.

Abstract

This thesis presents faster (in terms of worst-case running times) exact algorithms for special cases of graph problems through dynamic programming and 2-SAT programming. Dynamic programming describes the procedure of breaking down a problem recursively into overlapping subproblems, that is, subproblems with common subsubproblems. Given optimal solutions to these subproblems, the dynamic program then combines them into an optimal solution for the original problem. 2-SAT programming refers to the procedure of reducing a problem to a set of 2-SAT formulas, that is, Boolean formulas in conjunctive normal form in which each clause contains at most two literals. Computing whether such a formula is satisfiable (and computing a satisfying truth assignment, if one exists) takes linear time in the formula length. Hence, when satisfying truth assignments to some 2-SAT formulas correspond to a solution of the original problem and all formulas can be computed efficiently, that is, in polynomial time in the input size of the original problem, then the original problem can be solved in polynomial time. We next describe our main results.

Diameter asks for the maximal distance between any two vertices in a given undirected graph. It is arguably among the most fundamental graph parameters. We provide both positive and negative parameterized results for *distance-from-triviality*-type parameters and parameter combinations that were observed to be small in real-world applications.

In Length-Bounded Cut, we search for a bounded-size set of edges that intersects all paths between two given vertices of at most some given length. We confirm a conjecture from the literature by providing a polynomial-time algorithm for proper interval graphs which is based on dynamic programming.

k-Disjoint Shortest Paths is the problem of finding (vertex-)disjoint paths between given vertex terminals such that each of these paths is a shortest path between the respective terminals. Its complexity for constant $k \geq 3$ has been an open problem for over 20 years. Using dynamic programming, we show that k-Disjoint Shortest Paths can be solved in polynomial time for each constant k.

The problem Tree Containment asks whether a phylogenetic tree T is contained in a phylogenetic network N. A phylogenetic network (or tree) is a leaf-labeled single-source directed acyclic graph (or tree) in which each vertex has in-degree at most one or out-degree at most one. The problem stems from computational biology in the context of the *tree of life* (the history of speciation). We introduce a particular variant that resembles certain types of uncertainty in

the input. We show that if each leaf label occurs at most twice in a phylogenetic tree N, then the problem can be solved in polynomial time and if labels can occur up to three times, then the problem becomes *NP*-hard.

Lastly, Reachable Object is the problem of deciding whether there is a sequence of rational trades of objects among agents such that a given agent can obtain a certain object. A rational trade is a swap of objects between two agents where both agents profit from the swap, that is, they receive objects they prefer over the objects they trade away. This problem can be seen as a natural generalization of the well-known and well-studied Housing Market problem where the agents are arranged in a graph and only neighboring agents can trade objects. We prove a dichotomy result that states that the problem is polynomial-time solvable if each agent prefers at most two objects over its initially held object and it is *NP*-hard if each agent prefers at most three objects over its initially held object. We also provide a polynomial-time 2-SAT program for the case where the graph of agents is a cycle.

Preface

This thesis contains some of the results of my research at the Technische Universität Berlin in the Algorithmics and Computational Complexity group headed by Prof. Rolf Niedermeier from January 2017 to September 2020. The presented findings are partially based on published papers and partially based on papers that are only available on the arXiv repository yet. Many of these results were prepared in close collaboration with my coauthors. These are (in alphabetical order) Jiehua Chen, Vincent Froese, Klaus Heeger, Dušan Knop, Josef Malík, André Nichterlein, Malte Renken, Mathias Weller, Gerhard J. Woeginger, and Philipp Zschoche.

In the following, I sketch the story behind the research projects corresponding to the different chapters as well as briefly state my respective contributions.

Chapter 3. After finishing my master's thesis late 2016 in the young field of *FPT in P* and starting my PhD program in 2017, André Nichterlein (TU Berlin) suggested to further explore this field. He asked me to choose between either Diameter or Maximum Flow to work on next and I chose Diameter. Most of the results featured in our conference paper ([BN19]), which I presented at the *11th International Conference on Algorithms and Complexity (CIAC '19) in Rome, Italy*, are based on my ideas and André Nichterlein helped polishing both the results and the paper as a whole. An extended version featuring more details and all proofs is available in the arXiv repository and is submitted to a journal.

Chapter 4. From September 2018 to September 2019 Dušan Knop (Czech Technical University in Prague) had a postdoctoral position in our group. He suggested to study the problem Length-Bounded Cut. Initially, he was interested in certain *W[1]*-hardness results and started working on it with Klaus Heeger (TU Berlin). I joined the project soon after. During our research we found that the computational complexity of solving Length-Bounded Cut on interval graphs and proper interval graphs was stated as an open problem in the literature. We showed that the problem is polynomial-time solvable on proper interval graphs and we also proved the W[1]-hardness results that we were initially looking for, that is, for the feedback vertex number and the combined parameter pathwidth plus maximum degree. The polynomial-time algorithm was mostly my contribution while the W[1]-hardness results are mostly due to Klaus

Heeger. The corresponding paper ([BHK20]) was presented by Klaus Heeger at the *31st International Symposium on Algorithms and Computation (ISAAC '20)*, which was held virtually in December 2020. An extended version is available in the arXiv repository and is submitted to a journal.

Chapter 5. Our group holds a research retreat each year. In September 2019 at the retreat in Schloss Neuhausen (Brandenburg, Germany), André Nichterlein suggested to study a problem variant of Disjoint Paths and Anne-Sophie Himmel, Malte Renken, André Nichterlein, Philipp Zschoche (all TU Berlin), and I started working on it there. During the retreat, we studied different versions of Disjoint Paths and decided that we wanted to tackle the version Disjoint Shortest Paths after the retreat. It was known from the literature that this problem is *NP*-hard when the number k of shortest paths in the solution is part of the input and it was posed as an open problem for over twenty years whether there exists a polynomial-time algorithm for constant values of k. For $k = 2$ an $O(n^8)$-time algorithm was known, where n is the number of vertices in the input graph. Between December 2019 and January 2020, William Lochet (University of Bergen) and we independently answered the open question in the affirmative. William Lochet was the first to publish his paper at the 32nd *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '21)* [Loc21]. While his algorithm has a running time of $n^{O(k^{5^k})}$, where the Landau notation hides a constant 9^{55} in the exponent, we have since worked on improving the running time of our algorithm to $O(k \cdot n^{16k \cdot k! + k + 1})$. We also proved *W[1]*-hardness for Disjoint Shortest Paths with respect to k. All coauthors except for Anne-Sophie Himmel, who left academia shortly after the retreat and has withdrawn her authorship of the corresponding paper, have worked roughly equally on all parts of the paper. I was less involved in the *W[1]*-hardness result and instead designed a dynamic program for Disjoint Shortest Paths on directed acyclic graphs which is used as a subroutine in our main algorithm. I presented the corresponding paper at the 48th *International Colloquium on Automata, Languages, and Programming (ICALP '21)* [Ben+21]. An extended version of the paper is available in the arXiv repository.

Chapter 6. At the retreat in April 2017 near Boiensdorf (Mecklenburg-Vorpommern, Germany) Mathias Weller (University of Paris-Est) presented a problem called Tree Containment that stems from computational biology. Josef Malík (Czech Technical University in Prague), Mathias Weller, and I started working on it. Unfortunately, we had only limited success during the retreat but Mathias Weller and I wanted to continue working on it after the retreat. Josef Malík also wanted to participate further but did not have the required time to do so. For this reason, most of the results were achieved in equal parts by Mathias Weller and me in close collaboration. I presented the corresponding paper ([BMW18]) at the *16th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT '18)* in June 2018 in Malmö, Sweden. An extended version is available in the HAL repository and is accepted for publication in the *Journal of Graph Algorithms and Applications*.

Chapter 7. Rolf Niedermeier presented a paper on Reachable Object at the retreat in Darlingerode (Saxony-Anhalt, Germany) in March 2018. Jiehua Chen (TU Vienna), Vincent Froese (TU Berlin), Gerhard J. Woeginger (RWTH Aachen University), and I chose this problem to work on during the retreat. We achieved a few hardness results as well as a polynomial-time algorithm for short preference lists of all agents in close collaboration during the retreat. However, there was an intriguing open problem left when the input graph is a path that was described in the literature to be "at the frontier of tractability, despite its simplicity". Later this year, I resolved this case by providing a polynomial-time algorithm. A very similar algorithm was in the meantime developed independently by Sen Huang and Mingyu Xiao. We contacted the authors and invited them to join the two papers but they declined because of Chinese regulations. Their paper was presented at the *33rd AAAI Conference on Artificial Intelligence (AAAI '19)* and is published in *Autonomous Agents and Multi-Agent Systems* [HX20]. We since improved our algorithm to also work for cycles but so far the paper is only available in the arXiv repository [Ben+19a].

Acknowledgement. I am grateful to Rolf Niedermeier for giving me the opportunity to pursue my PhD in his group, for the many valuable lessons he taught me, and for the guidance he provided.

I am thankful to all my former and current colleagues and co-authors for countless hours of discussions and for providing an atmosphere in which working rarely felt like working: René van Bevern, Niclas Boehmer, Robert Bredereck, Markus Brill, Jiehua Chen, Alexander Dittmann, Till Fluschnik, Vincent Froese, Anne-Marie George, Roman Haag, Klaus Heeger, Christian Hofer, Anne-Sophie Himmel, Jonas Israel, Andrzej Kaczmarczyk, Leon Kellerhals, Dušan Knop, Tomohiro Koana, Junjie Luo, Josef Malík, Marcello G. Millani, Hendrik

Molter, Marco Morik, Luis Müller, André Nichterlein, Rolf Niedermeier, Malte Renken, René Saitenmacher, Ulrike Schmidt-Kraepelin, Piotr Skowron, Manuel Sorge, Christline Thielcke, Mathias Weller, Gerhard J. Woeginger, and Philipp Zschoche.

Finally, I want to express my sincere gratitude towards my family, my friends, and my choirs who gave me the strength to endure the few bad days and joy and companionship on so many other days.

Table of Contents

Chapter 1

Introduction

When confronted with a new problem, one of the first choices we face is to select a set of tools to tackle the problem with. Sometimes, none of the tools we know is useful for the task and we give up or we come up with a new (specialized) tool. Most often, however, (some of) the tools we already know are useful and our task becomes much easier once we figured out the correct tool for the job. So how do we choose the correct tool? Do we need to try every possible tool? Of course not. Is it up to experience to decide for the correct tool? While experience definitely helps, there are oftentimes rules or heuristics we can follow that guide us to the correct tool. Finding these rules and heuristics is important as it helps us and others to save time and effort not trying the wrong tools and not needing to collect years of experience before becoming efficient problem solvers.

While the above holds in general, we want to focus on algorithmic problems and exact algorithms in this thesis. The tools available for such tasks are numerous. When considering computationally easy problems (problems in P), we often start with *greedy* algorithms but also tools like *divide and conquer*, *dynamic programming*, or *modeling with a flow network* come to mind. When considering computationally hard (*NP*-hard) problems, then we can use some of the previous tools or we can refer to tools like *branch and bound*, *backtracking*, *integer linear programming (ILP)*, *modeling as a SAT problem*, *color-coding*, or *data reduction*. All of the mentioned tools are very well understood and we know at least some rules for each of them of when to apply them. Divide and conquer and dynamic programming for example are the first choices when a problem can be decomposed into smaller instances of the same problem. This is not to say, however, that we know *everything* about these tools already. In this thesis, we investigate two tools in more depth. These are *dynamic programming* and *2-SAT programming*. 2-SAT programming is a tool that has been used in the literature much less than dynamic programming. For dynamic programming we will not find additional rules for *when* to use it but rather some rules of *how* to apply it. For 2-SAT programming, we will investigate where and how it was used so far, develop our own experiences by applying it to two problems, and conclude with two heuristics of when 2-SAT programming might be a good fit.

One might ask why we chose exactly these two tools. On the one hand, this is to a certain degree up to pure chance. These tools just happened to work well for the problems we studied in the past. On the other hand, since we worked with these tools quite successfully, we feel confident that we can add something to the topic.

In the following, we give an introduction to *dynamic programming* and *2-SAT programming*. We conclude this chapter with an overview over the results in this thesis.

1.1 Dynamic Programming

Dynamic programming describes the procedure of recursively breaking down a problem into smaller *overlapping* subproblems and computing an optimal solution from solutions for these subproblems. Subproblems overlap if they have common subsubproblems. The analogous technique for non-overlapping subproblems is called *divide and conquer*. An example for divide and conquer is merge-sort, where in each step the array of numbers to sort is partitioned and independently sorted.

The term dynamic programming was coined around 1952 by Richard Bellman [Bel52]. Dynamic programming has since then become a staple of computer science which is taught in countless classes and books on algorithms, applied to computational problems such as Longest Common Subsequence, Longest Increasing Subsequence, Maximum Weight Independent Set on trees, and Approximate String Matching [Cor+09, Ski20]. It has been used in numerous fields including machine learning [BNK20, BSW89], computer vision [AWJ90], computational biology [Che+01, FT97, San00], and computational chemistry [Ari00, Gro+19]. It also had a large impact on parameterized algorithmics as *the* go-to tool for algorithms on tree decompositions of graphs [Bod88, LZ20, Mar20].

In the following, we will first give a general structure of how to apply dynamic programming. We then exercise a standard example for dynamic programming using our general structure and finally describe how this structure will guide us throughout the first part of this thesis. Dynamic programming is most often achieved by filling a table where each entry stores the solution to some subproblem. There are four main questions that one should answer when developing a dynamic program:

1. What does a table entry represent?
2. What dimension shall the table have?
3. How to compute the table entries?
4. How can the solution of the original problem can be computed once the table is completely filled?

We present a standard dynamic program for the problem Knapsack. For this problem, we are given a set X of n objects each with a positive integer weight (denoted by w) and a positive integer value (denoted by v) and two integers B and k. The question then is whether there is a subset of objects whose total weight is at most B and whose total value is at least k. Knapsack is known to be *NP*-complete but it allows for a pseudo-polynomial-time (polynomial if all number are encoded in unary) algorithm [TM90].

Without loss of generality, let $X = \{o_1, o_2, \ldots, o_n\}$. By answering the four questions above one by one, we explain how an existing $O(n \cdot B \cdot k)$-time dynamic program for Knapsack works [TM90].

1. What does a table entry represent?

Each entry in the table T represents a subproblem which is defined by a subset of objects and two bounds $1 \leq B' \leq B$ and $1 \leq k' \leq k$. The value in each entry in T is a binary value storing whether the respective subproblem is a yes- or a no-instance.

2. What dimension shall the table have?

Let $X_i \coloneqq \{o_1, o_2, \ldots, o_i\}$ be the set of the first i objects. The table T has entries for each subset X_i, where $i \in [n]$ and $[n] \coloneqq \{1, 2, \ldots, n\}$. Moreover, it has an entry for each combination of a subset X_i and values $1 \leq B' \leq B$ and $1 \leq k' \leq k$. The dimension or type of T is therefore

$$T\colon [n] \times [B] \times [k] \to \{\text{true}, \text{false}\}.$$

3. How to compute each table entry?

Initially, we set

$$T[1, B', k'] := \begin{cases} \text{true}, & \text{if } B' \geq w(o_1) \text{ and } k' \leq v(o_1), \text{ and} \\ \text{false}, & \text{otherwise.} \end{cases}$$

Once all entries $T[i, B', k']$ for a specific i are computed, we can compute an entry $t := T[i+1, B', k']$. We do so by distinguishing between the three cases $B' < w(o_{i+1})$, $B' = w(o_{i+1})$, and $B' > w(o_{i+1})$. If $B' < w(o_{i+1})$, then

$$t := T[i, B', k'].$$

If $B' = w(o_{i+1})$, then

$$t := \begin{cases} \text{true}, & \text{if } k' \leq v(o_{i+1}) \text{ and} \\ T[i, B', k'], & \text{otherwise.} \end{cases}$$

Finally, if $B' > w(o_{i+1})$, then

$$t := \begin{cases} \text{true}, & \text{if } k' \leq v(o_{i+1}) \text{ and} \\ T[i, B', k'] \vee T[i, B' - w(o_{i+1}), k' - v(o_{i+1})], & \text{otherwise.} \end{cases}$$

The idea is the following. If there is already a solution of total weight at most B' and total value at least k' using only the first i objects, then this solution is also a solution for the instance corresponding to $T[i+1, B', k']$. If no such solution exists, then the "new" object o_{i+1} has to be part of every solution. If $B' < w(o_{i+1})$, then no solution exists in this case. If $B' = w(o_{i+1})$, then there is a solution (the set $\{o_{i+1}\}$) if and only if $k' \leq v(o_{i+1})$. If $B' > w(o_{i+1})$, then either the set $\{o_{i+1}\}$ is a solution (if $k' \leq v(o_{i+1})$) or any solution set S contains o_{i+1} and $S' := S \setminus \{o_{i+1}\} \neq \emptyset$ (if $k' > v(o_{i+1})$) such that S' is a solution for the problem corresponding to $T[i, B' - w(o_{i+1}), k' - v(o_{i+1})]$.

4. How can the solution of the original problem be computed once the table is completely filled?

If each table entry is computed correctly, then the original instance is by definition a yes-instance if and only if $T[n, B, k] = \text{true}$.

We skip the formal proof of correctness and the analysis of the running time [TM90].

The first part of this thesis is about dynamic programming. Therein, we study the problems Diameter, Length-Bounded Cut, and k-Disjoint Shortest Paths. The Diameter problem asks for the longest shortest path between two vertices in a given graph. In Length-Bounded Cut, we are given an undirected graph, two terminal vertices s and t, and two integers k and ℓ. The question is whether there is a set of at most k edges such that removing those edges yields a graph in which the distance between s and t is larger than ℓ. For the problem k-Disjoint Shortest Paths, we are given an undirected graph and k terminal pairs (s_i, t_i) and the question is whether there are k disjoint paths P_i such that P_i is a shortest path between s_i and t_i.

These problems are in some sense very similar as all of the problems deal with shortest paths in a given undirected graph but are also quite different from one another: One the one hand, Diameter is polynomial-time solvable while Length-Bounded Cut and k-Disjoint Shortest Paths are *NP*-hard. On the other hand, Length-Bounded Cut is about removing (cutting) parts from the graph while Diameter and k-Disjoint Shortest Paths are more about routing (finding specific shortest paths in a graph). As equal and yet different the problems are, so are the algorithms we develop for each of them. The algorithms have in common that they are dynamic programs but they differ in which of our four guiding questions is hardest to answer for them. These respective questions therefore deserve additional consideration and these considerations will guide us through the first part of the thesis. In Chapter 3, we will study Diameter and we will encounter a dynamic program in which the dimension of the table is quite unique as it partially depends on the optimal solution and can therefore not be determined a priori. In Chapter 4, we study Length-Bounded Cut. The dynamic program we develop there does not allow to lookup the final answer in a specific table entry. Instead, the final answer is computed by iterating over a few specific table entries. Finally, in Chapter 5 we study k-Disjoint Shortest Paths and develop a dynamic program for it. The question of how to compute each table entry seems very easy to answer at first glance but it will turn out that considering it some more and ignoring some information given to us allows for a much faster algorithm.

1.2 2-SAT Programming

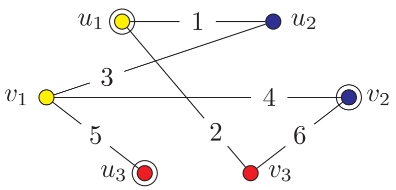
2-SAT programming[1] refers to the procedure of efficiently reducing a problem to a set of Boolean formulas in 2-CNF (conjunctive normal form with at most two literals per clause) such that the solution for the original problem can be constructed from the solutions for the 2-SAT formulas (satisfying truth assignments or the fact that formulas are unsatisfiable). The technique has been used in a wide range of contexts, e. g. subgraph detection [HL00, Jan17], graph transformation [HHW03], matrix partiotioning [Bul+16], computational biology [EHK03, GW09], resource allocation [HX20, MB20], and cartography [WW95]. However, we could not find many more examples of it being used and, to the best of our knowledge, 2-SAT programming has never been systematically analyzed as a general technique to solve computational problems. We start such an analysis by comparing how and when this technique was used in the literature so far. Before we do so, we first begin with an example of how to use 2-SAT programming.

We show how to solve the following special case of Independent Set in linear time in the input size.

Boolean Multicolored Independent Set
Input: An undirected graph $G := (V, E)$ where each vertex has a color and there are exactly two vertices of each color.
Question: Is there a colorful independent set in G, that is, is there a set that contains exactly one vertex of each color and no two vertices in this set share an edge in G?

Let $G = (V, E)$ be an instance of Boolean Multicolored Independent Set where u_i and v_i are the two vertices of the i^{th} color in G. Our constructed 2-SAT program consists only of a single formula Φ which contains a variable x_i for each color. Setting x_i to true corresponds to picking u_i into the solution and setting x_i to false corresponds to picking v_i into the solution. The formula Φ consist of one clause for each edge $\{y, z\}$ in G that evaluates to false if both y and z are picked into the solution. Let y have the i^{th} color and let z have the j^{th} color and let $i \neq j$ without loss of generality. We distinguish between the four possible cases (i) $y = u_i$ and $z = u_j$, (ii) $y = u_i$ and $z = v_j$, (iii) $y = v_i$ and $z = u_j$, and (iv) $y = v_i$ and $z = v_j$. In the first case, the clause shall evaluate to false if and only if $x_i =$ true and $x_j =$ true. This is achieved by the clause $\neg(x_i \wedge x_j) \equiv (\neg x_i \vee \neg x_j)$. Analogously, the clauses for the other three

[1] We mention that this method of problem solving has been used only a few times in the literature before. The name *2-SAT programming* is not established in the literature.



$$\begin{aligned}\Phi = {} & (\neg\, x_1 \vee \neg\, x_2) \wedge (\neg\, x_1 \vee x_3) \wedge \\ & (x_1 \vee \neg\, x_2) \wedge (x_1 \vee x_2) \wedge \\ & (x_1 \vee \neg\, x_3) \wedge (x_2 \vee x_3)\end{aligned}$$

Figure 1.1: An example instance of Boolean Multicolored Independent Set and the 2-SAT formula Φ constructed by our 2-SAT program. The encircled vertices form a solution and the edges are enumerated to allow easier verification of Φ. The first edge corresponds to the first clause in Φ, the second edge to the second clause in Φ, and so on. Note that the encircled vertices correspond to the truth assignment $x_1 =$ true, $x_2 =$ false, and $x_3 =$ true. It is easy to verify that this is a satisfying truth assignment to Φ. The encircled solution indeed is the only solution for the given instance and the described truth assignment is the only satisfying truth assignment for Φ.

cases are $(\neg\, x_i \vee x_j)$, $(x_i \vee \neg\, x_j)$, and $(x_i \vee x_j)$, respectively. An example of this construction is given in Figure 1.1. Note that the 2-SAT formula Φ can be computed in time linear in the size of G. Since Φ can be checked for a satisfying truth assignment in linear time (in the length of Φ which is linear in the size of G) [APT79], the total running time is linear in the input size. It is easy to verify that Φ is satisfied by some truth assignment if and only if the set of vertices corresponding to this truth assignment are pairwise non-adjacent in G. Since each such set contains exactly one vertex of each color, each solution of Φ corresponds to a solution of Boolean Multicolored Independent Set.

We conclude this introduction to 2-SAT programming with an analysis of how and when 2-SAT programming was used in the literature before and how our two new results fit into this picture. The majority of results that we could find that used 2-SAT programming ([Bul+16, EHK03, GW09, HHW03, HL00, WW95]) used it as follows. Variables describe whether or not to pick some element into a solution set and the clauses in each 2-SAT formula prevented that some conflicting elements where chosen in the same solution. In the remaining examples ([HX20, Jan17, MB20]) variables did not represent whether or not to pick some element into a solution but rather which element is picked into a solution (as in our example above). By exploring 2-SAT programming more in-depth in the thesis, we hope to find some indications of when 2-SAT programming should be considered for new (algorithmic) problems. Indeed, we will conclude that 2-SAT programming is promising when the considered problem is (thought to be) polynomial-time solvable and has some *independence structure*. By that, we mean that the a solution consists of some elements that can

- be partitioned into constant-size parts and at most one element from each part is picked into the solution, and

- a set of elements forms a solution if each pair of elements in this set can be contained in the same solution.

In the example above, the elements were the vertices of the graph, the partition was achieved by the colors of vertices, and a set of vertices forms a solution only if they are pairwise non-adjacent.

We present two new examples of 2-SAT programming in the second part of the thesis. These examples follow the distinction of 2-SAT programs stated above. In Chapter 6, we study a problem from computational biology called Tree Containment that asks whether a specific subtree exists in a given directed graph. Roughly speaking, our approach is to introduce a variable for each vertex in the input network that is set to true if the vertex belongs to the sought subtree and to false otherwise. In Chapter 7, we investigate Reachable Object, a problem stemming from the field of resource allocation. Therein, agents initially own objects, have different preferences over the objects, and are arranged in a social network. They may swap objects with one another under certain conditions including that they must be adjacent in the social network. The question is then whether a specific agent can obtain a given target object. In a cycle, each object is given from the agent that initially holds it to one of its two possible neighbors. The variables in the 2-SAT program we develop in Chapter 7 then represent for each object to which of the two respective agents it is swapped to.

We conclude this introduction to 2-SAT programming with describing a similarity and a dissimilarity between the two 2-SAT programs we develop in the thesis. They have in common that they are designed for very sparse graphs (trees in Chapter 6 and cycles in Chapter 7) and they differ in how they generalize to "k-SAT programs". While the algorithm for Tree Containment does generalize naturally to any constant k to a correct algorithm for a meaningful problem, the same cannot be said about the algorithm for Reachable Object.

1.3 Results

In this thesis, we design and analyze algorithms for (mostly *NP*-hard) graph problems. We achieve a wide range of different results, among others, parameterized hardness and algorithms, polynomial-time algorithms for special cases, and results within *FPT in P*, that is, parameterized algorithms and hardness results for problems in *P* [GMN17]. We also resolve some open problems from the literature.

In Chapter 3, we study the DIAMETER problem, which asks for the longest shortest path between any two vertices in a given graph. This parameter was observed to be very small in many different real-world application [LH08, Mil67, New03] and it is often used in network analysis [AJB99, WF94]. This has led to a wide spectrum of algorithms computing the diameter faster than the naïve algorithm (see Zwick [Zwi01]). We add to this spectrum by providing new parameterized algorithms for computing the diameter. On the one hand, we study *distance-from-triviality*-like parameters [GHN04] and show that graphs with small modulators to cographs, that is, small sets of vertices whose removal yield a cograph, allow for faster diameter computations while graphs with small modulators to bipartite graphs do not. On the other hand, we study parameter combinations that are expected to be small in real-world applications. Here, we show that the combined parameter h-index plus diameter allows for positive *FPT-in-P* results whilst similar combinations under standard complexity assumptions do not. The algorithms for graphs with small modulators to cographs and for the combined parameter h-index plus diameter are both based on dynamic programming.

In Chapter 4, we study the problem LENGTH-BOUNDED CUT which arises from the field of network flows. Given an undirected graph, two terminal vertices s and t, and two integers k and ℓ, the question is whether there is a set of at most k edges such that removing these edges yields a graph in which the distance between s and t is larger than ℓ. We prove a conjecture by Bazgan et al. [Baz+19] by providing a polynomial-time algorithm for LENGTH-BOUNDED CUT on proper interval graphs which is based on dynamic programming. We also briefly investigate interval graphs and show limitations of our approach for proper interval graphs.

In Chapter 5, we look at a long-standing open question regarding the complexity of k-DISJOINT SHORTEST PATHS for constant k [Eil98, Fom+19]. In k-DISJOINT SHORTEST PATHS, we are given an undirected graph and k terminal pairs (s_i, t_i), and the question is whether there are k disjoint paths P_i such that P_i is a shortest path between s_i and t_i. We present an algorithm whose running time is polynomial in the input size for each constant k. The algorithm is based on dynamic programming and a geometric representation of the problem that is quite intuitive yet, to the best of our knowledge, novel.

In Chapter 6, we investigate a problem variant of TREE CONTAINMENT which stems from computational biology. Given a leaf-labeled directed acyclic graph N (called a phylogenetic network) and a leaf-labeled directed tree T, the question in TREE CONTAINMENT is whether N *displays* T. This is the case if N contains a subdivision of T as a subgraph that respects leaf-labels [ISS10]. A version of TREE CONTAINMENT where N is a tree is used in the quest for finding the "tree of life", that is, given the current knowledge of speciation (modeled as a directed tree N) and some new data (modeled as another (possibly smaller) directed tree T), the question is whether N and T are consistent. We call the version we investigate SOFT TREE CONTAINMENT. It is motivated by soft polytomies, that is, multiple speciation events whose order is unknown. Another kind of uncertainty can be modeled by allowing N to have multiple leaves with the same label. Our main contribution is a dichotomy result regarding the maximal number of occurrences of a label in N. On the one hand, using 2-SAT programming, we show that SOFT TREE CONTAINMENT is polynomial-time solvable if N is a tree in which each leaf-label occurs at most twice. On the other hand, we show that SOFT TREE CONTAINMENT remains *NP*-hard when restricted to trees in which each leaf-label occurs at most thrice.

In Chapter 7, we study a problem called REACHABLE OBJECT. Therein, one is given a set of agents, a set of objects, a specific agent I, and a specific object x. Each agent has strict preferences over the objects and initially owns exactly one object. Additionally, the agents are arranged in a graph (social network) representing which agents know each other. The question is then whether there is a sequence of *rational swaps* such that agent I owns object x in the end [GLW17]. A rational swap is a trade between two agents that know each other such that both agents receive an object they prefer over the object they give away. Our contribution is twofold. First, we present a dichotomy result regarding the number of objects each agent prefers over its initially held object. If each agent prefers at most two objects over the one it initially holds, then REACHABLE OBJECT can be solved in polynomial using dynamic programming. The problem remains *NP*-hard even if each agent prefers at most three objects over its initially held object. Second, using 2-SAT programming, we provide a polynomial-time algorithm for REACHABLE OBJECT on cycles which is a generalization of a previous algorithm for REACHABLE OBJECT on paths [HX20]. The original algorithm for paths answered an open problem from the literature [GLW17].

Finally, we summarize our main results and give a broader overview over possible avenues for further research regarding dynamic programming and 2-SAT programming in Chapter 8.

Chapter 2

Preliminaries

In this chapter, we describe our notation and some general tools that will be used in the following chapters. If a specific notion is only used in a single chapter, then it will be introduced there. We assume familiarity with the basics of set theory, calculus, and the description and analysis of algorithms.

2.1 Number Theory

We denote by $\mathbb{Z} := \{\ldots, -1, 0, 1, \ldots\}$ the set of all integers, by $\mathbb{N} := \{0, 1, 2, \ldots, \}$ the set of all non-negative integers, and by $\mathbb{N}^+ := \mathbb{N} \setminus \{0\}$ the set of all positive integers. We use $\mathbb{Q} := \{{}^p/_q \mid p \in \mathbb{Z} \wedge q \in \mathbb{N}^+\}$ to denote the set of all rational numbers and $\mathbb{Q}_0^+ := \{{}^p/_q \mid p, q \in \mathbb{N}^+\}$ to denote the set of all positive rational numbers. The set of all real numbers is denoted by $\mathbb{R}$.

For two integers $a, b \in \mathbb{Z}$, we denote by $[a, b]$ the integer *interval* between a and b, that is, $[a, b] := \{i \in \mathbb{Z} \mid a \leq i \leq b\}$. Analogously, we denote the rational *interval* between a and b by $[a, b]^{\mathbb{Q}} := \{i \in \mathbb{Q} \mid a \leq i \leq b\}$. For $a > b$, let $[a, b] := [a, b]^{\mathbb{Q}} := \emptyset$. Finally, for a positive integer $\ell \in \mathbb{N}^+$ we use $[\ell]$ as an abbreviation for $[1, \ell] = \{1, 2, \ldots, \ell\}$.

2.2 Graph Theory

An undirected graph G is a tuple (V, E) where V is the set of vertices or nodes and $E \subseteq \binom{V}{2}$ is the set of edges. A directed graph is a tuple (V, A) where V is again the set of vertices or nodes and $A \subseteq \{(u, v) \mid u \neq v \wedge u, v \in V\}$ is the set of arcs. We will use $n := |V|$ to denote the number of vertices, $m := |E|$ ($m := |A|$)

to denote the number of edges or arcs, and $|G| := n + m$ to denote the *size* of G. All graphs in this thesis are undirected unless explicitly stated otherwise.

For a vertex subset $V' \subseteq V$, we denote by $G[V']$ the graph induced by V', that is, the graph $G[V'] := (V', E' := \{\{u, v\} \in E \mid u, v \in V'\})$ if G is an undirected graph and $G[V'] := (V', A' := \{(u, v) \in A \mid u, v \in V'\})$ if G is directed. We abbreviate $G - V' := G[V \setminus V']$. A path $P := (v_0, \dots, v_\ell)$ in a directed graph is a graph with a set $V(P) := \{v_0, \dots, v_\ell\}$ of vertices and arc set $A(P) := \{(v_i, v_{i+1}) \mid 0 \leq i < \ell\}$. A path $P := (v_0, \dots, v_\ell)$ in an undirected graph is a graph with a set $V(P) := \{v_0, \dots, v_\ell\}$ of vertices and a set $E(P) := \{\{v_i, v_{i+1}\} \mid 0 \leq i < \ell\}$ of edges. We say that ℓ is the *length* of P and a *shortest path* between two vertices is a path of minimum length. We define $A(P)$ to be the set of arcs $\{(v_{i-1}, v_i) \mid i \in [\ell]\}$ and $A^{-1}(P)$ to be the set of arcs $\{(v_i, v_{i-1}) \mid i \in [\ell]\}$. Intuitively, $A(P)$ and $A^{-1}(P)$ describe the two directed versions of P in an undirected graph. The vertices v_0 and v_ℓ are called the *end vertices* or *ends* of P and are denoted by $\operatorname{start}(P)$ and $\operatorname{end}(P)$. We also say that P is a path *from* v_0 *to* v_ℓ, a path *between* v_0 and v_ℓ, or a v_0-v_ℓ-path. When no ambiguity arises, we do not distinguish between a path and its set of vertices. We identify specific paths by just some of their vertices, e. g. we use the name *a-b-c-path* to denote a path that starts in a, then continues by some shortest a-b-path, and ends with some shortest b-c-path.

Let v, w be two vertices in a path P. We denote by $P[v, w]$ the subpath of P with end vertices v and w. For two paths $P_1 := (v_0, \dots, v_a)$ and $P_2 := (v'_0, \dots, v'_b)$ with $v'_0 = v_a$ or $\{v_a, v'_0\} \in E$ ($(v_a, v'_0) \in A$), we define $P_1 \bullet P_2 := (v_0, \dots, v_a, v'_1, \dots, v'_b)$ or $P_1 \bullet P_2 := (v_0, \dots, v_a, v'_0, \dots, v'_b)$, respectively. For two vertices $u, v \in V$, we denote with $\operatorname{dist}_G(u, v)$ the distance between u and v in G, that is, the number of edges in a shortest path between u and v. If G is clear from the context, then we omit the subscript. A *connected component* $C \subseteq V$ in a graph G is a maximal set of vertices such that there is a path between each pair of vertices in C.

The *degree* $\deg_G(v)$ of a vertex $v \in V$ in an undirected graph G is the number of edges that contain v. The *in-degree* of a vertex $v \in V$ in a directed graph is the number of arcs of the form (u, v) for $u \in V$. A vertex with in-degree zero is called a *source*. The *out-degree* of a vertex $v \in V$ in a directed graph is the number of arcs of the form (v, w) for $w \in V$. A vertex with out-degree zero is called a *sink*. The degree of a vertex $v \in V$ in a directed graph is the sum of its in-degree and its out-degree. The *neighborhood* $N_G(v)$ of a vertex is the set of all vertices that share an edge (or arc) with v in G and we use $N_G[v]$ to denote $N(v) \cup \{v\}$. Again, if G is clear from the context, then we omit the subscript. *Suppressing* a degree-two vertex $v \in V$ in an undirected graph G refers to the action of removing the vertex v from G and adding the edge between v's two neighbors u, w if it is not already contained in G. Suppressing a vertex $v \in V$ in a directed graph $G = (V, A)$ with in-degree one and out-degree one refers to the action of removing the vertex v from G and adding the arc (u, w) where $(u, v), (v, w) \in A$. Again, if this arc was already present in A, then we just remove v with its two incident arcs. *Subdividing* an edge $\{u, w\}$ in an undirected graph refers to the action of removing $\{u, w\}$ and adding a new vertex v and new edges $\{u, v\}$ and $\{v, w\}$. Subdividing an arc (u, w) in a directed graph refers to the action of removing the respective arc and adding a new vertex v and new arcs (u, v) and (v, w).

We continue with some notation for *directed acyclic graphs (DAGs).* We call a vertex d in a DAG G a *descendant* of another vertex a if there is a a-d-path in G. Moreover, we call a an *ascendant* of d. For a DAG G, let $<_G$ be a relation between vertices in G such that $v <_G u$ if and only if u is an ancestor of v. Moreover, let $u \leq_G v$ if and only if $u <_G v$ or $u = v$. Let define G_v to be the subgraph of G induced by $\{u \mid u \leq_G v\}$. The set of *least common ancestors* $\mathrm{LCA}_N(\{X\})$ of a set X of vertices contains all minima with respect to $\leq_G$ among all vertices u of N with $v \leq_G u$ for all $v \in X$. In particular, if G is a tree, then $\mathrm{LCA}_N(\{X\})$ contains a single vertex. If G is clear from the context, then we may drop the subscript.

Two undirected graphs $G := (V_G, E_G)$ and $H := (V_H, E_H)$ are *isomorphic* if there is a bijection f between V_G and V_H such that for any two vertices $u, v \in V_G$ it holds that $\{u, v\} \in E_G$ if and only if $\{f(u), f(v)\} \in E_H$. Analogously, two directed graphs $G := (V_G, A_G)$ and $H := (V_H, A_H)$ are isomorphic if there is a bijection f between V_G and V_H such that for any two vertices $u, v \in V_G$ it holds that $(u, v) \in A_G$ if and only if $(f(u), f(v)) \in E_H$. We call f the *mapping* between G and H.

2.2.1 Graph Classes

A *tree* is a connected acyclic (directed or undirected) graph, that is, a graph in which each pair of vertices is connected by an unique shortest path. A *rooted tree* is a tree T with a designated vertex r called the *root* of T. The *depth* of a vertex v in a rooted tree is the distance between v and r. The *height* of a vertex v in a rooted tree is the maximum distance between v and a leaf ℓ in T such that v is contained in a shortest r-ℓ-path. A *forest* is a graph in which each connected component is a tree.

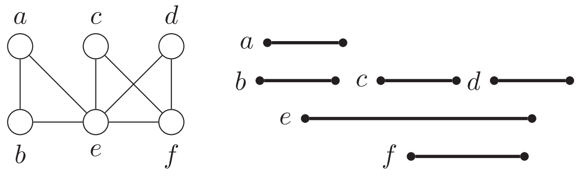


Figure 2.1: An example of an interval graph (left side) and its interval representation (right side).

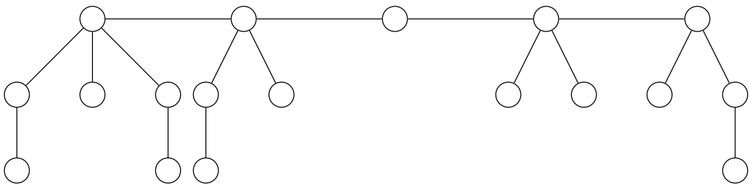


Figure 2.2: An example of a generalized caterpillar with hair length two. The topmost vertices form the central path and the paths below are the *hairs*.

A *clique* is a graph $G = (V, E)$ with $E = \{\{u, v\} \mid u, v \in V\}$. A graph is *bipartite* if its vertex set can be partitioned in two sets V_1, V_2 such that for each edge $\{u, v\} \in E$ it holds that $u \in V_1$ and $v \in V_2$ (or $u \in V_2$ and $v \in V_1$). Analogously, a graph is *k-partite* if its vertex set can be partitioned into k sets $V_1, V_2, \ldots, V_k$ such that it holds for each edge $\{u, v\} \in E$ that u and v are not contained in the same vertex set V_i.

An *interval graph* is a graph $G = (V, E)$ such that each vertex v can be represented by a rational interval $[b_v, f_v]^{\mathbb{Q}}$ such that two vertices u, w are adjacent in G if and only if $[b_u, f_u]^{\mathbb{Q}} \cap [b_w, f_w]^{\mathbb{Q}} \neq \emptyset$. A *proper interval graph* is an interval graph such that there are no two vertices v and w such that $[b_v, f_v]^{\mathbb{Q}} \subset [b_w, f_w]^{\mathbb{R}}$. Equivalently, a proper interval graph can be defined as an interval graph where each interval has length one, i. e., $b_v + 1 = f_v$ for each vertex v (see e. g. [BLS99]). An example of an interval graph and its interval representation is given in Figure 2.1. A *cograph* is a graph that does not contain a P_4 (a path of four vertices and three edges) as an induced subgraph. A *caterpillar* is a tree such that removing all leaves yields a path (i.e, all vertices are within distance at most one of a "central path"). A *generalized caterpillar* with hairs of length at most $h \geq 1$ is a tree such that removing paths of length at most h yields a path. A generalized caterpillar with hairs of length two is shown in Figure 2.2.

2.2.2 Graph Parameters

The *maximum degree* of a graph $G = (V, E)$ is the maximum number of incident edges to any single vertex in the graph, that is, $\max\{\deg(v) \mid v \in V\}$. Analogously, the *minimum degree* is defined as $\min\{\deg(v) \mid v \in V\}$ and the *average degree* of a graph is $^{2m}/_{n}$. We denote by $d(G)$ the *diameter* of G, that is, the length of the longest shortest path in G. The *h-index* of a graph G is the maximum number h such that the graph contains at least h vertices of degree at least h.

One of the most famous graph parameters is the *treewidth*. It is defined through *tree decompositions*. A tree decomposition of a graph $G = (V, E)$ is a tree $T = \{\mathcal{X}, E'\}$, where each $X_i \in \mathcal{X} = \{X_1, X_2, \ldots, X_\ell\}$ is a subset of V and the following three properties hold. First, each vertex $v \in V$ is contained in at least one $X_i \in \mathcal{X}$. Second, for each vertex $v \in V$, the set of all X_i with $v \in X_i$ induces a connected subgraph in T. Third, for every edge $\{u, v\} \in E$, there is a subset X_i that contains both u and v. The *width* of a tree decomposition is $\max\{|X| \mid X \in \mathcal{X}\} - 1$ and the treewidth of G is the minimum width among all possible tree decompositions of G. The *pathwidth* of a graph is defined similarly to its treewidth, but instead of tree decompositions only path decompositions are considered, that is, the tree T is required to be a path.

The *girth* of a graph is the size of a smallest induced cycle in the graph (or ∞ if the graph is a forest). The *bisection width* of a graph $G = (V, E)$ is defined as the size of a smallest set E' of edges such that V can be partitioned into two sets V_1, V_2 with $|V_1| = |V_2|$ (or $|V_1| = |V_2| + 1$ if $|V|$ is odd) such that all edges with one end in V_1 and one end in V_2 is contained in E'. Bisection width is illustrated in Figure 2.3. A *dominating set* in a graph is a set K of vertices such that each vertex in the graph is contained in K or has at least one neighbor in K. The *domination number* of a graph is the size of a minimum dominating set in it. The *acyclic chromatic number* of a graph is the minimum number of colors needed to color each vertex with one of the given colors such that each subgraph induced by all vertices of one color is an independent set and each subgraph induced by all vertices of two colors is acyclic.

Lastly, for some graph class Π, the *distance to* Π is the size of a minimum set of vertices such that the graph resulting from deleting this set of vertices is in Π. In this thesis, we will consider the *distance to cographs*, the *distance to bipartite graphs*, and the *distance to forests*. The distance to bipartite graphs is known as *odd cycle transversal number* and the distance to forests is known as *feedback vertex number* in the literature. We will hence use these names. The

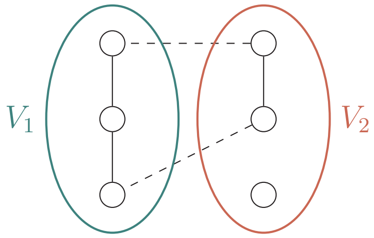


Figure 2.3: An example of the bisection width of a graph. The edges between the two parts are drawn using dashed lines and the bisection width is two (the number of dashed edges).

edge-deletion distance to forests, that is, the size of a smallest set of edges such that removing them yields a forest, is known as the *feedback edge number*.

2.3 Complexity Classes and Hypotheses

We assume familiarity with the basics of Turing machines and Random Access Machines. Otherwise, we refer to Papadimitriou [Pap94]. In this thesis, we will always analyze the running time of an algorithm in terms of Random Access Machines. However, complexity classes are classically defined using Turing machines.

The class P contains all decision problems (or languages) that can be decided in polynomial time by deterministic Turing machines. The class NP contains all decision problems that can be decided in polynomial time by non-deterministic Turing machines.

A *parameterization* for a problem L is formally a pair of functions (f, g) such that f maps each possible input I for P to some object $f(I)$ and g maps each such object to a non-negative integer. We use the *treewidth* of a graph as an example. Here, f maps each graph to a *tree decomposition* of G and g measures the *width* of the tree decomposition, that is, the maximum number of vertices in any bag of the tree decomposition (minus one). A *parameter* is then the resulting positive integer $g(f(I))$ of a parameterization. A *parameterized problem* is a tuple (L, κ), where L is a language (an unparameterized decision problem) and κ is a *parameter*. An instance of (L, κ) is a pair (x, k) where $k = g(f(x))$ for some parameterization (f, g). For a parameterized problem $\mathcal{L} = (L, \kappa)$, the language $\hat{\mathcal{L}} = \{x \in \Sigma^* \mid \exists k \colon (x, k) \in \mathcal{L}\}$ is called the *unparameterized problem* associated to $\mathcal{L}$. For a broader introduction into parameterized complexity theory, we refer the reader to the books by Cygan et al. [Cyg+15], Downey and Fellows [DF13], Flum and Grohe [FG06], and Niedermeier [Nie06].

A problem L is *fixed-parameter tractable* with respect to some parameter κ if there is an algorithm deciding whether $(x, k) \in (L, \kappa)$ (or equivalently $x \in L$) in $f(k) \cdot |x|^{O(1)}$ time, where $|x|$ denotes the size of x and f is some computable function depending only on k. The class *FPT* contains all parameterized problems (L, κ) where L is fixed-parameter tractable with respect to κ. The class *XP* contains all parameterized problems (L, κ) such that there is an algorithm deciding whether $(x, k) \in (L, \kappa)$ in $|x|^{f(k)}$ time, where f is again some computable function only depending on k. The class *W[1]* contains all parameterized problems (L, κ), where every instance (x, k) of (L, κ) can be transformed in $f(k) \cdot |x|^{O(1)}$ time to a combinatorial circuit that has weft at most one and constant depth for all instances, such that $(x, k) \in (L, \kappa)$ if and only if there is a satisfying truth assignment to the input circuit that assigns true to exactly k inputs. The *weft* of a combinatorial circuit is the largest number of logical units with unbounded fan-in on any path from an input to the output. The *depth* of a combinatorial circuit is the largest number of logical units on any path from an input to the output. Similarly to the assumption that $P \neq NP$, the assumption $FPT \neq W[1]$ is widely believed and is used to exclude *FPT*-results. A few years ago, the topic of *FPT in P* [GMN17] emerged from parameterized complexity theory. Therein, instead of designing $f(k) \cdot |x|^{O(1)}$-time algorithms for *NP*-hard problems where k is some superpolynomial function, one is interested in $f(k) \cdot |x|^c$-time algorithms for problems in P, where no $O(|x|^c)$-time algorithm is known for the unparameterized problem associated to it.

The problem k-SAT is a generalization of 2-SAT and defined as follows.

k-SAT
Input: A Boolean formula Φ in conjunctive normal form where each clause in Φ contains at most k literals.
Question: Is Φ satisfiable?

Analogously to 2-SAT programs, we use the term *k-SAT program* to refer to an algorithm that solves a problem by constructing and solving k-SAT instances (formulas conjunctive normal form where each clause contains at most k literals).

The Exponential-Time Hypothesis (ETH) of Impagliazzo and Paturi [IP01] postulates that there is no $2^{o(m)}$-time algorithm solving the Satisfiability problem, where m is the number of clauses. It is formalized as follows.

Hypothesis 2.1 (Exponential-Time Hypothesis (ETH))**.** *There is some constant $\delta > 0$ such that 3-SAT cannot be solved in $O(2^{\delta n})$ time, where n is the number of variables in the input formula.*

It is worth noting that assuming the ETH, there is no $f(k) \cdot n^{o(k)}$-time algorithm solving MULTICOLORED CLIQUE problem [Che+05], where f is a computable function and k is the solution size.

MULTICOLORED CLIQUE
Input: An integer k and a k-partite undirected graph $G := (V, E)$ with $V := \biguplus_{i=1}^{k} V_i$ and $|V_i| = n/k$ for all $i \in [k]$.
Question: Is there an induced clique of size at least k in G?

A stronger version of the ETH is the so-called Strong Exponential-Time Hypothesis (SETH) [IP01]. It states the following.

Hypothesis 2.2 (Strong Exponential-Time Hypothesis (SETH))**.** *For each $\delta < 1$ there is an integer k such that k-SAT cannot be solved in $O(2^{\delta n})$ time.*

Let Φ be a Boolean input formula for SATISFIABILITY. We remark that if the SETH is true, then there is no $|\Phi|^{2-\varepsilon} \cdot |\Phi|^{O(1)}$-time algorithm solving SATISFIABILITY [IP01].

2.4 Reductions Between Problems

A *(many-one) reduction* is a function $R\colon \Sigma^* \to \Sigma^*$ that transforms an instance x of some problem L to an equivalent instance y of a problem L', that is, $y \in L' \iff x \in L$. A *polynomial-time (many-one) reduction* is a reduction that can be computed in time polynomial in the input size $|x|$. To show that a problem A is presumably not in P, one can reduce an NP-hard problem B to A (written as $B \leq^p A$). Unless $P = NP$, this shows that $A \notin P$. A problem B is NP-hard if for all problems in NP there is a polynomial-time reduction to B. Famous examples of NP-hard problems are e. g. MULTICOLORED CLIQUE and k-SAT [Kar75]. If $B \leq^p A$ for a NP-hard problem B, then A is also NP-hard.

A *parameterized reduction* is a reduction $R\colon \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ that transforms a parameterized problem $\mathcal{L}$ to a parameterized problem $\mathcal{L}'$ in FPT-time, that is, for each instance (x, k) of $\mathcal{L}$ it produces an instance (y, ℓ) of $\mathcal{L}'$ such that

1. (y, ℓ) can be computed in $f(k) \cdot |x|^{O(1)}$ time for some computable function f,

2. $(y, \ell) \in \mathcal{L} \iff (x, k) \in \mathcal{L}'$, and
3. $\ell \leq g(k)$ for some computable function g.

To show that some parameterized problem is *presumably not* in *FPT*, one regularly uses the standard complexity assumption that *FPT* $\neq$ *W[1]* and shows that a problem is *W[1]*-hard. To show *W[1]*-hardness for some parameterized problem $\mathcal{L}$, we use parameterized reductions from *W[1]*-hard problems similar to the unparameterized setting. Probably the most famous example of a *W[1]*-hard problem is Multicolored Clique parameterized by the solution size k.

Concerning *FPT-in-P* studies, we use the notion of General-Problem-hardness which formalizes the types of reduction that allow us to exclude certain parameterized algorithms for problems in *P*. In a nutshell, we want to upper-bound the parameter in the constructed instance by some constant ℓ without increasing the running time or the instance size by too much. Since it holds for each computable function f that $f(\ell)$ is some constant, we can then hide any dependency on ℓ in the Landau notation.

Definition 2.1 ([Ben+19b, Definition 3.1])**.** Let $\mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem, let $\hat{\mathcal{L}} \subseteq \Sigma^*$ be the unparameterized decision problem associated to $\mathcal{L}$, and let $g\colon \mathbb{N} \to \mathbb{N}$ be a polynomial. We call $\mathcal{L}$ *ℓ-General-Problem-hard(g) (ℓ-GP-hard(g))* if there exists an algorithm $\mathcal{A}$ transforming any input instance x of $\hat{\mathcal{L}}$ into a new instance (y, k) of $\mathcal{L}$ such that

1. $\mathcal{A}$ runs in $O(g(|x|))$ time,
2. $(y, k) \in \mathcal{L} \iff x \in \hat{\mathcal{L}}$,
3. $k \leq \ell$, and
4. $|y| \in O(|x|)$.

We call $\mathcal{L}$ *General-Problem-hard(g) (GP-hard(g))* if there exists an integer ℓ such that $\mathcal{L}$ is ℓ-GP-hard(g). We omit the running time and call $\mathcal{L}$ *ℓ-General-Problem-hard (ℓ-GP-hard)* if g is a linear function.

Showing GP-hardness for some parameter κ allows to lift algorithms for the parameterized problem to the unparameterized setting as stated next. The idea behind this statement is that assuming a parameterized problem is both ℓ-GP-hard n^c and can be solved in $O(n^c \cdot f(k))$ time for some computable function f and some constant c, then we can solve the unparameterized problem associated to it in $O(n^c)$ time by first reducing an arbitrary input instance to an equivalent instance in which the parameter is at most ℓ and then use the parameterized algorithm where we can hide the dependency on the parameter in the Landau notation.

Lemma 2.3 ([Ben+19b, Lemma 3.2])**.** *Let $g\colon \mathbb{N} \to \mathbb{N}$ be a polynomial, and let $\mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem which is GP-hard(g). Let $\hat{\mathcal{L}} \subseteq \Sigma^*$ be the unparameterized decision problem associated to $\mathcal{L}$. If there is an algorithm solving each instance (y, k) of $\mathcal{L}$ in $f(k) \cdot g(|y|)$ time, then there is an algorithm solving each instance x of $\hat{\mathcal{L}}$ in $O(g(|x|))$ time.*

We conclude this chapter with a simple example to illustrate Lemma 2.3. Consider the problem of detecting whether a given undirected graph contains a clique of size five and the parameter bisection width. By simply copying the graph such that the resulting graph has twice as many vertices and edges, the bisection width becomes zero as there is no edge between the two copies of the original graph and both copies contain the same number of vertices. Moreover, the resulting graph contains a clique of size five if and only if the original graph contains a clique of size five. Now assume that there was an $O((n + m) \cdot f(k))$-time algorithm for this problem where k is the bisection width and f is some computable function. Then, this would imply that we could first construct an equivalent instance in linear time where $k = 0$ as described above, and then solve this equivalent problem in $f(k) \cdot (n + m) \in O(n + m)$ time.

Part I

Dynamic Programming

Chapter 3

Diameter

In this chapter, we study the problem Diameter which asks for the maximum distance between any two vertices in a given undirected graph. Regarding dynamic programming, this chapter features a dynamic program that is noteworthy as the solution for Diameter will not be related to any table entry but to the final size of the table. To the best of our knowledge, this is the first time the dimension of a dynamic program was used in this way.

Concerning the Diameter problem, many consider the diameter of a graph among the most fundamental graph parameters [Bac+18, New03, WF94]. Most known algorithms for determining the diameter first compute the shortest path between each pair of vertices (All-Pairs Shortest Paths) and then return the maximum [AVW16]. However, several more efficient algorithms have been proposed for special cases [AVW16, BHM20, Cor+01, FP80, Gaw+21] or for approximating the diameter [Ain+99, Bac+18, RW13, WY16].

In this chapter, we follow the *FPT-in-P* approach [AVW16, Ben+20, GMN17], that is, we propose parameterized algorithms for Diameter that run faster than known unparameterized algorithms when specific parameters are very small or show that such algorithms refute popular complexity assumptions. In Section 3.2, we follow the *distance-from-triviality-parameterization paradigm* [GHN04] aiming to augment a folklore algorithm for Diameter on cographs such that it also works for graphs with small modulators to cographs, that is, graphs with small sets of vertices whose removal yields a cograph. We also analyze graphs with small modulators to bipartite graphs. For the parameter distance k to cographs, we provide a $2^{O(k)}(n+m)$-time algorithm. For the parameter odd cycle transversal number k (the distance to bipartite graphs), we use our recently introduced notion of *General-Problem-hardness* [Ben+19b] to show that Diameter parameterized by k is "as hard" as the unparameterized Diameter problem. In Section 3.3, we investigate parameter combinations that are motivated by properties of social networks. Social networks often have special characteristics, including the *small-world* property (small diameter) and a *power-law degree distribution* (small average degree and small h-index) [LH08, Mil67, New03, New10, NP03]. Since social networks often have small diameter and small h-index, we investigate combinations of parameters closely related to the diameter and parameters closely related to the h-index.

The domination number d is a parameter that upper-bounds the diameter and the acyclic chromatic number a upper-bounds the average degree and is upper-bounded by the h-index. Hence, the standard $O(n \cdot m)$-time algorithm runs in $O(n^2 \cdot a)$ time. We will show that this is essentially the best one can hope for as, assuming the SETH, we can exclude $f(a,d) \cdot (n+m)^{2-\varepsilon}$-time algorithms for each $\varepsilon > 0$. Our result is based on a reduction by Roditty and Williams [RW13] which is modified such that the acyclic chromatic number and the domination number in the resulting graph are five and four, respectively. It is known that a $k^{O(1)}(n+m)^{2-\varepsilon}$-time algorithm where k is the combined parameter diameter plus maximum degree would refute the SETH [BN19]. Complementing this lower bound, we provide an $f(k)(n+m)$-time algorithm where k is the combined parameter diameter plus h-index. The maximum degree upper-bounds the h-index.

3.1 Problem Definition and Related Work

Diameter asks for the maximum distance between any two vertices in a given undirected and connected input graph. It is formally defined as follows and an example is given in Figure 3.1. Recall that $\operatorname{dist}_G(v,w)$ is the length of a shortest path between v and w in G.

Diameter
Input: An undirected and connected graph $G := (V, E)$.
Task: Compute the length of a longest shortest path in G, that is, $\max\{\operatorname{dist}_G(u,v) \mid u,v \in V\}$.

Due to its importance, Diameter is extensively studied. Concerning worst-case analysis, the theoretically fastest algorithms (in terms of dependence on the number n of vertices) are based on matrix multiplication and run in $O(n^{2.373})$ time [Sei95]. In terms of the dependence on the input size $n+m$, the currently fastest algorithms for All-Pairs Shortest Paths run in $O(n^3/2^{\Omega(\sqrt{\log n})})$ time in dense graphs [CW21] and in $O(nm)$ time in sparse graphs, respectively.

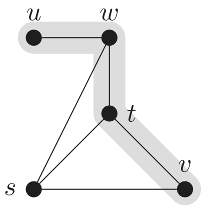


Figure 3.1: An undirected connected graph $G = (\{s, t, u, v, w\}, E)$. The diameter of G is 3 as $\text{dist}_G(u, v) = 3$ and $\text{dist}_G(x, y) \leq 3$ for all $x, y \in \{s, t, u, v, w\}$.

The $O(nm)$-time algorithm performs a breadth-first search from each vertex and algorithms for Diameter employed in practice are usually based on this approach. See e. g. Borassi et al. [Bor+15] for a recent example of such an algorithm which also yields good performance bounds using average-case analysis [BCT17].

Concerning special graph classes, Gawrychowski et al. [Gaw+21] showed how to solve Diameter on planar graphs in $\tilde{O}(n^{5/3})$ time. Other special cases include linear-time algorithms for outerplanar graphs [FP80] and chordal graphs [Cor+01].

In this chapter, we follow the line of *FPT in P* [GMN17]. Starting *FPT in P* for Diameter, Abboud et al. [AVW16] observed that, unless the SETH fails, there is no $k^{O(1)} \cdot (n+m)^{2-\varepsilon}$-time algorithm for Diameter for any $\varepsilon > 0$ if k is the treewidth of the graph. Their corresponding reduction also shows the same hardness result for the combined parameter h-index plus domination number and the parameter vertex cover number. Moreover, the reduction also implies that the SETH is refuted by any $f(k)(n+m)^{2-\varepsilon}$-time algorithm for Diameter for any computable function f and any $\varepsilon > 0$ when k is the distance to chordal graphs. Evald and Dahlgaard [ED16] adapted the reduction to prove the same for the parameter maximum degree.

Complementing the lower bound for the parameter treewidth by Abboud et al. [AVW16], Bringmann et al. [BHM20] showed that Diameter can be solved in $2^{O(k)}n^{1+o(1)}$ time where k is the treewidth of the graph. In the paper on which this chapter is based, we systematically explored the parameter space looking for parameters that allow for $k^{O(1)} \cdot (n+m)^{2-\varepsilon}$-time algorithms [BN19]. Figure 3.2 gives an overview over the parameterized results for Diameter and we will present in this chapter some selected results we achieved. The following results on approximating Diameter are known. A simple breadth-first search yields a linear-time 2-approximation. Aingworth et al. [Ain+99]

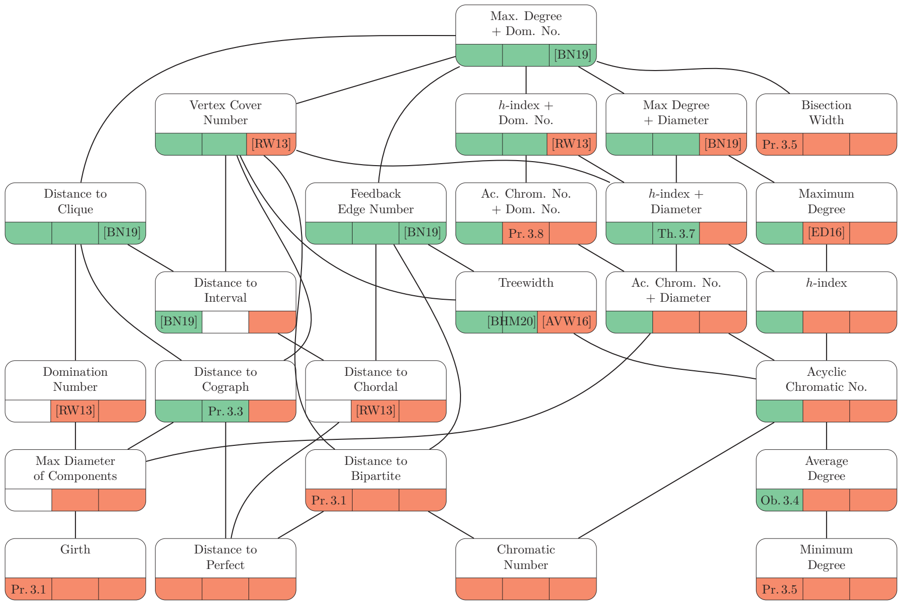


Figure 3.2: Overview of the relation between the structural parameters and the respective results for DIAMETER. An edge from a parameter α to a parameter β below of α means that β can be upper-bounded in a polynomial (usually linear) function in α (see also the work by Schröder [Sch19]). The three small boxes below each parameter indicate whether there exists (from left to right) an algorithm running in $f(k)n^2$, $f(k)(n+m)^{1+\varepsilon}$, or $k^{O(1)}(n+m)^{1+\varepsilon}$ time, respectively. If a small box is green (lighter), then a corresponding algorithm exists and the box to the left is also green. Similarly, a red (darker) box indicates that a corresponding algorithm would be a breakthrough. More precisely, if a middle box (right box) is red, then an algorithm running in $f(k) \cdot (n+m)^{2-\varepsilon}$ (or $k^{O(1)} \cdot (n+m)^{2-\varepsilon}$) time refutes the SETH. If a left box is red, then an algorithm with running time $f(k)n^2$ implies an $O(n^2)$-time algorithm for DIAMETER in general. Hardness results for a parameter α imply the same hardness results for the parameters below α. Similarly, algorithms for a parameter β imply algorithms for the parameters above β. White boxes indicate open problems.

improved the approximation factor to 3/2 at the expense of the higher running time of $O(n^2 \log n + m\sqrt{n \log n})$. Roditty and Williams [RW13] showed that approximating DIAMETER within a factor of $3/2 - \delta$ in $O(n^{2-\varepsilon})$ for any $\delta, \varepsilon > 0$

time refutes the SETH. Moreover, for any $\varepsilon, \delta > 0$ a $(3/2 - \delta)$-approximation in $O(m^{2-\varepsilon})$ time or a $(5/3 - \delta)$-approximation in $O(m^{3/2-\varepsilon})$ time also refute the SETH [Bac+18, CGR16]. On planar graphs, there is an approximation scheme with near linear running time [WY16].

3.2 Parameters Motivated by Graph Classes

In this section, we investigate parameterizations that measure the distance to special graph classes. We study the odd cycle transversal number (the distance to bipartite graphs) and the distance to cographs. Roditty and Williams [RW13] state that when computing the diameter of a graph, then distinguishing between diameter two and diameter three is among the most difficult cases. Nonetheless, detecting cographs (a subclass of graphs with diameter two) is easier than computing the diameter. Moreover, if the graph contains only few pairs of vertices of distance at least three, then the distance to cographs is often small. Thus, an efficient algorithm for DIAMETER parameterized by the distance to cographs might help dealing with the hard case of DIAMETER stated above. Besides cographs, we study bipartite graphs as these are among the most fundamental graph classes. Note that the lower bound of Abboud et al. [AVW16] for the parameter vertex cover number (distance to edgeless graphs) already implies that there is no $k^{O(1)} \cdot (n+m)^{2-\varepsilon}$-time algorithm for k being either of the two considered parameters as both are upper-bounded by the vertex cover number (see Figure 3.2).

Odd Cycle Transversal. We will show that, assuming the SETH, there is no $f(k) \cdot (n+m)^{2-\varepsilon}$-time algorithm for the odd cycle transversal number k for any computable function f. We do so by showing that DIAMETER is 4-GP-hard[1] with respect to the combined parameter odd cycle transversal number plus girth. Recall that the girth of a graph is the size of a smallest induced cycle in the graph and that DIAMETER is 4-GP-hard with respect to the parameter odd cycle transversal number plus girth if the following holds. Each instance (G, k) of the decision

[1]We remark that Definition 2.1 and Lemma 2.3 are stated for decision problems while DIAMETER is not a decision problem. However, the problem of deciding whether a given undirected connected graph has diameter *exactly* k for some given k is a decision problem and every algorithm for DIAMETER can be used to solve this decision problem with constant overhead. We call DIAMETER GP-hard with respect to some parameter, when this decision version of DIAMETER is GP-hard with respect to that parameter.

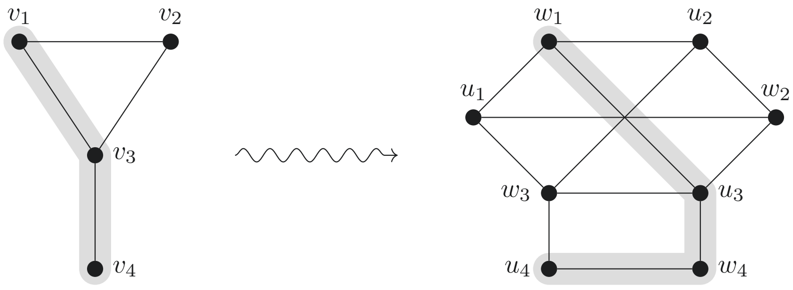


Figure 3.3: Example for the construction in the proof of Proposition 3.1. The input graph given on the left side has diameter two and the constructed graph on the right side has diameter three. In each graph one longest shortest path is highlighted.

version of DIAMETER can be transformed in $O(|G|)$ time into an instance (G', k') such that the odd cycle transversal number plus girth of G' is at most four and G' has diameter k' if and only if G has diameter k. Recall further that if DIAMETER is 4-GP-hard with respect to some parameter ℓ, then Lemma 2.3 states the following. If there is an algorithm solving each instance (G, k, ℓ) of the decision version of DIAMETER in $O(f(\ell) \cdot g(|G|))$ time for any computable functions f and g, then there is an algorithm that solves each instance (G', k') of the unparameterized decision version of DIAMETER in $O(g(|G'|))$ time. This then yields the following two results. First, any $f(k) \cdot n^{2.3}$-time algorithm can be transformed into an $O(n^{2.3})$-time algorithm for DIAMETER (which is faster than any known unparameterized algorithm). Second, any $f(k) \cdot (n+m)^{2-\varepsilon}$-time algorithm would refute the SETH.

Proposition 3.1. *DIAMETER is 4-GP-hard with respect to the combined parameter odd cycle transversal plus girth.*

Proof. Let $G = (V, E)$ be an arbitrary undirected connected input graph with $V = \{v_1, v_2, \ldots, v_n\}$. We construct a new bipartite graph $G' = (V', E')$, where

$$V' := \{u_i, w_i \mid v_i \in V\}, \text{ and}$$

$$E' := \{\{u_i, w_j\}, \{u_j, w_i\} \mid \{v_i, v_j\} \in E\} \cup \{\{u_i, w_i\} \mid v_i \in V\}.$$

An example of this construction can be seen in Figure 3.3. We will now prove that all properties of Definition 2.1 hold. It is easy to verify that the construction can be computed in linear time and therefore the resulting instance is of linear size as well. Observe that $\{u_i \mid v_i \in V\}$ and $\{w_i \mid v_i \in V\}$ are both independent sets and therefore G' is bipartite. Notice further that for any edge $\{v_i, v_j\} \in E$ there is an induced cycle in G' containing the vertices u_i, w_i, u_j, and w_j. Since G' is bipartite, there is no induced cycle of length three in G' and thus the girth of G' is four.

Lastly, we show that the diameter of G' is exactly one larger than the diameter of G. We do so by proving for each pair (v_i, v_j) of vertices in G that if $\operatorname{dist}(v_i, v_j)$ is odd, then

$$\operatorname{dist}(u_i, w_j) = \operatorname{dist}(v_i, v_j) \text{ and } \operatorname{dist}(u_i, u_j) = \operatorname{dist}(v_i, v_j) + 1,$$

and if $\operatorname{dist}(v_i, v_j)$ is even, then

$$\operatorname{dist}(u_i, u_j) = \operatorname{dist}(v_i, v_j) \text{ and } \operatorname{dist}(u_i, w_j) = \operatorname{dist}(v_i, v_j) + 1.$$

Since $\operatorname{dist}(u_i, w_i) = 1$ and $\operatorname{dist}(u_i, w_j) = \operatorname{dist}(u_j, w_i)$, this will conclude the proof.

In order to show that the diameter of G' is exactly one larger than the diameter of G, let $c = \operatorname{dist}(v_i, v_j)$ be odd and let $P = (v_{a_0}, v_{a_1}, \ldots, v_{a_c})$ be a shortest path from v_i to v_j where $v_{a_0} = v_i$ and $v_{a_c} = v_j$. Let

$$P' = (u_{a_0}, w_{a_1}, u_{a_2}, \ldots, w_{a_c})$$

be a path in G'. Clearly P' has length c and hence $\operatorname{dist}(u_i, w_j) \leq c = \operatorname{dist}(v_i, v_j)$. It also holds that $\operatorname{dist}(u_i, w_j) \geq c$. To verify this, assume towards a contradiction that there is a path $P'' = (u_{b_0}, w_{b_1}, u_{b_2}, \ldots, w_{b_{c'}})$ with $u_{b_0} = u_i$, $w_{b_{c'}} = w_j$, and $c' < c$. Then there is a path $P''' = (v_{b_0}, v_{b_1}, \ldots, v_{b_{c'}})$ between v_i and v_j. Note that if $v_{b_g} = v_{b_h}$ for some $b_g < b_h$, then $g = h$ and P''' can be replaced by a shorter path where the subpath $P'''[b_{g+1}, b_h]$ is removed. Thus, the distance between v_i and v_j is shorter than c, a contradiction.

Concerning $\operatorname{dist}(u_i, u_j)$, observe that G' is bipartite and hence $\operatorname{dist}(u_i, u_j)$ is even. It holds that $\operatorname{dist}(u_i, u_j) > c$ as $\operatorname{dist}(u_i, u_j) \geq c$ for the same reason as $\operatorname{dist}(u_i, w_j) \geq c$ and $\operatorname{dist}(u_i, u_j) \neq c$ as c is odd. Finally, since

$$P' \bullet (u_j) = (u_{a_0}, w_{a_1}, u_{a_2}, \ldots, w_{a_c}, u_{a_c})$$

is a path of length $c+1$ between u_i and $u_{a_c} = u_j$, it holds that $\operatorname{dist}(u_i, u_j) = c+1$.

It remains to analyze the case where the distance $c = \text{dist}(v_i, v_j)$ between two vertices in G is even. Let again $P = (v_{a_0}, v_{a_1}, \ldots, v_{a_c})$ be a shortest path from v_i to v_j where $v_{a_0} = v_i$ and $v_{a_c} = v_j$. This time, let

$$P' = (u_{a_0}, w_{a_1}, u_{a_2}, \ldots, u_{a_c})$$

be a path in G'. This shows that $\text{dist}(u_i, w_j) \leq \text{dist}(v_i, v_j)$. It again holds that $\text{dist}(u_i, w_j) \geq c$ as if there would be a path $P'' = (u_{b_0}, w_{b_1}, u_{b_2}, \ldots, u_{b_{c'}})$ with $u_{b_0} = u_i$, $u_{b_{c'}} = u_j$, and $c' < c$, then there would also be a shorter path $P''' = (v_{b_0}, v_{b_1}, \ldots, v_{b_{c'}})$ between v_i and v_j. Observe that $\text{dist}(u_i, w_j)$ is odd as G' is bipartite. Thus, $\text{dist}(u_i, w_j) > c$ as $\text{dist}(u_i, w_j) < c$ again implies $\text{dist}(v_i, v_j) < c$ and $\text{dist}(u_i, w_j) \neq c$ as one is odd and the other one is even. Finally,

$$P' \bullet (w_j) = (u_{a_0}, w_{a_1}, u_{a_2}, \ldots, u_{a_c}, w_{a_c})$$

proves that $\text{dist}(u_i, w_j) = c + 1 = \text{dist}(v_i, v_j) + 1$. □

Distance to cographs. We continue with the distance to cographs. A graph is a cograph if and only if it does not contain a P_4 as an induced subgraph, where a P_4 is a path on four vertices. Providing an algorithm that matches the lower bound of Abboud et al. [AVW16], we will show that DIAMETER parameterized by distance k to cographs can be solved in $O(k \cdot (n + m) + 2^{O(k)})$ time. We will use the following lemma.

Lemma 3.2. *Let $G = (V, E)$ be a graph and let $K \subseteq V$ a vertex subset such that each connected component in $G - K$ has diameter at most two. Then, the diameter of G can be computed in $O(|K| \cdot (n + m + 2^{4|K|}))$ time.*

Proof. Let $G = (V, E)$ be the input graph, let $K = \{x_1, x_2, \ldots, x_k\} \subseteq V$ be a set of vertices such that each connected component in G has diameter at most two and let $G' \coloneqq G - K$. We first compute the set of all connected components of G' and their respective diameter in linear time and store for each vertex the information in which connected component it is contained. Note that we only need to check for each connected component C whether C induces a clique in G', as otherwise C's diameter is by assumption two. In a second step, we perform from each vertex $x_i \in K$ a breadth-first search in G and store the distance between x_i and each other vertex v in a table. Since a single breadth-first search takes $O(n + m)$ time, this takes overall $O(k \cdot (n + m))$ time.

Next we introduce some notation. The *type* of a vertex $v \in V \setminus K$ is a vector of length k where the i^{th} entry describes the distance from v to x_i with the

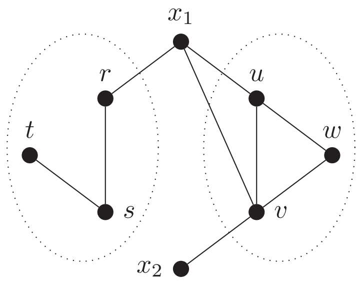


Figure 3.4: An example for *types*. The set K contains the two vertices x_1 and x_2 and the connected components in $G - K$ are depicted. The type of r is $(1, 3)$, the type of s is $(2, 4)$, the type of t is $(3, 4)$, the type of u is $(1, 2)$, the type of v is $(1, 1)$, and the type of w is $(2, 2)$.

addition that any value above three is set to four. An example is given in Figure 3.4. We say that a type is *non-empty* if there is at least one vertex of this type. We compute for each vertex $v \in V \setminus K$ its type. Additionally we store for each non-empty type the vertices of this type. Moreover, if all vertices of this type are in the same connected component, then we store this information, and otherwise we store that there are at least two different connected components containing a vertex of that type. This takes $O(n \cdot k)$ time and there are at most 4^k different types.

Lastly, we iterate over all of the at most 4^{2k} pairs (t_1, t_2) of non-empty types (including the pairs where both types are the same) and compute the largest distance between vertices of these types. Let y, z be two vertices with $\text{type}(y) = t_1$ and $\text{type}(z) = t_2$ that have maximum pairwise distance. We will first discuss how to find y and z and then show how to correctly compute their distance in $O(k)$ time. Once we iterated over all pairs of types and reported the maximum distance found, the diameter is either this or the largest distance from a vertex $x_i \in K$. Since we stored all of the latter distances in a table, we can also store the maximum with only constant overhead.

To compute y and z, we consider the following two cases. If both types only appear in the same connected component, then the distance between the two vertices of these types is at most two. Hence, we can discard this case (one can check in linear time whether the diameter of G is at least two). If two types appear in different connected components, then a longest shortest path between vertices of the respective types contains at least one vertex in K. Observe that since each connected component has diameter at most two, each third vertex in any shortest path must be in K. Thus a shortest y-z–path contains at least one vertex $x_i \in K$ with $\operatorname{dist}(x_i, y) < 3$. By definition, each vertex with the same type as y has the same distance to x_i and therefore the same distance to z unless there is no shortest path from it to z that passes through x_i, that is, it is in the same connected component as z. Hence, we can choose two arbitrary vertices of the respective types in different connected components. Observe that we already precomputed for each type its vertices and whether it is represented in multiple connected components or not. Thus, checking whether there are two vertices of the respective type in different connected components is just a table lookup. We can compute the distance between y and z in $O(k)$ time by computing $\min_{x \in K}\{\operatorname{dist}(y, x) + \operatorname{dist}(x, z)\}$. Observe that the shortest path from y to z contains x_i and therefore $\operatorname{dist}(y, x_i) + \operatorname{dist}(x_i, z) = \operatorname{dist}(y, z)$. In this way, we can compute the diameter of G in $O(k \cdot (n + m + 2^{4k}))$ time. □

Note that the algorithm described in the proof above does not verify whether K is a vertex set such that each connected component in $G - K$ has diameter at most two. Indeed, even distinguishing between diameter two and three in $O(n^{2-\varepsilon})$ time for any $\varepsilon > 0$ would refute the SETH [AVW16]. Thus, the above algorithm cannot efficiently verify whether the input meets the stated conditions. Hence, when using Lemma 3.2, we need a way to ensure that each connected component in $G - K$ has diameter two. In cographs each connected component has diameter two and hence we can show the following.

Proposition 3.3. *Diameter can be solved in $O(k \cdot (n + m + 2^{16k}))$ time when parameterized by the distance k to cographs.*

Proof. Recall that a cograph does not contain a P_4 as an induced subgraph. Thus, any cograph has diameter at most two (but not every diameter-two graph is a cograph, consider e. g. a cycle on five vertices). Moreover, given a graph G, one can determine in linear time whether G is a cograph and can return an induced P_4 if this is not the case [Bre+08, CPS85]. Iteratively searching for an induced P_4, adding all four vertices of a returned P_4 to a set K, and deleting those vertices from G until it is P_4-free hence computes a set $K \subseteq V$ with $|K| \leq 4k$ such that $G - K$ is a cograph. The running time for computing K is in $O(k \cdot (n + m))$. Applying Lemma 3.2 to this set K then yields a running time of $O(|K| \cdot (n + m + 2^{4|K|})) \subseteq O(k \cdot (n + m + 2^{16k}))$ for computing the diameter. □

Observe that when a minimum deletion set K to cographs is given, then we can solve Diameter parameterized by the distance k to cographs in $O(k\cdot(n+m+2^{4k}))$ time. We remark that computing the distance to cographs exactly is *NP*-complete [LY80].

3.3 Parameters Motivated by Properties of Social Networks

In this section, we study Diameter with respect to parameters that are expected to be small in social networks. It was observed that social networks have the *small-world* property and a *power-law degree distribution* [LH08, Mil67, New03, New10, NP03]. The small-world property directly transfers to the diameter. The power-law degree distribution is often captured by the h-index as only few high-degree vertices exist in the network. Thus, we investigate parameters related to the diameter and to the h-index. We start with some degree-based parameters that are upper-bounded by the h-index and then continue with parameter combinations.

Evald and Dahlgaard [ED16] showed that any $f(k)(n+m)^{2-\varepsilon}$-time algorithm for Diameter parameterized by the maximum degree k for any computable function f refutes the SETH. Observe that $2m = n \cdot a$, where a is the average degree and therefore the standard algorithm (run a breadth-first search from each vertex) takes $O(n \cdot (n+m)) = O(a \cdot n^2)$ time. Since the average degree is at most the maximum degree, this algorithm already matches the given lower bound.

Observation 3.4. *Diameter parameterized by average degree a is solvable in $O(a \cdot n^2)$ time.*

We next investigate the parameter minimum degree and check whether the average degree can be replaced by the minimum degree. Unsurprisingly, it cannot. We show that Diameter is 2-GP-hard with respect to the combined parameter bisection width plus minimum degree. In other words, if there is an $f(b) \cdot n^2$-time algorithm, where b is the value of the combined parameter, then there is also an $O(n^2)$-time algorithm for Diameter. The bisection width of a graph G is the minimum number of edges to delete from G in order to partition G into two connected component whose number of vertices differ by at most one. Computing the bisection width of a graph is known to be *NP*-hard [Bui+87].

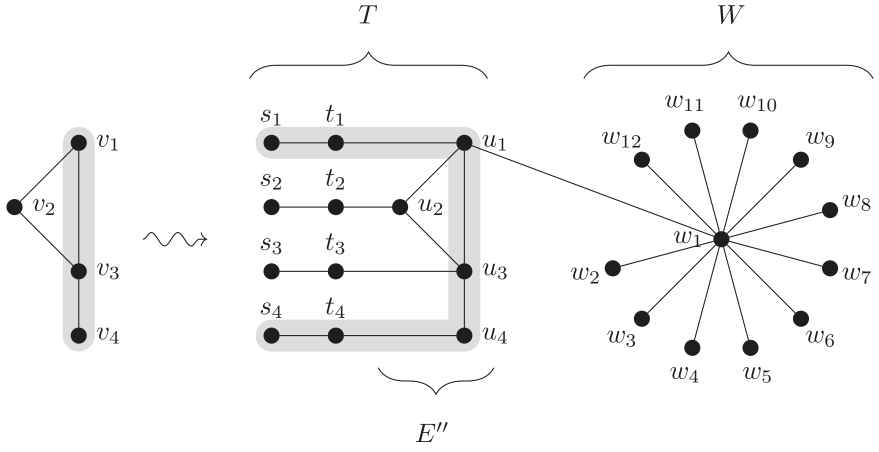


Figure 3.5: Example for the construction in the proof of Proposition 3.5. The input graph given on the left side has diameter two and the constructed graph on the right side has diameter $2 + 4 = 6$. The respective longest shortest paths are highlighted.

Proposition 3.5. *Diameter is 2-GP-hard with respect to the combined parameter bisection width plus minimum degree.*

Proof. Let $G = (V, E)$ be an arbitrary undirected connected input graph with $V = \{v_1, v_2, \ldots, v_n\}$ and let d be the diameter of G. We construct a new graph $G' = (V', E')$ with diameter $d + 4$ as follows. Let

$$\begin{aligned} V' &:= \{s_i, t_i, u_i \mid i \in [n]\} \cup \{w_i \mid i \in [3n]\}, \text{ and} \\ E' &:= T \cup W \cup E'', \text{ where} \\ T &:= \{\{s_i, t_i\}, \{t_i, u_i\} \mid i \in [n]\}, \\ W &:= \{u_1, w_1\} \cup \{\{w_1, w_i\} \mid i \in ([3n] \setminus \{1\})\}, \text{ and} \\ E'' &:= \{\{u_i, u_j\} \mid \{v_i, v_j\} \in E\}. \end{aligned}$$

An example of this construction can be seen in Figure 3.5. We will now prove that all properties of Definition 2.1 hold. It is easy to verify that the graph G' contains $6n$ vertices and $5n + m$ edges, and that G' can be computed in linear time. Notice that $\{s_i, t_i, u_i \mid i \in [n]\}$ and $\{w_i \mid i \in [3n]\}$ are both of size $3n$ and that there is only one edge ($\{u_1, w_1\}$) between these two sets of vertices. The bisection width of G' is therefore one and the minimum degree is also one as s_1 has only t_1 as neighbor.

It remains to show that G' has diameter $d+4$. First, notice that the subgraph of G' induced by $\{u_i \mid i \in [n]\}$ is isomorphic to G. Second, $\operatorname{dist}(s_i, u_i) = 2$ for all $i \in [n]$ and thus $\operatorname{dist}(s_i, s_j) = \operatorname{dist}(u_i, u_j) + 4 = \operatorname{dist}(v_i, v_j) + 4$ for all $s_i \neq s_j$. Hence, the diameter of G' is at least $d + 4$. Third, note that it holds for all vertices $x \in V' \setminus \{s_i\}$ that $\operatorname{dist}(s_i, x) > \operatorname{dist}(t_i, x)$. Lastly, observe that for all $i \in [3n]$ and all vertices $x \in V'$ it holds that $\operatorname{dist}(w_i, x) \leq \max\{\operatorname{dist}(s_1, x), 4\}$. Thus the longest shortest path in G' is between two vertices s_i and s_j and it is of length $\operatorname{dist}(u_i, u_j) + 4 = \operatorname{dist}(v_i, v_j) + 4 \leq d + 4$. □

We mention in passing that the constructed graph in the proof of Proposition 3.5 contains the original graph as an induced subgraph and if the original graph is bipartite, then so is the constructed graph. Thus, first applying the construction in the proof of Proposition 3.1 (see also Figure 3.3) and then the construction in the proof of Proposition 3.5 (see also Figure 3.5) shows that Diameter is GP-hard even when parameterized by the sum of girth, bisection width, minimum degree, and odd cycle traversal.

Corollary 3.6. *Diameter is 6-GP-hard with respect to the combined parameter odd cycle traversal number plus girth plus bisection width plus minimum degree.*

h-index and diameter. We next investigate the combined parameter h-index plus diameter. The reduction by Roditty and Williams [RW13] produces instances with constant domination number and logarithmic vertex cover number (in the input size). Since the diameter d is linearly upper-bounded by the domination number and the h-index is linearly upper-bounded by the vertex cover number, any algorithm that solves Diameter parameterized by the combined parameter $(d+h)$ in $(d+h)^{O(1)} \cdot (n+m)^{2-\epsilon}$ time disproves the SETH. We next present the main result in this chapter, that is, an algorithm for Diameter parameterized by h-index plus diameter that almost matches the lower bound. We say that the running time almost matches the lower bound since its dependence on the parameter is roughly $O(h^d + d^h) = O(2^{d \log h + h \log d})$. Hence, it remains open whether an algorithm with a running time of $(n+m) \cdot 2^{O(d+h)}$ exists. We consider the following algorithm our main result of this chapter for two reasons. First, its running time almost matches the lower bound for the relevant special case where the input graph has similar properties to social networks (namely small diameter and small h-index). Second, the dynamic program we develop here is quite unique in the sense that the solution of the problem is not related to some table entry but rather to the *size* of the table.

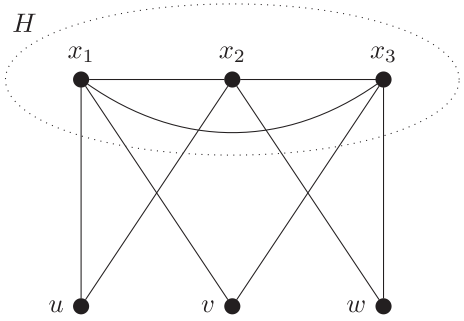
Theorem 3.7. *Diameter parameterized by* diameter d plus h-Index h *is solvable in* $O((n+m)\cdot h\cdot(h^d+d^h))$ *time.*

Proof. Let $G=(V,E)$ be an input graph for Diameter and let $H=\{x_1,\dots,x_h\}$ be a set of h vertices with highest degree in G. Clearly, H can be computed in linear time. Notice that all vertices in $V\setminus H$ have degree at most h in G.

We will describe a two-phase algorithm based on the following idea. In the first phase, it performs a breadth-first search from each vertex $x_i\in H$, stores the distance to each other vertex, and uses this to compute the *type* of each vertex, that is, a vector containing the distances to each vertex in H. In the second phase, the algorithm iteratively increases a value e and verifies whether there is a pair of vertices of distance at least $e+1$ using dynamic programming. If at any point no such pair is found, then the diameter of G is e.

The first phase is fairly straightforward. The algorithm performs a breadth-first search from each vertex $x_i\in H$ and stores the distance from x_i to each vertex v in a table. We denote the maximum entry in this table by a. It then iterates over each vertex $v\in V\setminus H$ and computes a vector of length h with the i^{th} entry representing the distance from v to x_i. An example of types is depicted in Figure 3.6. The algorithm also stores the number of vertices of each type (if there is at least one such vertex). Since the distance to any vertex is at most d, there are at most d^h different types. Let $\mathcal{T}$ be the set of all (non-empty) types and for some $t\in\mathcal{T}$ let $\#_t$ be the total number of vertices of type t.

For the second phase, we deploy a dynamic program that uses two tables N and T. The table $N\colon V\times\mathbb{N}\to 2^V$ keeps for each vertex v and each possible distance e track of all vertices that have distance exactly e in $G'=G-H$. The table $T\colon V\times\mathbb{N}\times\mathcal{T}\to\mathbb{N}$ stores for each vertex v, each distance e, and each type t the number of vertices of type t that have distance at most e from v in G. Initially $e=1$, $N[v,0]=\{v\}$, and $N[v,1]=N(v)$ for each vertex v. Before we show how to initialize T, we explain the main idea behind it. Note that a shortest path between v and a vertex w of type t either contains a vertex in H or it is completely contained in G'. If it contains a vertex $x_i\in H$, then the distance between v and w is $\operatorname{dist}(v,x_i)+\operatorname{dist}(x_i,w)$. Hence, assuming that a shortest path contains a vertex in H, the distance between v and w is the minimum entry in $\operatorname{type}(v)+\operatorname{type}(w)=\operatorname{type}(v)+t$. We denote this minimum entry by $\operatorname{mt}(v,t)$.



	x_1	x_2	x_3
x_1	0	1	1
x_2	1	0	1
x_3	1	1	0
u	1	1	2
v	1	2	1
w	2	1	1

Figure 3.6: An example of types. Each entry in the table on the right side displays the distance between the two respective vertices. Each column is computed by a breadth-first search from the respective vertex x_i and each row is the type of the respective vertex. The last row states for example that the distance between w and x_1, x_2, and x_3 are 2, 1, and 1, respectively. Thus, the type of w is $(2, 1, 1)^T$.

Since a path of length zero or one between two vertices $v, w \in V \setminus H$ cannot contain a vertex in H, the table can be initialized by

$$T[v, 0, t] = \begin{cases} 1, & \text{if } \operatorname{type}(v) = t, \\ 0, & \text{otherwise, and} \end{cases}$$
$$T[v, 1, t] = T[v, 0, t] + |\{u \in N[v, e] \mid \operatorname{type}(u) = t\}.$$

The algorithm now iteratively increases e and computes $N[v, e]$ and $T[v, e, t]$ for each $v \in V \setminus H$ and each $t \in \mathcal{T}$ until in one iteration $T[v, e, t] = \#_t$ for all v and all t. Once this is the case, all vertices in $V \setminus H$ have pairwise distance at most e. Since we already computed the distance from each vertex $x_i \in H$ to each other vertex, the maximum over all these distances and e is the diameter of G. The recursive formulas for N and T are as follows.

$$N[v, e] = |(\bigcup_{u \in N[v, e-1]} N(u)) \setminus (N[v, e-1] \cup N[v, e-2])| \text{ and}$$

$$T[v, e, t] = \begin{cases} \#_t, & \text{if } \operatorname{mt}(v, t) \leq e, \text{ and} \\ T[v, e-1, t] + |\{u \in N[v, e] \mid \operatorname{type}(u) = t\}| & \text{otherwise.} \end{cases}$$

If at some iteration it holds that $T[v, e, t] = \#_t$ for all vertices v and all types t, then the algorithm terminates and returns $\max\{e, a\}$. Observe that e

is equal to the number of table entries in T divided by $|V \setminus H| \cdot |\mathcal{T}|$. Thus, the solution returned by the dynamic program is not depending on any value stored within T but rather on the number of table entries in (the *size* of) T.

There are at most d iterations in which e is increased and table entries of N and T are computed. Note that all values of the function mt can be precomputed in $O(|\mathcal{T}|^2 \cdot h) \subseteq O(d^h \cdot h \cdot n)$ time as $|\mathcal{T}| \leq d^h$ and $|\mathcal{T}| \leq n$. Note that the computation of N closely resembles a breadth-first search in G' and since the maximum degree in G' is h and the maximum depth is d, computing all entries of N for a single vertex takes $O(h^d)$ time. To compute all entries $T[v, e, t]$ for all $t \in \mathcal{T}$ simultaneously, we iterate over each vertex $w \in N[v, e]$ and increase the entry $T[v, e, \mathrm{type}(w)]$ by one. This takes $O(\sum_{e=1}^{d} |N[v, e]|) \subseteq O(h^d)$ time for each vertex. The running time of our algorithm is $O(h \cdot (n + m))$ for the first phase and $O((d^h \cdot h \cdot n) + n \cdot h^d)$ for the second phase. This yields an overall running time of

$$O((n + m + h) \cdot d^h + n \cdot h^d) \subseteq O((n + m) \cdot h \cdot (h^d + d^h)). \qquad \square$$

Acyclic chromatic number and domination number. Finally, we analyze the parameterized complexity of DIAMETER parameterized by the acyclic chromatic number a plus domination number d. Note that this combined parameter is incomparable with the combined parameter h-index plus diameter as the h-index upper-bounds the acyclic chromatic number but the domination number upper-bounds the diameter. Recall that the acyclic chromatic number of a graph G is the smallest number a such that the vertices of G can be partitioned into a independent sets such that the induced subgraph of each combination of two of these independent sets is acyclic. We provide a SETH-based lower bound, adapting a reduction from SATISFIABILITY to DIAMETER by Roditty and Williams [RW13].

Proposition 3.8. *There is no $f(a, d) \cdot (n + m)^{2-\epsilon}$-time algorithm for any computable function f that solves* DIAMETER *parameterized by acyclic chromatic number a plus domination number d unless the SETH is false.*

Proof. We provide a reduction from SATISFIABILITY to DIAMETER where the input instance has constant acyclic chromatic number and constant domination number and such that an $O((n + m)^{2-\varepsilon})$-time algorithm refutes the SETH. We note that the reduction is an extension of the construction by Roditty and Williams [RW13, Theorem 9]. Let ϕ be a SATISFIABILITY instance with variable set W and clause set C. Assume without loss of generality that $|W|$ is even. We construct an instance graph $G = (V, E)$ for Diameter as follows.

Randomly partition W into two sets W_1 and W_2 of equal size. Add three sets V_1, V_2, and B of vertices to G, where each vertex in V_1 (in V_2) represents one of $2^{|W|/2}$ possible truth assignments of the variables in W_1 (in W_2) and each vertex in B represents a clause in C. Clearly, $|V_1|+|V_2| = 2\cdot 2^{|W|/2}$ and $|B| = |C|$. For each $v_i \in V_1$ and each $u_j \in B$, if the truth assignment corresponding to v_i does *not* satisfy the clause corresponding to u_j, then we add a new vertex s_{ij} and the two edges $\{v_i, s_{ij}\}$ and $\{u_j, s_{ij}\}$ to G. We call the set of all these newly introduced vertices S_1. Now repeat the process for all vertices $w_i \in V_2$ and all u_j in B and call the newly introduced vertices q_{ij}. Let S_2 be the set of all q_{ij}. Finally we add four new vertices t_1, t_2, t_3, t_4 and the sets

$$\begin{aligned}
&\{\{t_1, v\} \mid v \in V_1\},\\
&\{\{t_2, s\} \mid s \in S_1\},\\
&\{\{t_3, q\} \mid q \in S_2\},\\
&\{\{t_4, w\} \mid w \in V_2\},\\
&\{\{t_2, b\}, \{t_3, b\} \mid b \in B\}, \text{ and}\\
&\{\{t_1, t_2\}, \{t_2, t_3\}, \{t_3, t_4\}\}
\end{aligned}$$

of edges to G. See Figure 3.7 for a schematic illustration of the construction.

We will first show that ϕ is satisfiable if and only if G has diameter five and then show that the domination number and acyclic chromatic number of G are five and four, respectively. Observe that the diameter of G is at most five since each vertex is connected to some vertex in $\{t_1, t_2, t_3, t_4\}$ and these four vertices are of pairwise distance at most three. First assume that ϕ is satisfiable. Then, there exists some truth assignment β of the variables such that all clauses are satisfied, that is, the two partial truth assignments of β with respect to the variables in W_1 and W_2 satisfy all clauses. Let $v_1 \in V_1$ and $v_2 \in V_2$ be the vertices corresponding to β. Thus, for each $b \in B$ we have $\operatorname{dist}(v_1, b) + \operatorname{dist}(v_2, b) \geq 5$. Observe that all paths from a vertex in V_1 to a vertex in V_2 that do not pass a vertex in B pass through t_2 and t_3 and are hence of length at least five. Thus, the diameter of G is $\operatorname{dist}(v_1, v_2) = 5$.

For the reverse direction, assume that there is no satisfying truth assignment for Φ. Then for each pair of vertices $v_1 \in V_1$ and $v_2 \in V_2$ it holds that there is some clause in Φ that is not satisfied by either of the two partial truth assignments corresponding to v_1 and v_2. Hence, the vertex u_j corresponding to

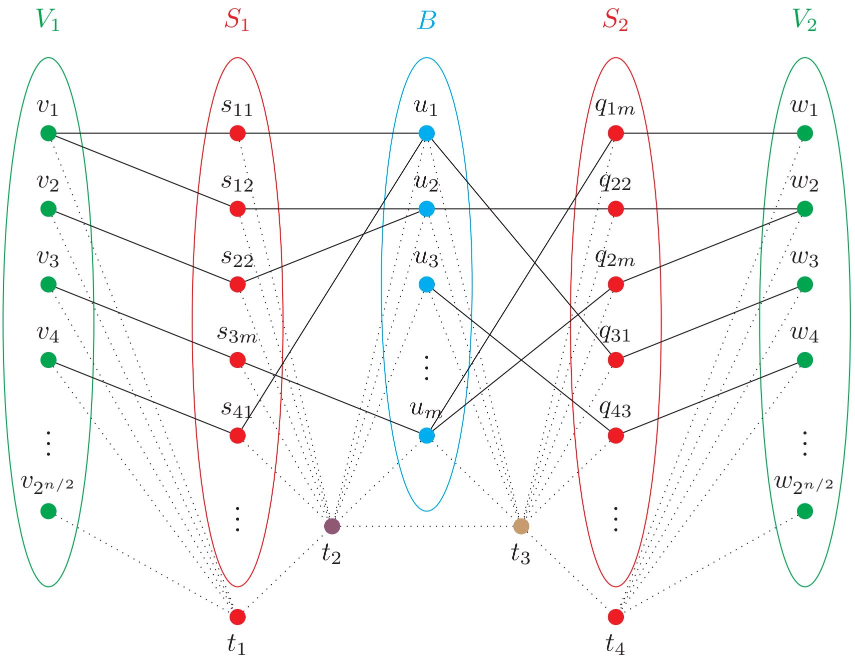


Figure 3.7: A schematic illustration of the construction in the proof of Proposition 3.8. A vertex $s_{i,j}$ is only connected to v_i and u_j and q_{ij} is only connected to w_i and u_j. Note that the resulting graph has acyclic chromatic number five (the five independent sets are $V_1 \cup V_2$, B, $S_1 \cup S_2 \cup \{t_1, t_4\}$, $\{t_2\}$, and $\{t_3\}$ and are also represented by colors). Moreover, the domination number of the graph is at most four as $\{t_1, t_2, t_3, t_4\}$ is a dominating set.

this clause guarantees that $\text{dist}(v_1, v_2) \leq \text{dist}(v_1, u_j) + \text{dist}(u_j, v_2) = 4$. Next, observe that each pair (v_1, v_2) of vertices where not both $v_1 \in V_1$ and $v_2 \in V_2$ (or $v_1 \in V_2$ and $v_2 \in V_1$) holds are of distance at most four as guaranteed by the vertices t_1, t_2, t_3, and t_4. Thus, the diameter of G is four.

The domination number of G is four since $\{t_1, t_2, t_3, t_4\}$ is a dominating set. The acyclic chromatic number of G is at most five as $V_1 \cup V_2, \{t_2\}, B, \{t_3\}$, and $S_1 \cup S_2 \cup \{t_1, t_4\}$ each induce an independent set and each combination of two of them not including $S_1 \cup S_2 \cup \{t_1, t_4\}$ only induces independent sets or stars. Moreover, note that $S_1 \cup S_2 \cup \{t_1, t_4\} \cup \{t_2\}$ and $S_1 \cup S_2 \cup \{t_1, t_4\} \cup \{t_3\}$ each only induces a star and an independent set. Lastly, $S_1 \cup S_2 \cup \{t_1, t_4\} \cup V_1 \cup V_2$ induces two trees of depth 2 (where t_1 and t_4 are the roots and S_1 and S_2 are the leaves) and $S_1 \cup S_2 \cup \{t_1, t_4\} \cup B$ induces a disjoint union of stars and isolated vertices as each vertex in $S_1 \cup S_2 \cup \{t_1, t_4\}$ has maximum degree one in $G[B \cup S_1 \cup S_2 \cup \{t_1, t_4\}]$.

Now assume that there was an $f(k) \cdot (n+m)^{2-\varepsilon}$-time algorithm for Diameter parameterized by domination number plus acyclic chromatic number for any computable function f and any $\varepsilon > 0$. The constructed graph has $O(2^{|W|/2} \cdot |C|)$ vertices and edges, and since $f(9)$ is some constant, this implies an algorithm with running time

$$\begin{aligned} & f(9) \cdot (2^{|W|/2} \cdot |C|)^{2-\varepsilon} \\ \in\ & O(2^{(|W|/2)(2-\varepsilon)} \cdot |C|^{(2-\varepsilon)}) \\ =\ & O(2^{|W|(1-\varepsilon/2)} \cdot |C|^{(2-\varepsilon)}) \\ =\ & 2^{|W|(1-\varepsilon')} \cdot (|C| + |W|)^{O(1)} \text{ for some } \varepsilon' > 0. \end{aligned}$$

Such an algorithm for Diameter would refute the SETH [RW13]. □

3.4 Concluding Remarks

We conclude this chapter with some possible avenues for further research regarding Diameter. We believe that a broader reflection on the techniques we used (e. g. dynamic programming) is better deferred to the concluding chapter of this thesis, where we can compare the different dynamic programs we develop in this thesis. Concerning the complexity landscape shown in Figure 3.2, only a few open cases remain. Perhaps most interesting among them are the following two questions. Is there a $k^{O(1)} \cdot (n+m)^{1+\varepsilon}$- time algorithm for the distance k to interval graphs and is there an $f(d)n^2$-time algorithm for Diameter parameterized by the diameter d? Our algorithms working with parameter combinations are probably not competitive to state-of-the-art unparameterized algorithms due to their exponential dependency on the parameter value(s) even in graphs with properties similar to social networks and even so they cannot be improved by much unless the SETH breaks. So the question remains whether there are parameters $k_1, \ldots, k_\ell$ (that are possibly not displayed in Figure 3.2) that are small in real-world applications and that allow for practically relevant running times like $\prod_{i=1}^{\ell} k_i \cdot (n+m)$ or even $(n+m) \cdot \sum_{i=1}^{\ell} k_i$. A parameter capturing the special community structures of social networks [GN02] might be a good candidate to be included in such a parameter combination.

Chapter 4

Length-Bounded Cuts

In this chapter, we investigate, on a conceptual level, a peculiar case of how to compute the solution for a problem, once the table of a dynamic program is completely filled. Similarly to Chapter 3, the first important question we have to answer is what a table entry should represent. Answering this question requires some structural observations and is by far the most complicated part of this chapter. However, once we have answered this question, determining the table dimension and computing each table entry are fairly straightforward while computing the solution from the filled table is not.

The problem we study in this chapter stems from the area of network flows. The study of network flows and, in particular, of the Edge-Disjoint Paths problem began in the 1950s with the work of Ford and Fulkerson [FF56] and has since then constituted a prominent research area in graph algorithms. In the Edge-Disjoint Paths problem, we are given an undirected graph G, two vertices s and t, called the *source* and the *target*, and a positive integer β. The question is whether there is a collection of at least β edge-disjoint s-t-paths in G. It is worth pointing out that nowadays there are many more efficient algorithms than the one by Ford and Fulkerson [FF56] for finding β edge-disjoint s-t-paths in a given graph (see e. g. the work by Dinitz [Din06]).

A natural counterpart of Edge-Disjoint Paths is Edge Cut. Therein, the question is whether there is a set F of at most β edges such that there is no s-t-path in the graph after removing the edges in F. There is a strong dual relationship between Edge-Disjoint Paths and Edge Cut in the sense that, if both problems admit a solution for a given β, then the value of β is optimal, that is, it is not possible to find $\beta + 1$ edge disjoint s-t-paths and the removal of any set of $\beta - 1$ edges leaves s and t in the same connected component. Consequently, since Edge Cut can be solved in polynomial time, so can Edge-Disjoint Paths. Quite naturally, there are many variants of the above described network flow/cut problems such as e. g. multicommodity flows, unsplittable flows, and the related cut problems (e. g. Schrijver [Sch03] provides further examples and formal definitions). Unlike Edge-Disjoint Paths and Edge Cut, it is not always the case that the respective flow and the cut problem belong to the same complexity class. We investigate a variant of Edge Cut called Length-Bounded Cut. It originates from network design and telecommunications and Gouveia et al. [GPS08], Huygens and Ridha Mahjoub [HR07], and Huygens et al. [Huy+07] describe further applications. Length-Bounded Cut is an example where the cut problem is harder than the respective flow problem. Length-Bounded Cut is *NP*-hard [Bai+10] while the respective flow problem is polynomial-time solvable [MM10].

Our main contribution in this chapter is a dynamic-programming-based polynomial-time algorithm for Length-Bounded Cut on proper interval graphs. This confirms a conjecture by Bazgan et al. [Baz+19]. We conclude this chapter with showing some limitations of our approach when trying to adapt it for interval graphs. The existence of a polynomial-time algorithm for Length-Bounded Cut on interval graphs was also posed as an open problem by Bazgan et al. [Baz+19].

4.1 Problem Definition and Related Work

In this chapter, we study Length-Bounded Cut, which is the cut problem related to the variant of Edge-Disjoint Paths where an additional bound λ is given and the sought collection of s-t-paths can only contain paths of length at most λ. This problem has been introduced by Adámek and Koubek [AK71] and is formally defined as follows.

Length-Bounded Cut
Input: An undirected graph $G := (V, E)$, two vertices s, t, and two positive integers β, λ.
Question: Is there a subset $F \subseteq E$ with $|F| \leq \beta$ such that there is no s-t-path of length at most λ in $G' := (V, E \setminus F)$?

An example of Length-Bounded Cut is given in Figure 4.1. For $\lambda = |V|$ one is left with the original problem Edge Cut which is polynomial-time-solvable. Length-Bounded Cut is also solvable in polynomial time if $\lambda \leq 3$ [MM10]. However, Baier et al. [Bai+10] showed that Length-Bounded Cut is NP-hard

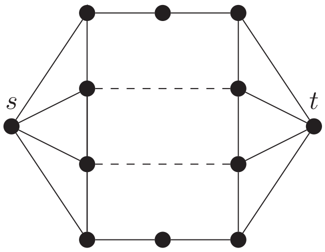


Figure 4.1: An example graph. The dashed edges form a solution for Length-Bounded Cut with $\beta = 2$ and $\lambda = 3$.

for $\lambda = 4$. The related flow problem Length-Bounded Flow, where we restrict the flow to paths of length at most λ, can be solved in polynomial time via a reduction to linear programming [Bai+10, KS06, MM10].

We note that the result of Baier et al. [Bai+10] in fact gives NP-hardness for Length-Bounded Cut for each constant $\lambda \geq 4$. Thus, in order to obtain tractability results, one presumably has to either consider a different parameterization or combine λ with some other parameter. Golovach and Thilikos [GT11] first studied Length-Bounded Cut from the viewpoint of parameterized complexity. They showed that Length-Bounded Cut is fixed-parameter tractable for the combined parameter $\beta + \lambda$. It is worth noting that the parameter β alone gives *W[1]*-hardness [GT11]. Later, Fluschnik et al. [Flu+18] proved that it is unlikely that a polynomial kernel in $\beta + \lambda$ exists. Dvořák and Knop [DK18] considered structural parameters for Length-Bounded Cut. They showed that it is *W[1]*-hard when parameterized by the pathwidth of the input graph while it is fixed-parameter tractable when parameterized by the treedepth of the input graph. Kolman [Kol18] gave an $O(\lambda^{\tau} \cdot (n + m))$-time algorithm for Length-Bounded Cut, where τ is the treewidth of G. Furthermore, Length-Bounded Cut is fixed-parameter tractable for the parameter λ if G is planar [Kol18] (it remains NP-complete on planar graphs [Flu+18]). Bazgan et al. [Baz+19] studied both restrictions on special graph classes as well as structural parameterizations for Length-Bounded Cut. They provided an *XP*-algorithm for the maximum degree of the input graph G and fixed-parameter tractability for the feedback edge number. Furthermore, they presented a polynomial-time algorithm for co-graphs while showing *NP*-completeness even if the input is restricted to bipartite graphs or split graphs. Finally, Length-Bounded Cut is *W[1]*-hard with respect to the combined parameter pathwidth and maximum degree and with respect to the feedback vertex number [BHK20].

4.2 Polynomial-Time Algorithm for Proper Interval Graphs

In this section, we present a polynomial-time algorithm for Length-Bounded Cut on proper interval graphs. To this end, for each vertex $v \notin \{s, t\}$, we define a set of vertices that contains v and t. The algorithm for Length-Bounded Cut on proper interval graphs is then a dynamic program that stores for each vertex v and each possible distance d ($2 \leq d \leq \lambda$) the minimum size of a cut that makes each vertex in the described set have distance at least d from s.

Recall that each vertex v in a proper interval graph can be represented by an interval $[b_v, f_v]^{\mathbb{Q}}$ such that two vertices u, w are adjacent in G if and only if $[b_u, f_u]^{\mathbb{Q}} \cap [b_w, f_w]^{\mathbb{Q}} \neq \emptyset$ and no interval representing a vertex is properly contained in the interval representing another vertex. Observe that we can assume without loss of generality that $b_s \leq b_t$ as we can otherwise "mirror" the graph by setting $b_v = -f_v$ and $f_v = -b_v$ for each vertex $v \in V$. It is folklore that one can assume that $|\{b_v \mid v \in V\}| = |V|$. We further assume that the vertices in $V \setminus \{s, t\}$ are named $v_1, v_2, \ldots, v_{n-2}$ such that $b_{v_i} < b_{v_j}$ for all $i < j$. We first show that we can safely ignore all vertices v with $f_v < b_s$ or $f_t < b_v$. It is worth noting that the following lemma holds for interval graphs and not only for proper interval graphs.

Lemma 4.1. *Let $I = (G = (V, E), s, t, \beta, \lambda)$ be an instance of Length-Bounded Cut where G is an interval graph and $b_s < b_t$ in the interval representation. Let $L := \{u \in V \mid f_u < b_s\}$ and $R := \{u \in V \mid f_t < b_u\}$. Then, $I' := (G-(L \cup R), s, t, \beta, \lambda)$ is an equivalent instance of Length-Bounded Cut.*

Proof. Let $I, I', G, s, t, \beta, \lambda, L,$ and R be as defined above. We first show that $I_L = (G - R, s, t, \beta, \lambda)$ is an equivalent instance. Note that $s, t \notin L \cup R$ and hence I_L and I' are instances of Length-Bounded Cut. Note further that deleting vertices from any input graph cannot decrease the distance between any pair of vertices and hence if I is a yes-instance, then so are I_L and I'. Hence it remains to show that if I_L is a yes-instance, then so is I.

Assume towards a contradiction that I_L is a yes-instance and I is a no-instance. Then there is a set F_L of β edges in $G - R$ such that the distance between s

and t in $G_L := (V \setminus R, E \setminus (F_L \cup \{\{u,v\} \in E \mid u \in R\}))$ is at least $\lambda + 1$. Since I is a no-instance, there is a path P of length at most λ between s and t in $G^* := (V, E \setminus F_L)$. As G_L and G^* only differ in R, each path of length at most λ between s and t in G^* contains at least one vertex from R. We will show that $\deg_G(t) \leq |F_L|$ and hence there is an s-t-cut of size at most β in G and thus I is a yes-instance. This contradicts the assumption that I is a no-instance and hence finishes the proof that I_L is equivalent to I.

We start by giving some basic notation for the proof to come. We use sets of vertices that have a certain distance from s in some subgraph H of G. To this end, we define $X_H^p := \{u \in V \mid \mathrm{dist}_H(s,u) = p\}$ for each distance p. Analogously, we define $X_H^{\leq p} := \{u \in V \mid \mathrm{dist}_H(s,u) \leq p\}$ and $X_H^{\geq p} := \{u \in V \mid \mathrm{dist}_H(s,u) \geq p\}$.

Let $d := \mathrm{dist}_{G^*}(s,t)$ and let t' be the vertex in P with maximum $b_{t'}$. Since P contains a vertex from R, it holds that $b_{t'} > f_t$ and hence $t' \notin N_G(t)$. Since t' is on a shortest s-t-path in G^* and $t' \notin N_G(t)$, it holds that $t' \in X_{G^*}^{\leq d-2}$. Now consider the set K of vertices that are part of a shortest s-t'-path in G^* and that are neighbors of t in G. By construction $K \subseteq X_{G^*}^{\leq d-3}$ and for each $y \in [b_t, f_t]^{\mathbb{Q}}$ there is a vertex $v \in K$ with $y \in [b_v, f_v]^{\mathbb{Q}}$. We next show that $|F_L| \geq \deg_G(t)$. To this end, consider any vertex $u \in N_G(t)$. If $u \in X_{G^*}^{\leq d-2}$, then it holds that $\{u,t\} \in F_L$. Otherwise $u \in X_{G^*}^{\geq d-1}$. Observe that for each $u \in N_G(t)$ it holds by definition that there is a $y \in [b_u, f_u]^{\mathbb{Q}} \cap [b_t, f_t]^{\mathbb{Q}} \neq \emptyset$ and hence there is a vertex $v \in K$ with $y \in [b_v, f_v]^{\mathbb{Q}}$ and hence $\{u,v\} \in E$. Note that $u \in X_{G^*}^{\geq d-1}$ and $v \in K \subseteq X_{G^*}^{\leq d-3}$. Since

$$\mathrm{dist}_{G^*}(s,u) \geq d-1 > d-3+1 \geq \mathrm{dist}_{G^*}(s,v) + 1,$$

it holds that $\{u,v\} \in F_L$. Since $\{v,t\} \in F_L$ for all $v \in N_G(t) \cap X_{G^*}^{\leq d-2}$, since for all $v \in N_G(t) \cap X_{G^*}^{\geq d-1}$ there is some $w \in K$ such that $\{v,w\} \in F_L$, and since $K \cap X_{G^*}^{\geq d-1} = \emptyset$, there is a unique edge for each $v \in N_G(v)$ in F_L. Hence, $\beta = |F_L| \geq \deg_G(t)$ and thus there is a trivial s-t-cut of size β in G that contains all edges incident to t. Thus, I is a yes-instance.

We conclude the proof by showing that I' is equivalent to I_L. Note that we consider undirected graphs and hence we can exchange the roles of s and t and *mirror* the graph by setting $b'_v := -f_v$ and $f'_v := -b_v$ for each vertex $v \in V$. Note that all vertices in L (originally fulfilling $f_u < b_s$) now satisfy $b'_u > f'_s$. Hence, if we interchange the names of s and t, then they satisfy the condition of R and hence we can use the argument above to show that I' and I_L are equivalent. $\square$

Using Lemma 4.1, we can always assume that there is no vertex v with $f_v < b_s$ or $b_v > f_t$. We next show that if there is a solution, then there is also a solution in which the distance from s to v_j is non-decreasing in j.

Lemma 4.2. *Let $G = (V, E)$ be a proper interval graph where no vertex v satisfies $f_t < b_v$ or $b_s > f_v$ and let F be a set of edges. Let d be the distance from s to t in $G' := (V, E \setminus F)$. Then, there is a set F' of edges with $|F'| \leq |F|$ such that $\operatorname{dist}_{G''}(s, t) \geq d$ in $G'' := (V, E \setminus F')$ and $\operatorname{dist}_{G''}(s, v_i) \leq \operatorname{dist}_{G''}(s, v_j)$ for each $v_i, v_j \in V \setminus \{s, t\}$ with $b_{v_i} < b_{v_j}$.*

Proof. Let G, s, t, F, G', and d be as defined above. The main idea of this proof is to construct a sequence of graphs which starts with the graph G' and ends with the sought graph G''. To this end, we define for each vertex $v \in V$ in a graph $H = (V, E_H)$ a specific distance $D_H(v)$. We define $D_H(v)$ to be the length of a shortest path $P = (s = u_0, u_1, u_2, \dots, u_\alpha = v)$ from s to v in H such that for all $\gamma \in [\alpha - 1]$ it holds that $b_{u_\gamma} < b_{u_{\gamma+1}}$. As a special case, if $u_\alpha = t$, then we only require that for all $\gamma \in [\alpha - 2]$ it holds that $b_{u_\gamma} < b_{u_{\gamma+1}}$. We call such paths monotone, and if no monotone s-v-path exists, then we set $D_H(v) := \infty$. Observe that for each graph H it holds that $D_H(s) = 0$ and $D_H(v) \geq \operatorname{dist}_H(s, v)$. Let $\mathcal{G} := \{G^* := (V, E^*) \mid E^* \subseteq E \wedge |E^*| \geq |E \setminus F|\}$. We present a sequence of graphs $(G' := G_1, G_2, \dots G_k)$ such that

(1) $G_\ell = (V, E_\ell) \in \mathcal{G}$ for each $\ell \in [k]$,

(2) $D_{G_\ell}(t) \leq D_{G_{\ell+1}}(t)$ for each $\ell \in [k - 1]$, and

(3) $D_{G_k}(v) \leq D_{G_k}(w)$ for all $v, w \in V \setminus \{s, t\}$ with $b_v < b_w$.

Claim 4.3. *If such a sequence of graphs exists, then $F_k = E \setminus E_k$ and $G'' := G_k$ satisfy Lemma 4.2.*

Proof of Claim 4.3. First, we show that $\operatorname{dist}_{G_k}(s, v) = D_{G_k}(v)$ for all v. Assume towards a contradiction that there is a vertex $v \neq t$ with $\operatorname{dist}_{G_k}(s, v) \neq D_{G_k}(v)$. Consider any shortest s-v-path P in G_k. Let w be the first vertex on P with $\operatorname{dist}_{G_k}(s, w) \neq D_{G_k}(w)$ and let w' be its predecessor. By this definition, it holds that $\operatorname{dist}_{G_k}(s, w') = D_{G_k}(w')$ and $b_w < b_{w'}$ as otherwise $\operatorname{dist}_{G_k}(s, w) = D_{G_k}(w)$. Since $D_{G_k}(w) \geq \operatorname{dist}_{G_k}(w)$ and $D_{G_k}(w) \neq \operatorname{dist}_{G_k}(s, w)$, it follows that

$$D_{G_k}(w) > \operatorname{dist}_{G_k}(s, w) = \operatorname{dist}_{G_k}(s, w') + 1 = D_{G_k}(w') + 1,$$

a contradiction to (3) and $b_w < b_{w'}$.

Now assume that $\mathrm{dist}_{G_k}(s,t) \neq D_{G_k}(t)$. Let P be a shortest s-t-path in G_k. Let v be the predecessor of t in P. We have shown that $\mathrm{dist}_{G_k}(v) = D_{G_k}(v)$ and hence $\mathrm{dist}_{G_k}(t) = \mathrm{dist}_{G_k}(v) + 1 = D_{G_k}(v) + 1 = D_{G_k}(t)$. The last step follows from the fact that $D_{G_k}(t) \leq D_{G_k}(v) + 1$ as v is a neighbor of t in G_k and the special case in the definition of D that allows to ignore b_t.

The claim now easily follows. Note that (1) ensures that $G_k \in \mathcal{G}$ and hence $|F_k| \leq F$. It follows from (2) that

$$\mathrm{dist}_{G_k}(s,t) = D_{G_k}(t) \geq D_{G_{k-1}}(t) \geq \ldots \geq D_{G_1}(t) = D_{G'}(t) \geq \mathrm{dist}_{G'}(s,t) \geq d.$$

Finally, (3) states that for all $v, w \in V \setminus \{s,t\}$ with $b_v < b_w$ that

$$\mathrm{dist}_{G''}(s,v) = D_{G''}(v) \leq D_{G''}(w) = \mathrm{dist}_{G''}(s,w). \qquad \diamond$$

We now describe how to obtain the sequence $(G' = G_1, G_2, \ldots, G_k)$ of graphs. To this end, we need a rather technical order over the graphs in $\mathcal{G}$. We say that $(V, E_\alpha) = G_\alpha <_\Delta G_\gamma = (V, E_\gamma)$ for $G_\alpha, G_\gamma \in \mathcal{G}$ if and only if

- $|E_\alpha| > |E_\gamma|$,
- $|E_\alpha| = |E_\gamma|$ and there exists a $v \in V \setminus \{t\}$ such that $D_{G_\alpha}(v) < D_{G_\gamma}(v)$ and $D_{G_\alpha}(w) = D_{G_\gamma}(w)$ for all $w \in V \setminus \{t\}$ with $b_w < b_v$, or
- $|E_\alpha| = |E_\gamma|$, $D_{G_\alpha}(v) = D_{G_\gamma}(v)$ for all $v \in V \setminus \{t\}$, and $D_{G_\alpha}(t) < D_{G_\gamma}(t)$.

Notice that $<_\Delta$ defines a total preorder on $\mathcal{G}$, that is, the order $<_\Delta$ is transitive, reflexive, and for each two graphs $G_\alpha, G_\beta \in \mathcal{G}$ with $G_\alpha \neq G_\beta$ it holds that $G_\alpha <_\Delta G_\beta$ or $G_\beta <_\Delta G_\alpha$.

Let G_ℓ be a graph in the sequence. We will guarantee that each graph in the sequence fulfills (1) and (2). Consequently, if G_ℓ satisfies (3), then we have found the last graph in the sequence. Otherwise, we describe how to obtain another graph $G_{\ell+1} \in \mathcal{G}$ such that (2) holds for G_ℓ and $G_{\ell+1}$ and $G_{\ell+1} <_\Delta G_\ell$. Since $<_\Delta$ is a total preorder, we can only build a finite sequence and hence at some point a graph has to satisfy (3).

Since $G_\ell = (V, E_\ell)$ does not satisfy (3), there is some minimum j such that $D_{G_\ell}(v_j) > D_{G_\ell}(v_{j+1})$. Let

$$X := \{x \in N_G(v_{j+1}) \mid (b_x < b_{v_j} \lor x = s) \land \{v_j, x\} \in E \setminus E_\ell \land \{v_{j+1}, x\} \in E_\ell\},$$

$$Y := \{y \in N_G(v_j) \mid (b_y > b_{v_{j+1}} \lor y = t) \land \{v_{j+1}, y\} \in E \setminus E_\ell \land \{v_j, y\} \in E_\ell\}.$$

See Figure 4.2 for an example of X and Y. We distinguish between the two cases $|X| \geq |Y|$ and $|X| < |Y|$.

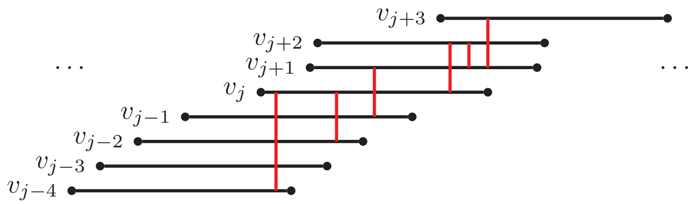


Figure 4.2: An example for X and Y. Red (vertical) edges are contained in $E \setminus E_\ell$. For the sake of readability we do not depict edges in E_ℓ. Note that $X = \{v_{j-2}\}$ and $Y = \{v_{j+3}\}$.

Case 1 ($|X| \geq |Y|$): Let

$$E_{\ell+1} := (E_\ell \setminus \{\{v_j, y\} \mid y \in Y\}) \cup \{\{v_j, x\} \mid x \in X\},$$

and $G_{\ell+1} := (V, E_{\ell+1})$. Since $|X| \geq |Y|$, $X \cap Y = \emptyset$, and $G_\ell \in \mathcal{G}$, it holds that $|E_{\ell+1}| \geq |E_\ell| \geq |E \setminus F|$ and thus $G_{\ell+1} \in \mathcal{G}$. Clearly, for all $v \in V \setminus \{t\}$ with $b_v < b_{v_j}$, we have $D_{G_{\ell+1}}(v) = D_{G_\ell}(v)$ as $E_{\ell+1}$ and E_ℓ only differ in edges incident to v_j. Let w be the predecessor of v_{j+1} in a shortest monotone s-v_{j+1}-path in G_ℓ. Since $D_{G_\ell}(v_j) > D_{G_\ell}(v_{j+1})$ it holds that $w \neq v_j$ and hence $b_w < b_{v_j} < b_{v_{j+1}} \leq f_w$. Moreover, vertex w is contained in X as otherwise $D_{G_\ell}(v_j) \leq D_{G_\ell}(w) + 1 = D_{G_\ell}(v_{j+1})$. Thus,

$$D_{G_{\ell+1}}(v_j) = D_{G_\ell+1}(w) + 1 = D_{G_\ell}(w) + 1 = D_{G_\ell}(v_{j+1}) < D_{G_\ell}(v_j)$$

and combined with $|E_{\ell+1}| \geq |E_\ell|$ this yields $G_{\ell+1} <_\Delta G_\ell$.

It remains to show that $D_{G_\ell}(t) \leq D_{G_{\ell+1}}(t)$. Consider a shortest monotone s-t-path P in $G_{\ell+1}$. If P does not pass through v_j, then it is also a monotone s-t-path in G_ℓ and hence $D_{G_\ell}(t) \leq D_{G_{\ell+1}}(t)$. If P passes through v_j, then let z be the successor of v_j in P. Note that if $z \in Y$, then $\{v_{j+1}, z\} \in E_\ell$, and if $z \notin Y$, then $z = v_{j+1}$ or $\{v_{j+1}, z\} \in E_\ell$ as $b_{v_j} \leq b_z$. Hence it holds that

$$D_{G_\ell}(z) \leq D_{G_\ell}(v_{j+1}) + 1 = D_{G_{\ell+1}}(v_j) + 1 = D_{G_\ell+1}(z).$$

Finally, let P' be a shortest monotone s-z-path in G_ℓ and let $P'' = P' \bullet P[z, t]$. Note that P'' is a monotone s-t-path of length at most $D_{G_{\ell+1}}(t)$ in G_ℓ and thus $D_{G_\ell}(t) \leq D_{G_{\ell+1}}(t)$.

Case 2 ($|X| < |Y|$): We set

$$E_{\ell+1} := (E_\ell \setminus \{\{v_{j+1}, x\} \mid x \in X\}) \cup \{\{v_{j+1}, y\} \mid y \in Y\}.$$

Since $|X| < |Y|$, $X \cap Y = \emptyset$, and $G_\ell \in \mathcal{G}$, it holds that $|E_{\ell+1}| > |E_\ell|$ and therefore $G_{\ell+1} \in \mathcal{G}$ and $G_{\ell+1} <_\Delta G_\ell$. It remains to show $D_{G_\ell}(t) \leq D_{G_{\ell+1}}(t)$.

Since E_ℓ and $E_{\ell+1}$ only differ in edges incident to v_{j+1}, for all v with $b_v < b_{v_j}$ it holds that $D_{G_{\ell+1}}(v) = D_{G_\ell}(v)$. Let P be a shortest monotone s-v_{j+1}-path in $G_{\ell+1}$ and let w be the predecessor of v_{j+1} in P. By definition of $E_{\ell+1}$ it holds that $w = v_j$ or $\{w, v_j\} \in E_\ell$ and hence $D_{G_\ell}(v_j) \leq D_{G_{\ell+1}}(v_{j+1})$. Let P' be a shortest monotone s-t-path in $G_{\ell+1}$. If P' does not pass through v_{j+1}, then it is also a monotone s-t-path in G_ℓ as E_ℓ and $E_{\ell+1}$ only differ in edges incident to v_{j+1} and hence $D_{G_\ell}(t) \leq D_{G_{\ell+1}}(t)$. If P' passes through v_{j+1}, then let z be the successor of v_{j+1} in P'. Since only edges between v_{j+1} and the vertices in Y are contained in $E_{\ell+1}$ but not in E_ℓ, it holds that $z \in Y$. Hence, it holds that $\{v_j, z\} \in E_\ell$ and thus

$$D_{G_\ell}(z) \leq D_{G_\ell}(v_j) + 1 \leq D_{G_{\ell+1}}(v_{j+1}) + 1 = D_{G_{\ell+1}}(z).$$

Finally, let P'' be a shortest monotone s-v_j-path in G_ℓ and let $P''' = P'' \bullet P[z, t]$. Since P''' is a monotone s-t-path of length at most $D_{G_{\ell+1}}(t)$ in G_ℓ, we obtain $D_{G_{\ell+1}}(t) \geq D_{G_\ell}(t)$.

This concludes the proof as we have shown that the sought sequence of graphs is finite and how to obtain each graph in it from the previous. ☐

Using Lemma 4.2, we now provide the main result of this chapter, that is, a dynamic program that solves Length-Bounded Cut on proper interval graphs in polynomial time. The dynamic program stores for each vertex $v \in V \setminus \{t\}$ and each possible distance d the minimum size of a cut that makes each vertex u with $b_u \geq b_v$ or $u = t$ have distance at least d from s.

Theorem 4.4. *Length-Bounded Cut can be solved in $O(n^2 \cdot m)$ time if the input graph is a proper interval graph.*

Proof. We prove the statement by developing a dynamic program. We first state some general observations and derive from them the main idea behind the dynamic program. We then show how the entries of the table of the dynamic program are computed and how to compute the solution for Length-Bounded Cut from the filled table. We continue with proving the correctness of our algorithm and conclude with analyzing its running time.

We assume that, in the input graph $G := (V, E)$, there is no s-t-cut of size at most β as this cut can be detected in $O(n \cdot m)$ time [FF56] and the answer for Length-Bounded Cut is then always yes. Thus, $\deg_G(s) > \beta$ and $\deg_G(t) > \beta$ as otherwise the set of edges incident to s or t are an s-t-cut of size at most β. Furthermore, by Lemma 4.1, we can assume that there is no vertex v with $f_v < b_s$ or $b_v > f_t$. By Lemma 4.2, we can assume that we search for a solution in which for all $v_i, v_j \in V \setminus \{s, t\}$ with $i < j$ it holds that $\operatorname{dist}(s, v_i) \leq \operatorname{dist}(s, v_j)$. Hence, we construct a table $T \colon V \times \mathbb{N} \to \mathbb{N}$ which stores for each vertex $v_i \in V \setminus \{s, t\}$ and each possible distance $2 \leq d \leq \lambda$ the minimum number of edges that have to be deleted from $G' := (V', E') := G - \{t\}$ to ensure the following. First, $\operatorname{dist}(s, v_k) \leq \operatorname{dist}(s, v_\ell)$ for all $k \leq \ell \leq i$. Second, each vertex $v_j \in V \setminus \{s, t\}$ with $j \geq i$ has distance at least d from s.

We start with showing how to initialize the table T. Note that $v_1, v_2, \ldots, v_{\deg(s)}$ are neighbors of s and $v_{\deg(s)+1}, v_{\deg(s)+2}, \ldots, v_{n-2}$ are not. Hence, to increase the distance of each v_j with $j \geq i$ for some given i to at least two, one has to delete all edges between s and the vertices in $\{v_j \mid i \leq j \leq \deg(s)\}$. Thus, the table is initialized with $T[v_i, 2] = 0$ for all $i > \deg(s)$ and $T[v_i, 2] = \deg(s) - i + 1$ for all $i \leq \deg(s)$. We further initialize $T[v_1, d] = \deg(s)$ for all $d \geq 3$.

We next show how to compute the solution to Length-Bounded Cut once the table T is completely filled. Since we seek a solution F such that in $H := (V, E \setminus F)$ it holds that $\operatorname{dist}_H(s, t) > \lambda$, each vertex $u \in N_H(t)$ has to satisfy $\operatorname{dist}_H(s, u) \geq \lambda$. Note that $\deg_G(t) > \beta$ and hence there is at least one vertex $v \in N_H(t)$. Thus, to compute the solution for Length-Bounded Cut, we iterate over $v \in N_G(t) \setminus \{s\}$ and compute $T[v, \lambda] + |\{u \in N(t) \mid u = s \vee b_u < b_v\}|$. Note that this corresponds to the statement that each neighbor u of t in G has distance at least λ from s or the edge $\{u, t\}$ was removed. Hence, the distance between s and t in the resulting graph is at least $\lambda + 1$. Further, if we take the minimum value over all iterations and compare it to β, then this solves Length-Bounded Cut.

It remains to present the recursive formula for T, to prove the correctness of our dynamic program, and to analyze its running time. We start with showing how to compute T. For the sake of simplicity, we also store, for each table entry $T[v_i, d]$ with $d \geq 2$, in a second table $S[v_i, d]$ the vertex $v_j \in V \setminus \{s, t\}$ with minimum j such that v_j has distance $d - 1$ from s in some solution corresponding to $T[v_i, d]$. We initialize $S[v_i, 2] = v_1$ for all v_i and $S[v_1, d] = v_1$ for all $d \geq 3$. Note that $S[v_i, d] = v_1$ might not represent what we claimed if we seek to remove the edge $\{s, v_1\}$. However, in this case there is no solution as we assume that $\deg(s) > \beta$. For increasing values of $d \geq 3$, we iterate over $2 \leq i \leq n-2$ and compute

$$T[v_i, d] = \min_{j<i}\{T[v_j, d-1] + |C[S[v_j, d-1], v_j, v_i]\}|,$$
$$S[v_i, d] = v_j \text{ with } j = \min\{\arg\min_{j<i}\{T[v_j, d-1] + |C[S[v_j, d-1], v_j, v_i]|\}\},$$

where $C[v_h, v_j, v_i]$ is a function that represents for each triple (v_h, v_j, v_i) of vertices with $h < j < i$ the set of edges between a vertex v_ℓ with $h \leq \ell < j$ and a vertex v_r with $r \geq i$. For technical reasons we exclude s and t here and hence the formal definition is

$$C[v_h, v_j, v_i] \coloneqq \{\{v_\ell, v_r\} \in E \mid h \leq \ell < j < i \leq r\}.$$

The vertex v_h is used to avoid double counting.

We continue by proving that S and T store exactly what they are supposed to. Assume towards a contradiction that there is a vertex v_i and a distance $d \geq 2$ such that $S[v_i, d]$ or $T[v_i, d]$ were computed incorrectly. Then there is also a smallest d such that there is a vertex v_i for which $S[v_i, d]$ or $T[v_i, d]$ are computed incorrectly and we assume that v_i is the vertex with the smallest index i such that $S[v_i, d]$ or $T[v_i, d]$ is computed incorrectly. Since we have already shown that the initialization for $d = 2$ is correct, we focus on the case $d > 2$ and distinguish between the three cases that $S[v_i, d]$ was computed incorrectly, that $T[v_i, d] > c$, or that $T[v_i, d] < c$, where c is the correct value of $T[v_i, d]$.

If $T[v_i, d] < c$, then let v_j be a vertex with $j < i$ that minimizes the sum $T[v_j, d-1] + |C[S[v_j, d-1], v_j, v_i]|$. Since we assume that $T[v_j, d-1]$ is computed correctly (recall that d was chosen to be the minimum value for which S or T was computed incorrectly), there is a set F_1 of $T[v_j, d-1]$ edges such that in the graph $H' \coloneqq (V', E' \setminus F_1)$ it holds that $\text{dist}_{H'}(s, v_r) \geq d-1$ for all $r \geq j$ and $\text{dist}_{H'}(s, v_\ell) \leq \text{dist}_{H'}(s, v_k)$ for all $\ell \leq k \leq j$. Let $v_h = S[v_j, d-1]$. Since $S[v_j, d-1]$ is by assumption computed correctly, it holds for all v_ℓ with $\ell < h$ that $\text{dist}_{H'}(s, v_\ell) \leq d-3$. Thus, F_1 contains all edges between vertices in $\{v_\ell \mid \ell < h\}$ and $\{v_r \mid r \geq i\}$. Since $C[v_h, v_j, v_i]$ is the set of all edges between vertices $v_{\ell'}$ with $h \leq \ell' < j$ to vertices v_r with $r \geq i$, it holds that there is no edge between a vertex of distance at most $d-2$ from s to a vertex v_r with $r \geq i$ in $H \coloneqq (V', E' \setminus (F_1 \cup C[v_h, v_j, v_i]))$. Hence each such vertex v_r is of distance at least d from s in H. It remains to show that $\text{dist}_H(s, v_\ell) \leq \text{dist}_H(s, v_k)$ for all $\ell \leq k \leq i$. Note that H and H' only differ in edges in $C[v_h, v_j, v_i]$, that is, in edges between vertices v_ℓ and v_r with $h \leq \ell \leq j$ and $r \geq i$. Since those v_ℓ have distance $d-2$ and those v_r have distance at least $d-1$ from s in H, it holds that $\text{dist}_H(s, v_\ell) = \text{dist}_H(s, v_\ell)$ for all $\ell \leq i$ and thus also H fulfills $\text{dist}_H(s, v_\ell) \leq \text{dist}_H(s, v_k)$ for all $\ell \leq k \leq i$. Since $T[v_i, d] = |F_1 \cup C[v_h, v_j, v_i]|$ and $F_1 \cap C[v_h, v_j, v_i] = \emptyset$, it holds that

$$T[v_i, d] = |F_1| + |C[v_h, v_j, v_i]| = T[v_j, d-1] + |C[S[v_j, d-1], v_j, v_i]|,$$

and thus $T[v_i, d] \geq c$, a contradiction.

If $T[v_i, d] > c$, then there is a cut F' that contains less than $T[v_i, d]$ edges such that in the respective graph $H' := (V', E' \setminus F')$ all vertices v_r with $r \geq i$ have distance at least d from s and $\text{dist}_{H'}(s, v_k) \leq \text{dist}_{H'}(s, v_\ell)$ for all $k \leq \ell \leq i$. Then, there is a vertex v_j such that v_j and all vertices v_r with $r \geq j$ have distance at least $d-1$ from s in H' and all vertices v_ℓ with $\ell < j$ have distance at most $d-2$ from s in H'. Hence, F' has to contain all edges in $F'' := \{\{v_\ell, v_r\} \in E \mid \ell < j < i \leq r\}$ as otherwise a vertex v_r with $r \geq i$ would have distance at most $d-1$ from s in H'. Let $v_h := S[v_j, d-1]$. We partition the set F'' into two disjoint sets $F''_L := \{\{v_\ell, v_r\} \in E \mid \ell < h < i \leq r\}$ and $F''_R := \{\{v_\ell, v_r\} \in E \mid h \leq \ell < j < i \leq r\}$. Let $H := (V', E' \setminus (F' \setminus F''_R))$. Notice that H and H' only differ in edges in F''_R, that is, in edges incident to vertices v_ℓ and v_r with $h \leq \ell < j$ and $r \geq i$. Since those v_ℓ have distance $d-2$ and those v_r have distance at least $d-1$ from s in H, the distance between s and vertices v_ℓ with $\ell < j$ is the same in H' and H. Hence, $\text{dist}_H(s, v_k) \leq \text{dist}_H(s, v_\ell)$ for all $k \leq \ell \leq j$. Thus, it holds that $T[v_j, d-1] \leq |F' \setminus F''_R|$ as $T[v_j, d-1]$ was computed correctly by assumption. Note further that F''_R is by definition equal to $C[S[v_j, d-1], v_i, v_j]$ and

$$T[v_j, d-1] + |C[v_h, v_j, v_i]| \geq \min_{j' \leq i}\{T[v'_j, d-1] + |C[S[v'_j, d-1], v'_j, v_i]|\}.$$

Thus, $c = |F'| = |F' \setminus F''_R| + |F''_R| \geq T[v_j, d-1] + |C[v_h, v_j, v_i]| \geq T[v_i, d]$, a contradiction.

Finally, assume towards a contradiction that $S[v_i, d]$ is computed incorrectly but $T[v_i, d]$ is computed correctly. Since $T[v_i, d]$ is computed correctly, there is a set F' of $T[v_i, d]$ edges such that in $H = (V', E' \setminus F')$ it holds for all $k \leq \ell \leq i$ that $\text{dist}_H(s, v_k) \leq \text{dist}_H(s, v_\ell)$ and for all $j \geq i$ that $\text{dist}_H(s, v_j) \geq d$. Then, there is a vertex v_j such that $\text{dist}(s, v_\ell) \leq d-2$ for all $\ell < j$ and $\text{dist}(s, v_r) \geq d-1$ for all $r \geq j$. Let, without loss of generality, F' be a set of edges such that there is no edge set F'' with the same property as described above where the respective vertex $v_{j'}$ satisfies $j' < j$. Let $S[v_i, d] = v_h$. We show that $h = j$. If $j < h$, then $T[v_h, d-1] + |C[S[v_h, d-1], v_h, v_i]| < T[v_i, d]$ as otherwise h would have been chosen smaller. This, however, contradicts the assumption that $T[v_i, d]$ is computed correctly. If $j > h$, then by definition, $T[v_i, d] = T[v_h, d-1] + |C[S[v_h, d-1], v_h, v_i]|$. Thus, there is a set F'' such that $|F''| = T[v_i, d] = |F'|$ and in $H' := (V', E' \setminus F'')$ it holds for all $k \leq \ell \leq i$ that $\text{dist}_{H'}(s, v_k) \leq \text{dist}_{H'}(s, v_\ell)$. Moreover, it holds for all $j \geq i$ that $\text{dist}_{H'}(s, v_j) \geq d$, for all $\ell < h$ that $\text{dist}_{H'}(s, v_\ell) \leq d - 2$, and for all $r \geq h$ that $\text{dist}(s, v_r) \geq d - 1$. This contradicts the definition of F'.

We conclude this prove with analyzing the running time of our algorithm. We first show how to compute $C[v_h, v_i, v_j]$ for all triples (v_h, v_j, v_i) of vertices in $O(n^2 \cdot m)$ time. To this end, we first compute a tables $A[v_j, v_i]$, where

$$A[v_j, v_i] := |\{\{v_\ell, v_r\} \in E \mid \ell < j < i \leq r\}|.$$

Note that A can be computed in $O(n^2 \cdot m)$ time by iterating over all edges $\{v_\ell, v_r\}$ (we assume $\ell < r$) and all entries in $A[v_j, v_i]$ and if $\ell < j < i \leq r$, then increment the entry. Once A is computed, we compute $C[v_h, v_j, v_i] := A[v_j, v_i] - A[v_h, v_i]$ in constant time per table entry. Since there are $O(n^3)$ table entries, the overall running time for this preprocessing is $O(n^2 \cdot m)$ (note that the input graph is a connected interval graph and hence $O(n) \subseteq O(m)$).

Each table entry $S[v_i, d]$ and $T[v_i, d]$ can be computed in $O(n)$ time by iterating over at most n vertices and computing the sum of a table entry in T and the size of a table entry in C, thereby keeping track of the minimum value and which iteration led to this minimum. Since there are $O(n \cdot \lambda)$ table entries, the overall running time is $O(n^2 \cdot \lambda)$. As we may assume that $\lambda < n$ (each path has length at most n), the running time is bounded by $O(n^3)$. Lastly, computing the solution takes $O(n)$ time as we have to iterate over up to n neighbors v_i of t and for each we have to compute $|\{v_\ell \mid \ell < i \wedge \{v_\ell, t\} \in E\}|$. This computation takes constant time as we can compute the smallest index j of a vertex that is adjacent to t in G and then compute $i - j + 1$. Thus, the overall running time for our algorithm is $O(n^2 \cdot m)$. □

The main point in the proof of Theorem 4.4 where we need to assume that the input graph is a proper interval graph and not an interval graph is the application of Lemma 4.2. In the following section, we will investigate problems arising when trying to adapt Lemma 4.2 for interval graphs. Concluding this section, we want to emphasis the way the solution is computed in the proof of Theorem 4.4 once the tables T and S are completely filled. Rather than looking at a single entry or taking the maximum or minimum entry in a given column, we iterate over all entries with $d = \lambda$ in $T[v_i, d]$ and add to it the number of neighbors v_j with $j < i$ of t. The solution is then corresponding to the minimum such sum. This way of finding a solution goes to show that each of the four guiding questions (even the one that looks the simplest) can have a surprising or non-trivial answer.

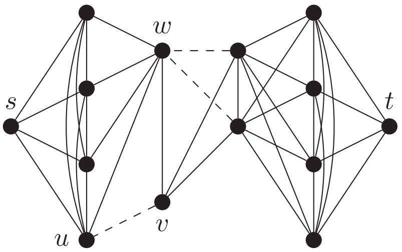
4.3 Falsifying Assumptions for Interval Graphs

In this section, we discuss some problems that arise when trying to adapt the algorithm behind Theorem 4.4 for interval graphs. The only difference between interval graphs and proper interval graphs is that in interval graphs there can be pairs (v, w) of vertices such that $N[v] \subset N[w]$. Intuitively, it does not seem to make sense to remove an edge $\{u, v\}$ while leaving an edge $\{u, w\}$ in the solution graph as each shortest s-t-path containing v and using the edge $\{u, v\}$ can then be replaced by a path containing w and $\{u, w\}$. This leads to the following conjecture.

Conjecture 4.5. *Let $G = (V, E)$ be an interval graph and let F be a set of edges. Let d be the distance from s to t in $G' = (V, E \setminus F)$. Then, there is a set F' of edges with $|F'| \leq |F|$ such that for $G'' := (V, E \setminus F')$ it holds that $\operatorname{dist}_{G''}(s, t) \geq d$ and for each $v, w \in V \setminus \{s, t\}$ with $N[v] \subset N[w]$ it holds that if $\{u, v\} \in F'$ for some $u \in V$, then also $\{u, w\} \in F'$.*

Conjecture 4.5 would be helpful to show that Lemma 4.2 also holds for interval graphs. Unfortunately, Conjecture 4.5 is false as shown in the example in Figure 4.3. Therein, the only solution for removing three edges deletes some edges incident to w and one edge incident to v such that the only remaining path between s and t passes through both v and w. A next natural conjecture could be that a similar approach to the dynamic program behind Theorem 4.4 could still work, where we order the vertices by their b- or their f-values.

Conjecture 4.6. *Let $G = (V, E)$ be an interval graph and let F be a set of edges. Let d be the distance from s to t in $G' := (V, E \setminus F)$. Let F_b and F_f be sets of minimum size such that $G_b := (V, E \setminus F_b)$ and $G_f := (V, E \setminus F_f)$ fulfill $\operatorname{dist}_{G_b}(s, t) \geq d$, $\operatorname{dist}_{G_f}(s, t) \geq d$, and the following. For each $v, w \in V \setminus \{s, t\}$ it holds that if $b_v < b_w$, then $\operatorname{dist}_{G_b}(s, v) \leq \operatorname{dist}_{G_b}(s, w)$. Moreover, if $f_v < f_w$, then $\operatorname{dist}_{G_f}(s, v) \leq \operatorname{dist}_{G_f}(s, w)$. Then, $|F_b| \leq |F|$ or $|F_f| \leq |F|$.*



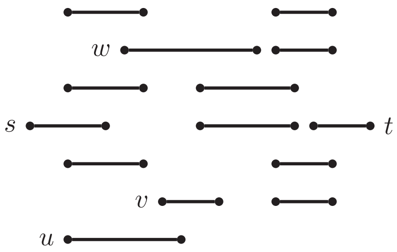
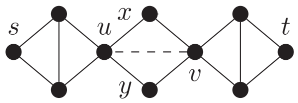


Figure 4.3: An interval graph and its interval representation. The dashed edges show the unique solution for Length-Bounded Cut with $\beta = 3$ and $\lambda = 5$. Note that $N[v] \subset N[w]$, that the edge $\{u, v\}$ is dashed, and that the edge $\{u, w\}$ is not. Since the dashed edges are the only solution, Conjecture 4.5 is false.



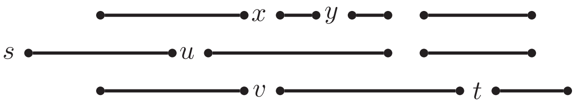


Figure 4.4: An interval graph and its interval representation. The dashed edge is the unique solution for Length-Bounded Cut with $\beta = 1$ and $\lambda = 5$. Let G' denote the graph without the dashed edge. Note that it holds in G' that $\text{dist}_{G'}(s, u) < \text{dist}_{G'}(s, x) = \text{dist}_{G'}(s, y) < \text{dist}_{G'}(s, v)$ but $b_v < b_y$ and $f_u < f_x$. Note further that the dashed edge is the only edge whose removal increases the distance between s and t and hence Conjecture 4.6 is false.

Again, Conjecture 4.6 is false as shown in Figure 4.4. The idea behind this counterexample is to include short intervals with only two neighbors that prevent any reordering of their neighbors after the removal of some edges. This shows that the basic idea of our algorithm for proper interval graphs cannot work for interval graphs as we cannot order the vertices by their b- or their f-values for the dynamic program. Moreover, note that the dashed edge in Figure 4.4 is the only solution and that after removing this edge, the resulting graph contains a C_4 induced by the vertices u, x, v and y. Thus, we cannot even assume that removing a solution from the input interval graph yields an interval graph. This is in contrast to Theorem 4.4 where the graph resulting from removing a solution from the input proper interval graph is again a proper interval graph.

4.4 Concluding Remarks

In this chapter, we studied Length-Bounded Cut in the special case where the input graph is a proper interval graph and showed polynomial-time solvability. This confirms a conjecture by Bazgan et al. [Baz+19]. A natural next step is to investigate interval graphs. We showed some limitations for adapting our approach from proper interval graphs to interval graphs. We still conjecture that Length-Bounded Cut on interval graphs should allow for a polynomial-time algorithm.

Bazgan et al. [Baz+19] provide a hierarchy of parameters with known results and open problems for Length-Bounded Cut. In the paper on which this chapter is based, we solved some of their open problems [BHK20]. Tackling the remaining ones is left as a challenge for future research.

Chapter 5

Disjoint Shortest Paths

This is the final chapter in the dynamic-programming part of the thesis. With regards to content, our main contribution in this chapter is an *XP*-algorithm for the *NP*-hard k-Disjoint Shortest Paths problem. This is a variant of the fundamental and well-studied combinatorial problem k-Disjoint Path. On a conceptual level, we will complete our journey through the intricacies of dynamic programming. For the dynamic program we develop in this chapter, there is a very simple way of computing each table entry. However, when we consider a natural generalization of our problem, then this way is not feasible any more. Further investigating the generalization yields another way of computing each table entry that will turn out to be much more efficient even for the special case we are mainly interested in.

k-Disjoint Path describes the question of whether there are k pairwise disjoint[1] paths between vertex terminal pairs $(s_i, t_i)_{i \in [k]}$ in a given undirected graph G. Karp [Kar75] showed that the problem is *NP*-hard when k is part of the input. On the positive side, Robertson and Seymour [RS95] provided an algorithm running in $O(n^3)$ time for any constant k. Later, Kawarabayashi et al. [KKR12] improved the running time to $O(n^2)$—again for fixed k. On directed graphs, in contrast, the problem is NP-hard even for $k = 2$ [FHW80]. However, on directed acyclic graphs (DAGs), the problem becomes again polynomial-time solvable for constant k [FHW80].

We study a variant called k-Disjoint Shortest Paths. Therein, all paths in a sought solution have to be shortest paths between the respective terminal pairs. This problem has applications in transportation networks, circuit layout, and circuit routing (see e. g. the work by Kawarabayashi et al. [KKR12] and

[1] Here and in the following this means vertex-disjoint.

references therein) and its complexity for constant k has been a long-standing open problem [Eil98, Fom+19]. Very recently, Lochet [Loc21] settled this question by showing that k-Disjoint Shortest Paths can be solved in $n^{O\left(k^{5^k}\right)}$ time, that is, polynomial time for every constant k. We provide a new approach with a novel geometric perspective that simplifies many arguments and leads to an overall streamlined algorithm with a running time of $O(k \cdot n^{16k \cdot k! + k + 1})$. Notably, k-Disjoint Shortest Paths is *W[1]*-hard with respect to k and, assuming the ETH, there is no $f(k) \cdot n^{o(k)}$-time algorithm for k-Disjoint Shortest Paths [Ben+21]. The asymptotic gap between the lower bound of $n^{o(k)}$ and our upper bound of $n^{O((k+1)!)}$ is, however, still quite large.

We formalize our novel geometric view for k-Disjoint Shortest Paths and provide some structural observations regarding solutions to k-Disjoint Shortest Paths in Section 5.2. In Section 5.3, we present a dynamic-programming-based approach to solve a special case of k-Disjoint Shortest Paths. Afterwards, we provide our algorithm for the general problem that uses the dynamic program as a subprocedure and prove our main theorem.

5.1 Problem Definition and Related Work

k-Disjoint Shortest Paths is defined as follows.

k-Disjoint Shortest Paths
Input: An undirected graph $G = (V, E)$ and k pairs $(s_i, t_i)_{i \in [k]}$ of vertices.
Question: Are there k disjoint paths P_i such that, for each $i \in [k]$, P_i is a shortest s_i-t_i-path?

Eilam-Tzoreff [Eil98] introduced this variant of k-Disjoint Path, showed that it is *NP*-hard when k is part of the input, and provided a dynamic-programming-based $O(n^8)$-time algorithm for 2-Disjoint Shortest Paths. This was later improved to an $O(n^2 m)$-time algorithm [Ben+21]. The $O(n^8)$-time algorithm for 2-Disjoint Shortest Paths works for positive edge lengths and, recently, Gottschau et al. [GKW19] and Kobayashi and Sako [KS19] independently extended this result by providing polynomial-time algorithms for the case where the edge lengths are non-negative. Concerning directed graphs, Bérczi and Kobayashi [BK17] provided a polynomial-time algorithm for positive edge lengths for 2-Disjoint Shortest Paths. Note that setting all edge length to zero results in 2-Disjoint Path on directed graphs, which is *NP*-hard [FHW80]. Extending the problem to finding two disjoint s_i-t_i-paths of minimal total length (in undirected graphs), Björklund and Husfeldt [BH19] provided an $O(n^{11})$-time randomized algorithm. Finally, Tragoudas and Varol [TV96] showed that it is *NP*-hard to decide whether the number of solutions of an instance of 2-Disjoint Paths is at least some given threshold.

5.2 A Geometric View on Shortest Paths

In this section, we delineate our geometric perspective on k-Disjoint Shortest Paths, make some structural observations, and give a characterization of solutions with regard to their geometry. In the following sections, we will then use these observations to design a dynamic-programming-based algorithm for k-Disjoint Shortest Paths. We start with some basic intuition and a small example. In Subsection 5.2.1 we then formalize the geometry-based ideas and provide a characterization of solutions for 2-Disjoint Shortest Paths. In Subsection 5.2.2 we then generalize this characterization to solutions of k-Disjoint Shortest Paths.

For the geometric representation, we define a k-dimensional vector[2] $\vec{v}$ for each vertex v. The i^{th} coordinate of this vector is the distance between s_i and v. An example of this vector representation is given in Figure 5.1. Note that there can be multiple vertices with the same vector. The geometric perspective is based on the following two observations. First, since each path $P_i = (v_0^i, v_1^i, \ldots, v_{d_i}^i)$ in the sought solution is a shortest s_i-t_i-path, it holds for each $j \in [d_i]$ that $\operatorname{dist}(s_i, v_j^i) = \operatorname{dist}(s_i, v_{j-1}^i) + 1$, that is, the path P_i is strictly monotone in the i^{th} coordinate. We say that paths which are strictly monotone in the i^{th} coordinate have *color* i. Second, for each vertex v_j^i in P_i, it holds that $\operatorname{dist}(s_i, t_i) = \operatorname{dist}(s_i, v_j^i) + \operatorname{dist}(v_j^i, t_i)$. Thus, any vertex w with $\operatorname{dist}(s_i, t_i) \neq \operatorname{dist}(s_i, w) + \operatorname{dist}(w, t_i)$ cannot be part of a shortest s_i-t_i-path. We can formulate this into a necessary (but not sufficient) condition in terms of our geometric perspective as follows. Consider the k-dimensional hyperrectangle that has the vectors of s_i and t_i as two corners and whose sides form an angle of 45° with the coordinate axes. We say that this hyperrectangle is *spanned* by s_i and t_i. The (hyper)rectangle spanned by s_1 and t_1 in the right-hand side of Figure 5.1 is highlighted in gray. We will prove that any vertex whose vector is not within the area of this hyperrectangle cannot be part of a shortest s_i-t_i-path. Moreover, if we consider any vertex v_j^i in P_i and the two hyperrectangles

[2]We use the term vector interchangeably with the point the vector is pointing to.

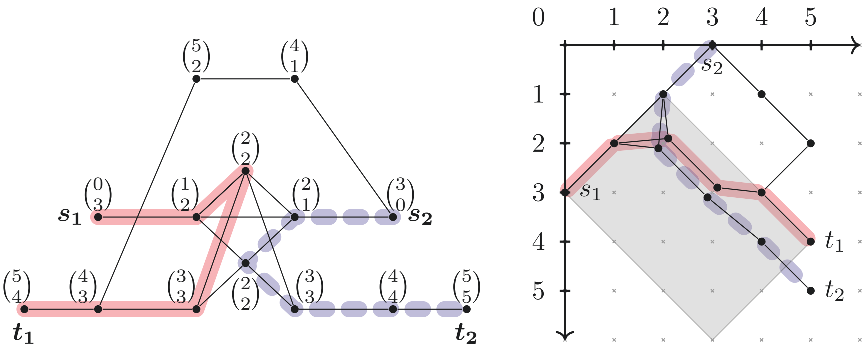


Figure 5.1: *Left side:* A simple, undirected graph with four distinguished vertices s_1, s_2, t_1, and t_2. The vectors with the distances to each s_i are written next to the vertices. Two disjoint shortest paths are highlighted.
Right side: A two-dimensional coordinate system. Each vertex is represented at its vector and edges are drawn as lines between their respective end points. When multiple vertices share the same vector, then vertices are depicted close to their actual vector. The rectangle spanned by s_1 and t_1 is drawn in gray. The two disjoint shortest paths are again depicted. Note that the s_1-t_1-path is going strictly monotone to the right and the s_2-t_2-path is strictly monotone going down.

spanned by v_j^i and either s_i or t_i, then it holds that the vector of each vertex in P_i is contained in the area of these two hyperrectangles.

We use these two observations as follows. Assume that there is a solution (a set of pairwise disjoint shortest s_i-t_i-paths $(P_i)_{i\in[k]}$). We will show that each path P_i can be split into ℓ_i subpaths $P_i^1, P_i^2, \ldots, P_i^{\ell_i}$ such that

1. $\ell_i \leq f(k)$ for all $i \in [k]$ and for some computable function f,

2. the last vertex in P_i^j is the first vertex in P_i^{j+1} for each $i \in [k]$ and each $j \in [\ell - 1]$, and

3. the subpaths can be partitioned such that
   - subpaths in the same part of the partition share a common color and
   - for two subpaths P_i^j and P_p^q in different parts of the partition, it holds that the areas of the hyperrectangles spanned by the end vertices of P_i^j and P_p^q, respectively, are disjoint.

Our algorithm works in two phases. In the first phase, we *guess*[3] the end vertices of each of the described subpaths (we call the end vertices *marbles*). In the second phase, we compute the described partition and solve k-Disjoint Shortest Paths independently for each part of the partition. For each part of the partition, there is a color c that all subpaths in this part have. We assume that each subpath in this part is strictly increasing in its c-coordinates as we can otherwise swap the two endpoints. We then ignore all edges that are not monotone in c (the two endpoints of the edge have the same c-coordinate) and direct the edges so that they are pointing towards the higher c-coordinate. Note that the resulting graph is a DAG and since each subpath is strictly increasing in its c-coordinates, the directed version of each subpath is still contained in the constructed DAG. Hence, we can use the algorithm of Fortune et al. [FHW80] for k-Disjoint Shortest Paths on DAGs to find pairwise disjoint shortest subpaths. We then also present our own dynamic program for k-Disjoint Shortest Paths on DAGs and, as Fortune et al. [FHW80] only state $n^{O(k)}$ time, provide a precise running-time analysis.

We remark that Lochet [Loc21] used the same two-step approach but how these steps are achieved is different. In particular, he does not use a geometric view on shortest paths (as we do). As a result, even for $k = 2$ he can only upper-bound the number of vertices his algorithm has to guess to ensure that no two parts can intersect by 9^{91} ([Loc21, Lemma 13]) while our approach produces at most five parts. Moreover, our geometric view allows us to use a more efficient way of splitting the paths for general k (in $O((k+1)!)$ parts instead of $O(k^{5^k})$ as done by Lochet).

We continue with some intuition for the described subpaths and the partition of them. We start with the two-dimensional case and distinguish between four cases. Figure 5.2 gives an overview over these cases and the vertices (the marbles) we guess in each case. It is easy to see that only in the case in the top right-hand corner the areas of two subpaths intersect (the dashed line). However, in this area both paths are strictly monotone in both coordinates. Thus, the depicted marbles ensure in each case that a partition as described exists.

We continue with the case where $k > 2$. The basic idea is to recursively partition the paths by considering two-dimensional projections of the respective hyperrectangles. Note that if these projections are disjoint, then also the areas

[3]Whenever we pretend to guess something, we mean that we iterate over all possible choices and consider for the explanation or proof the respective correct iteration.

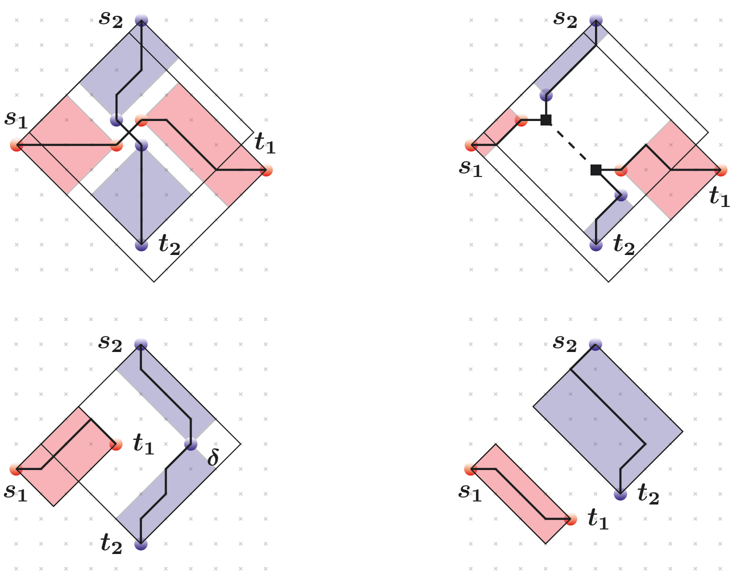


Figure 5.2: The four cases for the two-dimensional projection of two paths P_1 and P_2. The thick black lines represent P_1 and P_2 and the colored rectangles are the ones spanned by the respective terminals and marbles. For easier distinction, we colored everything related to the s_1-t_1-path red and related to the s_2-t_2-path blue (except for the respective paths themselves). A black square represents a vector on which we guessed a marble on both paths. The dashed line represents the subpaths of P_1 and P_2 that have a common color.
(top left-hand corner): The lines cross in one point with non-integer coordinates.
(top right-hand corner): The lines cross in at least one point with integer coordinates.
(bottom left-hand corner): The rectangles defined by s_i and t_i intersect (in the gray (darker) area), but the lines do not.
(bottom right-hand corner): The rectangles defined by s_i and t_i do not intersect.

of the respective hyperrectangles are disjoint. For each pair (P_i, P_j) of paths, we start with the orthogonal projection to the coordinates i and j. This yields a set of marbles for P_i and P_j such that the respective subpaths either have colors i and j or cannot interfere with the respective other path. Assume that we guessed for each two-dimensional (i,j)-projection the intersection of P_i and P_j in this projection. Unfortunately, we cannot partition the respective subpaths as stated above. Instead, we store for each subpath P'_i of P_i the set Φ of all colors that P'_i has and recursively refine these subpaths until a partition as stated is possible. Roughly speaking, we check for each pair of subpaths whether the areas of their respective hyperrectangles intersect and if they do, then we find a two-dimensional projection and use this to find new marbles. The resulting subpaths are then either disjoint from the respective other path or have an additional color. We continue this procedure until the areas of any two subpaths with different colors are disjoint. These subpaths are then partitioned by their respective sets of colors. Note that by construction different subpaths in one part of the partition share a common color and the areas of the hyperrectangles spanned by the end vertices of two subpaths in different parts of the partition are disjoint.

In Subsection 5.2.1 we formalize the geometry-based ideas and provide a characterization of solutions for 2-Disjoint Shortest Paths. In Subsection 5.2.2 we generalize this characterization to solutions of k-Disjoint Shortest Paths.

5.2.1 Two Shortest Paths

We now formalize and generalize the idea behind the geometric view (visualized in Figures 5.1 and 5.2). We start with some notation for projections. For any $\emptyset \subset I \subseteq [k]$ and any vector $x \in \mathbb{R}^k$, we denote with $x^I \in \mathbb{R}^{|I|}$ the orthogonal projection of x to the coordinates in I. That is, x^I is the $|I|$-dimensional vector obtained by deleting all dimensions in x that are not in I. We usually drop the brackets in the exponent, thus writing e. g. $(5,6,7,8,9)^{1,3,4} := (5,7,8)$ or $(5,6,7)^2 := (6)$. Similarly, for $R \subseteq \mathbb{R}^k$ we define $R^I := \{x^I \mid x \in R\} \subseteq \mathbb{R}^{|I|}$.

We associate with each vertex $v \in V$ a vector in the k-dimensional Euclidean vector space. Formally, $\vec{v} := (\vec{v}^i)_{i\in[k]} := (\operatorname{dist}(s_i, v))_{i\in[k]} \in \mathbb{N}^k$ and for $U \subseteq V$ we use $\vec{U} := \{\vec{u} \mid u \in U\}$ to denote the set of all vectors of vertices in U. For a given instance of k-Disjoint Shortest Paths, one can compute the vector of each vertex in $O(km)$ time by performing a breadth-first-search from each vertex s_i.

We use the following notations for any non-empty index set $I \subseteq [k]$ in order to compare vectors of vertices v, w or sets V, W of vertices:

$$\begin{aligned} v \simeq^I w &\iff \forall c \in I.\ \vec{v}^c \simeq \vec{w}^c && \text{for } \simeq \in \{<, \leq, =, \geq, >\}, \text{ and} \\ V \simeq^I W &\iff \{\vec{v}^I \mid v \in V\} \simeq \{\vec{w}^I \mid w \in W\} && \text{for } \simeq \in \{\subset, \subseteq, =, \supseteq, \supset\}. \end{aligned}$$

We further write $x \in^I X$ if there is an $x' \in X$ with $x' =^I x$ and $x \notin^I X$ otherwise.

Lemma 5.1. *For any pair of vertices $v, w \in V$, we have $\|\vec{v} - \vec{w}\|_\infty \leq \operatorname{dist}(v, w)$.*

Proof. Let P be a shortest v-w-path. Each edge $\{p, q\}$ in P fulfills $\|\vec{p} - \vec{q}\|_\infty \leq 1$ as $|\operatorname{dist}(s_i, p) - \operatorname{dist}(s_i, q)| \leq 1$ for each vertex s_i. Thus, by the triangle inequality, $\|\vec{v} - \vec{w}\|_\infty \leq \sum_{a \in A(P)} 1 = \operatorname{dist}(v, w)$. □

For two vertices $u, w \in V$, let

$$u \diamond w := \{v \in V \mid \operatorname{dist}(u, v) + \operatorname{dist}(v, w) = \operatorname{dist}(u, w)\}$$

be the set of all vertices that lie on a shortest u-w-path. Similarly, for any $x, y \in \mathbb{N}^k$, let

$$x \diamond y := \{z \in \mathbb{R}^k \mid \|x - z\|_\infty + \|z - y\|_\infty = \|x - y\|_\infty\}$$

be the hyperrectangle spanned by x and y (whose sides form an angle of 45° with the coordinate axes (see Figure 5.1)). We continue with a formal definition of *colors*.

Definition 5.1. Let s, t be two vertices and let P be a shortest s-t-path. The pair (s, t) and the path P are *colored* if $\operatorname{dist}(s, t) = \|\vec{s} - \vec{t}\|_\infty$. Let

$$C(P) := C(s, t) := \{c \in [k] \mid |\vec{s}^{\,c} - \vec{t}^{\,c}| = \|\vec{s} - \vec{t}\|_\infty\}$$

be the set of all colors of P. The pair (s, t) and the path P are *c-colored* for each $c \in C(s, t)$.

Note that this definition of a c-colored path is equivalent to saying that P is strictly monotonous in its c-coordinates. Note further that for arbitrary $u, w \in V$ we do *not* always have $\overrightarrow{u \diamond w} \subseteq \vec{u} \diamond \vec{w}$, that is, the vectors of all vertices on a shortest u-w-path are not necessarily contained in the set of vectors "spanned" by $\vec{u}$ and $\vec{w}$. However, this inclusion holds for colored vertex pairs as shown next.

Lemma 5.2. *Let $v, w \in V$ be a b-colored pair. Then, $\overrightarrow{v \diamond w} \subseteq \vec{v} \diamond \vec{w}$.*

Proof. Without loss of generality $v \leq^b w$. Let u be an arbitrary vertex in $v \diamond w$. Then, $\operatorname{dist}(v, w) = \operatorname{dist}(v, u) + \operatorname{dist}(u, w)$. Definition 5.1 and Lemma 5.1 yield

$$\vec{w}^{\,b} = \vec{v}^{\,b} + \operatorname{dist}(v, w) = \vec{v}^{\,b} + \operatorname{dist}(v, u) + \operatorname{dist}(u, w) \geq \vec{u}^{\,b} + \operatorname{dist}(u, w) \geq \vec{w}^{\,b}.$$

Hence, $\vec{u}^b = \vec{v}^b + \text{dist}(v,u)$ and $\vec{w}^b = \vec{u}^b + \text{dist}(u,w)$. Lemma 5.1 then states that $\text{dist}(v,u) = \|\vec{v} - \vec{u}\|_\infty$ and $\text{dist}(u,w) = \|\vec{v} - \vec{u}\|_\infty$. Hence,

$$\|\vec{v} - \vec{w}\|_\infty = \text{dist}(v,w) = \text{dist}(v,u) + \text{dist}(u,w) = \|\vec{v} - \vec{u}\|_\infty + \|\vec{u} - \vec{w}\|_\infty.$$

This leads to $\vec{u} \in \vec{v} \diamond \vec{w}$ and thus $\overrightarrow{v \diamond w} \subseteq \vec{v} \diamond \vec{w}$. □

Throughout this chapter, we will be particularly interested in two-dimensional projections of areas $\vec{v} \diamond \vec{w}$ for some vertices v and w. Note in this context that $(\vec{v} \diamond \vec{w})^I = \vec{v}^I \diamond \vec{w}^I$. Recall that the area defined by $x \diamond y$ for $x, y \in \mathbb{N}^2$ is a rectangle in the plane whose sides form an angle of 45° to the coordinate axes. The following lemma lists necessary and sufficient conditions for those rectangles to intersect.

Lemma 5.3. *Let $x, y, \hat{x}, \hat{y} \in \mathbb{N}^2$. Then $x \diamond y \cap \hat{x} \diamond \hat{y} \neq \emptyset$ if and only if all of the following hold:*

(i) $\min\{x^1 - x^2, y^1 - y^2\} \leq \max\{\hat{x}^1 - \hat{x}^2, \hat{y}^1 - \hat{y}^2\}$,

(ii) $\min\{\hat{x}^1 - \hat{x}^2, \hat{y}^1 - \hat{y}^2\} \leq \max\{x^1 - x^2, y^1 - y^2\}$,

(iii) $\min\{x^1 + x^2, y^1 + y^2\} \leq \max\{\hat{x}^1 + \hat{x}^2, \hat{y}^1 + \hat{y}^2\}$, *and*

(iv) $\min\{\hat{x}^1 + \hat{x}^2, \hat{y}^1 + \hat{y}^2\} \leq \max\{x^1 + x^2, y^1 + y^2\}$.

Proof. Let $R_1, R_2 \subseteq \mathcal{R}^2$ be two axis-parallel rectangles defined by the opposite corners $q, r \in \mathcal{R}^2$ and $\hat{q}, \hat{r} \in \mathcal{R}^2$. It is easy to see that R_1 and R_2 intersect if and only if

a) $\min\{q^1, r^1\} \leq \max\{\hat{q}^1, \hat{r}^1\}$ and $\min\{\hat{q}^1, \hat{r}^1\} \leq \max\{q^1, r^1\}$
(i. e., there is an overlap in the first coordinate), and

b) $\min\{q^2, r^2\} \leq \max\{\hat{q}^2, \hat{r}^2\}$ and $\min\{\hat{q}^2, \hat{r}^2\} \leq \max\{q^2, r^2\}$
(i. e., there is an overlap in the second coordinate).

Since the intersection of two rectangles is invariant under rotation and scaling, we simply rotate $x \diamond y$ and $\hat{x} \diamond \hat{y}$ by 45° (and scale it by factor $\sqrt{2}$) by multiplying all vectors with the matrix

$$R = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}.$$

Now the above characterization for axis-parallel rectangles translates into the conditions stated in the lemma. □

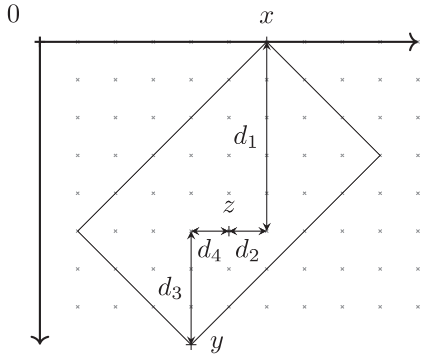


Figure 5.3: The rectangle $x \diamond y$ spanned by two points x and z in two dimensions and a point $z \in x \diamond y$. Note that $\|y - x\|_\infty$ is the vertical distance between x and y. Lemma 5.4 states that $d_1 \geq d_2$ and $d_3 \geq d_4$.

The next lemma states that the distance between x and any $z \in x \diamond y$ where x and y have distance $|y^c - x^c|$ is at most $|z^c - x^c|$. Intuitively, this is clear as $x \diamond y$ is a hyperrectangle whose sides form an angle of 45° with the coordinate axes and hence half of its borders exactly define all points z whose distance to x is exactly $|z^c - x^c|$. See Figure 5.3 for an illustration.

Lemma 5.4. *Let $b, c \in [k]$ and let $x, y \in \mathbb{N}^k$ with $\|y - x\|_\infty = y^c - x^c$. Then, for all $z \in x \diamond y$ it holds that $z^c - x^c \geq |z^b - x^b| \geq 0$ and $y^c - z^c \geq |y^b - z^b| \geq 0$.*

Proof. By assumption and the definition of $x \diamond y$, it holds that

$$\begin{aligned} y^c - x^c &= \|y - x\|_\infty = \|y - z\|_\infty + \|z - x\|_\infty \\ &\geq |y^c - z^c| + |z^c - x^c| \geq (y^c - z^c) + (z^c - x^c) \\ &= y^c - x^c. \end{aligned}$$

Thus, we have equality everywhere, in particular $y^c - z^c = \|y - z\|_\infty \geq |z^b - x^b|$ and $z^c - x^c = \|z - x\|_\infty \geq |y^b - z^b|$ (as shown by the equality between the last term in the first row and the first term in the second row). □

We next formalize the lines we used in Figure 5.1 to connect the vectors of vertices in a path. To this end, for any path $P = (v_1, v_2, \ldots, v_i)$ we define $\zeta(P) \subset \mathbb{R}^k$ as the piecewise linear curve connecting the points of $\overrightarrow{P}$ in the order given by P. Recall that $C(P)$ denotes the set of all colors a such that P is a-colored. The next observation states that $\zeta(P)^{C(P)}$ is a straight line, which is equivalent to the statement that P is strictly monotone in each coordinate in $C(P)$.

Observation 5.5. *Let P be a colored path. Then $\zeta(P)^{C(P)}$ is a straight line segment.*

Proof. Let $\ell := \|\overrightarrow{s_P} - \overrightarrow{t_P}\|_\infty$ and $k' := |C(P)|$. The path P contains exactly ℓ edges, each of which has an Euclidean length of at most $\sqrt{k'}$ in the projection $\zeta(P)^{C(P)}$. Thus the length of $\zeta(P)^{C(P)}$ is at most $\ell \cdot \sqrt{k'}$ which is exactly the Euclidean distance between $\overrightarrow{s_P}^{C(P)}$ and $\overrightarrow{t_P}^{C(P)}$. $\square$

As a consequence of Observation 5.5, the intersection of two paths P, Q in the $(C(P) \cup C(Q))$-projection is also a straight line segment with an angle of 45° to the coordinate axes as shown in Figure 5.1 (right side) and Figure 5.2 (top right).

Lemma 5.6. *Let P and Q be two colored paths, and $C \subseteq C(P) \cup C(Q)$. Then $\zeta(P)^C \cap \zeta(Q)^C$ is a (possibly empty) straight line segment.*

Proof. For the sake of notation, we assume that $C = [|C|]$. Note that $\zeta(P_a)$ and $\zeta(P_b)$ are piecewise linear curves. Moreover, according to Lemma 5.2 for any two points $x, y \in \zeta(P)$, it holds that $\|x - y\|_\infty = |x^c - y^c|$ for all $c \in C(P)$ and $\|x' - y'\|_\infty = |x'^b - y'^b|$ for any two points $x', y' \in \zeta(Q)$ and all $b \in C(Q)$. So, for $x, y \in \mathbb{R}^k$ with $x^C, y^C \in \zeta(P)^C \cap \zeta(Q)^C$, it follows that

$$|x^c - y^c| = \|x - y\|_\infty = |x^b - y^b|$$

for all $c \in C \cap C(P)$ and all $b \in C \cap C(Q)$. Thus, $C(x, y) \supseteq C$ and the claim follows from Observation 5.5. $\square$

Note that even if $\zeta(P)^{C(P)\cup C(Q)} \cap \zeta(Q)^{C(P)\cup C(Q)}$ is non-empty, then it does not need to contain points from $\mathbb{N}^{|C(P)\cup C(Q)|}$ as can be seen in the top left example in Figure 5.2.

The following definition starts to formalize the notion of marbles, that is, the special vertices in the different cases in Figure 5.2. We start with the two cases in which the lines of P and Q cross (the upper two).

Definition 5.2. Let P, Q be two colored paths, let $b \in C(P)$, and let $c \in C(Q)$. The paths P and Q are *b, c-crossing* if the intersection

$$X := \zeta(P)^{b,c} \cap \zeta(Q)^{b,c}$$

is non-empty and they are *b, c-non-crossing* otherwise.

If $\overrightarrow{P}^{b,c} \cap X \neq \emptyset$, then we define $\alpha_P^{b,c}(Q)$ and $\omega_P^{b,c}(Q)$ to be the first and last vertex v of P with $v^{b,c} \in X$. If $\overrightarrow{P}^{b,c} \cap X$ only contains non-integer coordinates, then $\alpha_P^{b,c}(Q) := \omega_P^{b,c}(Q) := \perp$. We further define $\partial_P^{b,c}(Q)$ and $\varpi_P^{b,c}(Q)$ to be the last vertex before and the first vertex after that intersection. If $\overrightarrow{P}^{b,c} \cap X = \emptyset$, then $\alpha_P^{b,c}(Q) := \omega_P^{b,c}(Q) := \partial_P^{b,c}(Q) := \varpi_P^{b,c}(Q) := \perp$.

Regarding notation, we will use α_P instead of $\alpha_P^{b,c}(Q)$ (and the same for ω, ∂, and ϖ) if b, c, and Q are clear from the context. Note that the subpaths between the respective α- and ω-vertices are by Lemma 5.6 straight lines and $\{b, c\}$-colored.

Observation 5.7. *If P, Q are two paths with $\alpha_P^{b,c}(Q) \neq \perp$, then*

$$P[\alpha_P^{b,c}(Q), \omega_P^{b,c}(Q)] =^{b,c} Q[\alpha_Q^{b,c}(P), \omega_Q^{b,c}(P)].$$

In particular, both of these subpaths are b, c-colored.

It remains to consider the subpaths between s- and α-vertices and between ω- and t-vertices. By Lemma 5.2 these have to lie in the rectangle areas

$$\overrightarrow{s_P} \diamond \overrightarrow{\partial_P^{b,c}(Q)}, \overrightarrow{\varpi_P^{b,c}(Q)} \diamond \overrightarrow{t_P}, \overrightarrow{s_Q} \diamond \overrightarrow{\partial_Q^{b,c}(P)}, \text{ and } \overrightarrow{\varpi_Q^{b,c}(P)} \diamond \overrightarrow{t_Q}. \tag{5.1}$$

Figure 5.2 (top left-hand corner and top right-right hand corner) suggests that these areas are pairwise disjoint. We will show that this is indeed the case and, to this end, we show the following two observations. The first one states that the ∂-vertex on P has a b-coordinate that is at most the b-coordinate of the ∂-vertex on Q, where b is the "original" color of P. Note that this ∂-vertex is right before the respective α-vertex or before the single crossing point with non-integer coordinates. Since P is strictly b-monotone, the path Q can at most increase or decrease as fast as P from the point of intersection.

Observation 5.8. *Let P, Q be two b, c-crossing paths with $\partial_P^{b,c}(Q) \neq \perp$. If P is a subpath of a shortest s_b-t_b-path and Q is a subpath of a shortest s_c-t_c-path, then $\overrightarrow{\partial_P^{b,c}(Q)}^{b} \leq \overrightarrow{\partial_Q^{b,c}(P)}^{b}$ and $\overrightarrow{\partial_Q^{b,c}(P)}^{c} \leq \overrightarrow{\partial_P^{b,c}(Q)}^{c}$.*

Proof. Let $z \in \zeta(P)^{b,c} \cap \zeta(Q)^{b,c}$ have minimal b-coordinate. Note that

$$\left\| z - \overrightarrow{\partial_P(Q)}^{b,c} \right\|_\infty = \left\| z - \overrightarrow{\partial_Q(P)}^{b,c} \right\|_\infty ,$$

and since $\zeta(P)$ is strictly increasing in its b-coordinate, we can infer that

$$z^b - \overrightarrow{\partial_P(Q)}^b = \left\| z - \overrightarrow{\partial_P(Q)}^{b,c} \right\|_\infty = \left\| z - \overrightarrow{\partial_Q(P)}^{b,c} \right\|_\infty \geq z^b - \overrightarrow{\partial_Q(P)}^b .$$

This yields $\overrightarrow{\partial_P(Q)}^b \leq \overrightarrow{\partial_Q(P)}^b$ and the second inequality follows analogously. $\square$

The second observation is a simple but useful restatement of Lemma 5.1.

Observation 5.9. *Let $b, c \in [k]$ and let (v, w) be a b-colored pair of vertices with $v <^b w$. Then $\vec{w}^b - \vec{w}^c \geq \vec{v}^b - \vec{v}^c$.*

Proof. By Lemma 5.1, $\vec{w}^c - \vec{v}^c \leq \operatorname{dist}(v, w) = \vec{w}^b - \vec{v}^b$. A simple arithmetic reformulation yields $\vec{w}^c - \vec{w}^b \leq \vec{v}^c - \vec{v}^b$ and multiplying both sides with -1 completes the proof. $\square$

We are now in the position to prove the statement that the four areas defined in Term (5.1) are pairwise disjoint.

Lemma 5.10. *Let P and Q be two b, c-crossing paths. The sets*

$$\left(\overrightarrow{s_P} \diamond \overrightarrow{\partial_P^{b,c}(Q)}\right)^{b,c}, \left(\overrightarrow{\varpi_P^{b,c}(Q)} \diamond \overrightarrow{t_P}\right)^{b,c}, \left(\overrightarrow{s_Q} \diamond \overrightarrow{\partial_Q^{b,c}(P)}\right)^{b,c}, \text{ and } \left(\overrightarrow{\varpi_Q^{b,c}(P)} \diamond \overrightarrow{t_Q}\right)^{b,c}$$

are pairwise disjoint (or undefined).

Proof. Without loss of generality, let P be b-colored, Q be c-colored, $s_P <^b t_P$, and $s_Q <^c t_Q$. Recall that s_P and t_P are the start and end vertices of P, respectively. We further assume that all above sets are defined, that is, none of the described end points is $\perp$. By Lemma 5.4, for any $x \in \overrightarrow{s_P} \diamond \overrightarrow{\partial_P}$ and $y \in \overrightarrow{\varpi_P} \diamond \overrightarrow{t_P}$ it holds that $x \leq \overrightarrow{\partial_P}^b < \overrightarrow{\varpi_P}^b \leq y^b$, and thus $\left(\overrightarrow{s_P} \diamond \overrightarrow{\partial_P}\right)^{b,c} \cap \left(\overrightarrow{\varpi_P} \diamond \overrightarrow{t_P}\right)^{b,c} = \emptyset$.

An analogous argument holds for $\left(\overrightarrow{s_P} \diamond \overrightarrow{\partial_P}\right)^{b,c}$ and $\left(\overrightarrow{\varpi_P} \diamond \overrightarrow{t_P}\right)^{b,c}$.

We will now use Lemma 5.3 to show that $\left(\overrightarrow{s_P} \diamond \overrightarrow{\partial_P}\right)^{b,c} \cap \left(\overrightarrow{s_Q} \diamond \overrightarrow{\partial_Q}\right)^{b,c} = \emptyset$. Since all other remaining cases are analogous, this will conclude the proof. By Observation 5.9, it holds that

$$\max\{\overrightarrow{s_P}^b - \overrightarrow{s_P}^c, \overrightarrow{\partial_P}^b - \overrightarrow{\partial_P}^c\} = \overrightarrow{\partial_P}^b - \overrightarrow{\partial_P}^c \text{ and}$$
$$\min\{\overrightarrow{s_Q}^b - \overrightarrow{s_Q}^c, \overrightarrow{\partial_Q}^b - \overrightarrow{\partial_Q}^c\} = \overrightarrow{\partial_Q}^b - \overrightarrow{\partial_Q}^c.$$

Observe that $\overrightarrow{\partial_P}^{b,c} \neq \overrightarrow{\partial_Q}^{b,c}$ as otherwise ∂_P would lie on the intersection of P and Q, a contradiction. Hence $\overrightarrow{\partial_P}^b \neq \overrightarrow{\partial_Q}^b$ or $\overrightarrow{\partial_P}^c \neq \overrightarrow{\partial_Q}^c$. In the former case, Observation 5.8 states that $\partial_Q >^b \partial_P$ and in the latter case it states $\partial_P >^c \partial_Q$. Hence, $\overrightarrow{\partial_P}^c + \overrightarrow{\partial_Q}^b > \overrightarrow{\partial_P}^b + \overrightarrow{\partial_Q}^c$ and thus $\overrightarrow{\partial_P}^b - \overrightarrow{\partial_P}^c < \overrightarrow{\partial_Q}^b - \overrightarrow{\partial_Q}^c$.

Setting $x = \overrightarrow{s_P}^{b,c}$, $y = \overrightarrow{\partial_P}^{b,c}$, $\hat{x} = \overrightarrow{s_Q}^{b,c}$, and $\hat{y} = \overrightarrow{\partial_Q}^{b,c}$ in Lemma 5.3 then violates condition (ii) and hence $\left(\overrightarrow{s_P} \diamond \overrightarrow{\partial_P}\right)^{b,c} \cap \left(\overrightarrow{s_Q} \diamond \overrightarrow{\partial_Q}\right)^{b,c} = \emptyset$. $\square$

We continue with the definition of marbles for the cases where the two paths P and Q are b, c-non-crossing. Figure 5.2 shows that even in this case $s_P \diamond t_P$ and $s_Q \diamond t_Q$ in general are not disjoint (bottom left-hand corner). Since the two bottom cases are distinguished by the intersection of $s_P \diamond t_P$ and $s_Q \diamond t_Q$, we start with a definition of this intersection.

Definition 5.3. Let $b, c \in [k]$. Let P be a b-colored path and let Q be a c-colored path. The *common* b, c*-area* of P and Q is

$$\Delta^{b,c}(P, Q) := (\overrightarrow{s_P} \diamond \overrightarrow{t_P})^{b,c} \cap (\overrightarrow{s_Q} \diamond \overrightarrow{t_Q})^{b,c}.$$

Note that if $\Delta^{b,c}(P, Q) = \emptyset$, then by Lemma 5.2, they do not share vertices with common vectors and hence they do not share common vertices. If $\Delta^{b,c}(P, Q) \neq \emptyset$ and P and Q are b, c-crossing, then we can use Definition 5.2 to define the marbles. It hence remains to study the case where $\Delta^{b,c}(P, Q) \neq \emptyset$ and P and Q are b, c-non-crossing. In this case we need at most one marble per path and this marble is defined as follows.

Definition 5.4. Let P be a b-colored path, Q be a c-colored path, and let without loss of generality be $s_Q <^b t_Q$. Define

$$B := \{v \in V \mid v =^b s_Q \wedge v <^c s_Q\} \cup \{v \in V \mid v =^b t_Q \wedge v >^c t_Q\}.$$

If $P \cap B \neq \emptyset$, then $\delta_P^{b,c}(Q)$ is the unique vertex in $P \cap B$. If $P \cap B = \emptyset$, then $\delta_P^{b,c}(Q) = \bot$.

Observation 5.11. *$\delta_P^{b,c}(Q)$ is well-defined.*

Proof. Since P is strictly monotone in the b-coordinate, it can clearly contain at most one point from $B_1 := \{v \in V \mid v =^b s_Q \wedge v <^c s_Q\}$ and one from $B_2 := \{v \in V \mid v =^b t_Q \wedge v >^c t_Q\}$. It remains to show that it cannot intersect both sets. To this end, observe that Lemma 5.4 states for any $b_1 \in B_1$ and $b_2 \in B_2$ that $|b_1^c - b_2^c| > |s_Q^b - t_Q^b| \geq |s_Q^b - t_Q^b| = |b_1^b - b_2^b|$, and therefore the pair (b_1, b_2) is not b-colored and thus P cannot contain both b_1 and b_2. $\square$

We next show that if $\Delta^{b,c}(P,Q) \neq \emptyset$ and P and Q are b,c-non-crossing, then at least one of the vertices $\delta_P^{b,c}(Q)$ or $\delta_Q^{b,c}(P)$ exists. Afterwards we will show that these vertices guarantee that the respective new areas are disjoint.

Lemma 5.12. *Let P, Q be two b,c-non-crossing paths with $\Delta^{b,c}(P,Q) \neq \emptyset$. Then, $\delta_P^{b,c}(Q) \neq \bot$ or $\delta_Q^{b,c}(P) \neq \bot$.*

Proof. Note that the definition of $\Delta^{b,c}(P,Q)$ requires that without loss of generality P is b-colored and Q is c-colored. We further assume without loss of generality that $\{b,c\} = [2]$ and that $s_P <^b t_P$ and $s_Q <^c t_Q$.

Suppose towards a contradiction that $\delta_P^{b,c}(Q) = \delta_Q^{b,c}(P) = \bot$. Since P, Q are b,c-non-crossing and $\delta_Q^{b,c}(P) = \bot$, the path Q cannot cross the curve

$$\{x \in \mathbb{N}^2 \mid x^c = s_P \wedge x^b < s_P\} \cup \zeta(P)^{b,c} \cup \{x \in \mathbb{N}^2 \mid x^c = t_P \wedge x^b > t_P\}.$$

Since Q cannot cross the line, it is located completely on one side of it. Assume without loss of generality that Q (and thus in particular t_Q) is located on the side containing $(0,0)$, that is, for all v in P and w in Q with $v^b = w^b$ it holds that $v^c > w^c$.

Then there are three possible cases: $t_Q^b < s_P^b$, $t_Q^b \in [s_P^b, t_P^b]$, or $t_Q^b > t_P^b$. Note that in the first case by Lemma 5.4 it holds for any $z \in \Delta^{b,c}(P,Q)$ that

$$z^c - t_Q^c \geq z^b - t_Q^b > z^b - s_P^b \geq z^c - s_P^c,$$

a contradiction to $z \in (\overrightarrow{s_Q} \diamond \overrightarrow{t_Q})^{b,c}$. In the last case, a similar argument holds with

$$s_P^b < t_Q^b - z^b \leq t_Q^c - z^c < s_P^c - z^c,$$

which is a contradiction to $z \in (\overrightarrow{s_P} \diamond \overrightarrow{t_P})^{b,c}$. It remains to analyze the case where $t_Q^b \in [s_P^b, t_P^b]$. Note that in this case there is a vertex p in P with $p^b = t_Q^b$. Hence $t_Q^c < p^c$, and $p \in \{v \in V \mid v =^b t_Q \wedge v >^c t_Q\}$. Thus $\delta_P^{b,c}(Q) = p \neq \bot$, a contradiction. □

The next lemma shows that if P and Q are b,c-non-crossing, but they have a common area $\Delta_{b,c}$ and $\delta_P^{b,c}(Q) \neq \bot$, then the area between s_Q and t_Q is disjoint from the two areas between s_P and $\delta_P^{b,c}(Q)$ and between $\delta_P^{b,c}(Q)$ and t_P. Hence $\delta_P^{b,c}(Q)$ is the last type of marble needed.

Lemma 5.13. *Let P be a b-colored path and let Q a c-colored path such that $\Delta_{b,c}(P,Q) \neq \bot$ and $\delta_P^{b,c}(Q) \neq \bot$. Then,*

$$\left(\overrightarrow{s_Q} \diamond \overrightarrow{t_Q}\right)^{b,c} \text{ is disjoint from } \left(\overrightarrow{s_P} \diamond \overrightarrow{\delta_P^{b,c}(Q)}\right)^{b,c} \cup \left(\overrightarrow{\delta_P^{b,c}(Q)} \diamond \overrightarrow{t_P}\right)^{b,c}.$$

Proof. For the sake of readability, we use $\delta := \delta_P^{b,c}(Q)$. We will show that

$$\left(\overrightarrow{s_Q} \diamond \overrightarrow{t_Q}\right)^{b,c} \cap \left(\overrightarrow{s_P} \diamond \overrightarrow{\delta}\right)^{b,c} = \emptyset.$$

The proof for $(\overrightarrow{\delta} \diamond \overrightarrow{t_P})^{b,c}$ is then completely analogous. Assume without loss of generality that $\delta =^b t_Q$ and $\delta >^c t_Q$. Notice that

$$\begin{aligned}\max\{\overrightarrow{s_P}^b - \overrightarrow{s_P}^c, \overrightarrow{\delta}^b - \overrightarrow{\delta}^c\} &\overset{\text{Obs. 5.9}}{=} \overrightarrow{\delta}^b - \overrightarrow{\delta}^c \\ &< \overrightarrow{t_Q}^b - \overrightarrow{t_Q}^c \overset{\text{Obs. 5.9}}{=} \min\{\overrightarrow{s_Q}^b - \overrightarrow{s_Q}^c, \overrightarrow{t_Q}^b - \overrightarrow{t_Q}^c\}.\end{aligned}$$

Setting $x := \overrightarrow{s_P}^{b,c}$, $y := \overrightarrow{\delta}^{b,c}$, $\hat{x} := \overrightarrow{s_Q}^{b,c}$, and $\hat{y} := \overrightarrow{t_Q}^{b,c}$ in Lemma 5.3 then yields that condition (ii) is violated and thus $(\overrightarrow{s_P} \diamond \overrightarrow{\delta})^{b,c} \cap (\overrightarrow{s_Q} \diamond \overrightarrow{t_Q})^{b,c} = \emptyset$. □

We are finally in the position to define the set of marbles for a pair of paths. Afterwards we conclude this subsection with the main proposition that states that marbles uniquely classify solutions.

Definition 5.5. Let P be a b-colored s_P-t_P-path and Q be a c-colored s_Q-t_Q-path. The set of $\{b,c\}$-*marbles* of P with respect to Q is

$$\mathcal{M}_P^{b,c}(Q) := \{s_P, t_P, \mu_P^{b,c}(Q) \mid \mu \in \{\alpha, \omega, \partial, \varpi, \delta\}\} \setminus \{\bot\}.$$

The next proposition states that if there are two different solutions and in particular two pairs (P, Q) and (P', Q') of solution paths with the same marbles, then P and Q share exactly the same vectors as P' and Q' do. Recall that the shared vectors are a straight line segment and that once their ends are fixed, we can use the dynamic program by Fortune et al. [FHW80] to find disjoint paths between these ends.

Proposition 5.14. *Let P and P' be b-colored s_P-t_P-paths, and let Q and Q' be c-colored s_Q-t_Q-paths. If $\mathcal{M}_P^{b,c}(Q) \subseteq P'$ and $\mathcal{M}_Q^{b,c}(P) \subseteq Q'$, then*

$$\{v \in P' \mid v \in^{b,c} Q'\} =^{b,c} \{v \in P \mid v \in^{b,c} Q\}.$$

Proof. Let R be the subpath of P that starts at $\alpha_P^{b,c}(Q)$ and ends at $\omega_P^{b,c}(Q)$ (or $R = \emptyset$ if $\alpha_P = \omega_P = \perp$). From the definition of α and ω and Lemma 5.6, it follows that $\{v \in P \mid v \in^{b,c} Q\} =^{b,c} R$. We now consider the two cases whether or not P and Q are b, c-crossing.

If P and Q are b, c-crossing, then by definition $\partial_P \neq \perp$ and $\varpi_P \neq \perp$. It follows from Lemma 5.2 that the subpaths of P' from s_P to ∂_P and from ϖ_P to t_P use only vectors from $\overrightarrow{s_P} \diamond \overrightarrow{\partial_P}$ and $\overrightarrow{\varpi_P} \diamond \overrightarrow{t_P}$, respectively. As the analogous statement holds for the corresponding subpaths of Q', it follows from Lemma 5.10 that all these subpaths do not intersect in the projection to the b-c-plane. It remains to consider the subpath from α_P to ω_P. If $\alpha_P = \omega_P = \perp$, then

$$\{v \in P' \mid v \in^{b,c} Q'\} = \emptyset = R = \{v \in P \mid v \in^{b,c} Q\}.$$

Otherwise, $\overrightarrow{R}^{b,c}$ is by Lemma 5.6 a straight diagonal line and by Observation 5.5 so is $\{v \in P' \mid v \in^{b,c} Q'\}$. Since those two straight line segments have the same ends, they are the same and thus $\{v \in P' \mid v \in^{b,c} Q'\} =^{b,c} R$.

If P and Q are b, c-non-crossing, then $\{v \in P' \mid v \in^{b,c} Q'\}^{b,c} \subseteq \Delta^{b,c}(P, Q)$ and $\emptyset = R$. We consider the two cases $\Delta^{b,c}(P, Q) = \emptyset$ and $\Delta^{b,c}(P, Q) \neq \emptyset$. In the former case it holds that

$$\{v \in P \mid v \in^{b,c} Q\} = R = \emptyset = \Delta^{b,c}(P, Q) = \Delta^{b,c}(P', Q') = \{v \in P' \mid v \in^{b,c} Q'\}.$$

In the latter case, by Lemma 5.12 there is a $\delta_P^{b,c}(Q) \neq \bot$ or $\delta_Q^{b,c}(P) \neq \bot$. Without loss of generality, assume that $\delta_P^{b,c}(Q) \neq \bot$. Then, $\delta_P^{b,c}(Q) \in \mathcal{M}_P^{b,c}(Q) \subseteq P'$ and by Lemma 5.13

$$\begin{aligned}
\overrightarrow{\{v \in P' \mid v \in^{b,c} Q'\}}^{b,c} &\subseteq \left(\overrightarrow{s_{Q'}} \diamond \overrightarrow{t_{Q'}}\right)^{b,c} \cap \left((\overrightarrow{s_{P'}} \diamond \overrightarrow{\delta_P^{b,c}(Q)})^{b,c} \cup (\overrightarrow{\delta_P^{b,c}(Q)} \diamond \overrightarrow{t_{P'}})^{b,c} \right) \\
&= \left(\overrightarrow{s_Q} \diamond \overrightarrow{t_Q}\right)^{b,c} \cap \left((\overrightarrow{s_P} \diamond \overrightarrow{\delta_P^{b,c}(Q)})^{b,c} \cup (\overrightarrow{\delta_P^{b,c}(Q)} \diamond \overrightarrow{t_P})^{b,c} \right) \\
&= \emptyset.
\end{aligned}$$

Thus, $\{v \in P' \mid v \in^{b,c} Q'\} = \emptyset = R = \{v \in P \mid v \in^{b,c} Q\}$. $\square$

5.2.2 More than Two Shortest Paths

In the previous subsection, we looked at two shortest paths P and Q from s_P to t_P and s_Q and t_Q, respectively. We showed that selecting at most ten vertices from P and Q (five per path; see Definition 5.5) is sufficient to ensure that each pair (P', Q') of shortest s_P-t_P- and s_Q-t_Q-paths that also contain these vertices ($\mathcal{M}_P^{b,c}(Q)$ and $\mathcal{M}_Q^{b,c}(P)$) "behave" like P and Q in the sense that P' and Q' intersect in the same vectors as P and Q do (see Proposition 5.14). In this subsection, we define a set $\mathcal{C}$, $|\mathcal{C}| \in O(k \cdot k!)$, that basically ensures the same properties for k paths. To formalize our goal for this subsection, we first introduce the concept of *avoiding* paths which is a generalization of a slightly modified version of b,c-non-crossing paths. The modification is to ignore the ends of P and Q to ensure that we can split paths at certain vertices and still can ensure that these different parts are avoiding.

Definition 5.6 (I-avoiding)**.** Let $\emptyset \subset I \subseteq [k]$. Two paths P and Q are *I-avoiding* if $p \notin^I Q$ for each inner vertex p of P and $q \notin^I P$ for each inner vertex q of Q. Two vertex pairs (s_p, t_p) and (s_q, t_q) are I-avoiding if

$$(\overrightarrow{s_p}^I \diamond \overrightarrow{t_p}^I) \cap (\overrightarrow{s_q}^I \diamond \overrightarrow{t_q}^I) \subseteq \{\overrightarrow{s_p}^I, \overrightarrow{t_p}^I\} \cap \{\overrightarrow{s_q}^I, \overrightarrow{t_q}^I\}.$$

Note that being I-avoiding implies being I'-avoiding for all $I' \supseteq I$. We use *avoiding* as a shorthand for $[k]$-avoiding. Two paths P_1 and P_2 are *internally vertex-disjoint* if neither of them contains an inner vertex of the other path. Avoiding paths are clearly internally vertex-disjoint.

Observation 5.15. *Let P, Q be two avoiding paths. Then P is internally vertex-disjoint from Q.*

Moreover, for each pair of avoiding vertex pairs (s, t) and (u, w), the shortest s-t- and u-v-paths are internally vertex-disjoint.

Lemma 5.16. *Let (s, t) and (u, w) be two colored pairs of vertices. If the pairs (s, t) and (u, w) are avoiding, then each shortest s-t-path is internally disjoint from each shortest u-w-path.*

Proof. If (s, t) and (u, w) are avoiding, then by definition and Lemma 5.2

$$\overrightarrow{s_p \diamond t_p}^I \cap \overrightarrow{s_q \diamond t_q}^I \subseteq (\vec{s_p}^I \diamond \vec{t_p}^I) \cap (\vec{s_q}^I \diamond \vec{t_q}^I) \subseteq \{\vec{s_p}^I, \vec{t_p}^I\} \cap \{\vec{s_q}^I, \vec{t_q}^I\},$$

and thus each shortest s-t-path and each shortest u-w-path only intersect in $\{s, t\} \cap \{u, w\}$ and are therefore internally vertex-disjoint. □

With the notation of avoiding pairs, we can formulate our goal for this subsection. To this end, fix a solution $\mathcal{P} = (P_i)_{i \in [k]}$ for a given instance $(G, (s_i, t_i)_{i \in [k]})$ of k-Disjoint Shortest Paths, that is, P_i is the s_i-t_i-path in the solution. Essentially, we want to partition the paths in $\mathcal{P}$ into subpaths and assign a set Φ of *labels* to each subpath ($\Phi \subseteq [k]$) such that the following two conditions are satisfied.

(1.) Let P be a subpath with labels $\Phi \subseteq [k]$. For each $b \in \Phi$, P is b-colored.

(2.) Let P and Q be subpaths from $P_i, P_j \in \mathcal{P}$ with labels $\Phi_P, \Phi_Q \subseteq [k]$, respectively. If $\Phi_P \neq \Phi_Q$, then (s_P, t_P) and (s_Q, t_Q) are avoiding.

Note that (2.) will be the central argument in our algorithm for k-Disjoint Shortest Paths. The algorithm guesses the endpoints of these subpaths and based on (2.) the algorithm can then compute the inner vertices of subpaths with different label sets independently.

Note that for $k = 2$ the partition of P_1 and P_2 along the sets $\mathcal{M}_P^{b,c}(Q)$ and $\mathcal{M}_Q^{b,c}(P)$ satisfies the above. Each subpath of $P_i, i \in [2]$, has label i. Moreover, the subpaths between the α- and ω-vertices have both labels 1 and 2. Hence, (1.) above is satisfied. Furthermore, (2.) follows from Proposition 5.14.

We now generalize this to arbitrary constant k. The basic idea behind defining a respective set $\mathcal{C}$ of marbles is depicted in Figure 5.4. Initially, each path P_i has label i. Whenever two paths P_i and P_j in the solution intersect in the (i, j)-projection (that is, the respective α- and ω-vertices are not $\perp$), then the subpaths P_i' and P_j' in the intersection get both labels i and j. If a third path P' also intersects with P_j', then we try to use the intersections to move

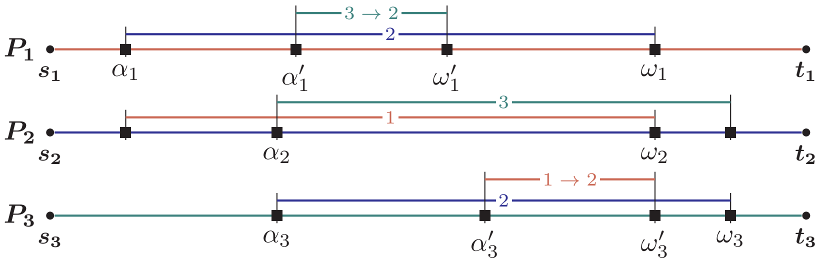


Figure 5.4: The three main lines represent three paths P_1, P_2, and P_3. The small black rectangles represent marbles on the respective path P_i and the j-colored lines above a path indicate that P_i and P_j intersect in the (i, j)-projection, that is, they contain vertices with vectors that are identical when projected in the (i, j)-plane. The paths P_1 and P_2 intersect in the $(1, 2)$-projection, P_2 and P_3 intersect in the $(2, 3)$-projection, but P_1 and P_3 do not intersect in the $(1, 3)$-projection. The subpaths of P_1 and P_3 where they intersect with P_2 in the $(1, 2, 3)$-projection are depicted by $\alpha'_1, \omega'_1, \alpha'_3$, and ω'_3. The colors above each subpath (and also the first number therein) represent the labels of the respective subpath and the number (or sequence of numbers) display the sequence that led to the respective marbles (end vertices) of this subpath.

the label i via path P_j to some subpath of P'. Generalizing this, we consider for each $\sigma = (\ell_1, \ell_2, \ldots, \ell_h)$ whether label ℓ_1 could be "transported" from P_{ℓ_1} to P_{ℓ_2}, from P_{ℓ_2} to P_{ℓ_3}, and so on until from $P_{\ell_{h-1}}$ to $P_{\ell_{h-1}}$. While the idea of transporting labels would also work with triples (transport label a via path P_b to path P_c), we do not have any bound on the number of resulting subpaths (as for each triple there might be many such subpaths). The reason for using sequences is that we will show that for each $\sigma = (\ell_1, \ell_2, \ldots, \ell_h)$ at most one subpath of P_{ℓ_h} can receive label ℓ_1 via σ.

In the following, we use $\text{set}(\tau) := \{\ell_1, \ldots, \ell_h\}$ to denote the set with all entries in a sequence $\tau = (\ell_1, \ldots, \ell_h)$. We next define the *crossing set* $\mathcal{C}$ recursively for each $\Phi \subseteq [k]$. This should be seen as the set of marbles of a solution. We will then show a result similar to Proposition 5.14 for arbitrary k that then allows us to find the desired partition of paths.

Definition 5.7. Let $(G, (s_i, t_i)_{i \in [k]})$ be an instance of k-Disjoint Shortest Paths and let $\mathcal{P} = (P_i)_{i \in [k]}$ be a solution to this instance, that is, P_i is the path between s_i and t_i in the solution. For each $\Phi \subseteq [k]$ and each permutation $\sigma = (\ell_1, \dots, \ell_{|\Phi|})$ of Φ, we define the crossing set $\mathcal{C}^\sigma$ and the endpoints $\mathcal{T}(\sigma)$ of intersections as follows.

- If $|\Phi| = 1$ with $\sigma = (i)$, then let $\mathcal{C}^\sigma := \mathcal{T}(\sigma) := \{s_i, t_i\}$.
- If $|\Phi| = 2$ with $\sigma = (i, j)$, then let
$$\mathcal{T}(\sigma) := \{\alpha^{i,j}_{P_j}(P_i), \omega^{i,j}_{P_j}(P_i)\} \text{ and } \mathcal{C}^\sigma := \mathcal{M}^{i,j}_{P_j}(P_i) \setminus \{\bot\}.$$
- If $|\Phi| \geq 3$, then let $\sigma_{\text{start}} := (\ell_1, \dots, \ell_{|\Phi|-1})$ and $\sigma_{\text{end}} := (\ell_2, \dots, \ell_{|\Phi|})$. We denote by Q the maximum common subpath of $P_{\ell_{|\Phi|-1}}[\mathcal{T}(\sigma_{\text{start}})]$ and $P_{\ell_{|\Phi|-1}}[\mathcal{T}((\ell_{|\Phi|}, \ell_{|\Phi|-1}))]$. If $\mathcal{T}(\sigma_{\text{start}}) = \{\bot\}$, $\mathcal{T}(\sigma_{\text{end}}) = \{\bot\}$, or $V(Q) = \emptyset$, then let $\mathcal{T}(\sigma) := \mathcal{C}^\sigma := \{\bot\}$. Otherwise, let
$$\begin{aligned} P &:= P_{\ell_{|\Phi|}}[\mathcal{T}(\sigma_{\text{end}})], \\ \mathcal{T}(\sigma) &:= \{\alpha^{\ell_1,\ell_{|\Phi|}}_P(Q), \omega^{\ell_1,\ell_{|\Phi|}}_P(Q)\}, \text{ and} \\ \mathcal{C}^\sigma &:= (\mathcal{M}^{\ell_1,\ell_{|\Phi|}}_P(Q) \cup \mathcal{M}^{\ell_1,\ell_{|\Phi|}}_Q(P)) \setminus \{\bot\}. \end{aligned}$$

The set $\mathcal{C} := \bigcup_\sigma \mathcal{C}^\sigma$ is the *crossing set* of $\mathcal{P}$.

Observation 5.17. *Let $\sigma := (\ell_1, \dots, \ell_{|\Phi|})$ be any permutation of any $\Phi \subseteq [k]$. If $\mathcal{T}(\sigma) \neq \{\bot\}$, then*

(i) $\mathcal{T}(\sigma) \subseteq P_{\ell_{|\Phi|}}$, and

(ii) $\mathcal{T}(\sigma)$ is c-colored for each $c \in \Phi$.

In particular, crossing sets and endpoints are well-defined.

Proof. We prove both claims by an induction over $|\Phi|$. For $|\Phi| = 1$, note that $\mathcal{T}(\sigma) = \{s_{\ell_1}, t_{\ell_1}\}$. Clearly $\{s_{\ell_1}, t_{\ell_1}\} \subseteq P_{\ell_1}$ as these are the ends of P_{ℓ_1} and the pair $\{s_{\ell_1}, t_{\ell_1}\}$ is by definition ℓ_1-colored.

Now assume that both claims hold for all Φ' with $|\Phi'| < |\Phi|$. Since $\mathcal{T}(\sigma) \neq \{\bot\}$, it holds that $\mathcal{T}(\sigma) = \{\alpha^{\ell_1,\ell_{|\Phi|}}_P(Q), \omega^{\ell_1,\ell_{|\Phi|}}_P(Q)\}$, where
$$Q = P_{\ell_{|\Phi|-1}}[\mathcal{T}((\ell_1, \ell_2, \dots, \ell_{|\Phi|-1}))] \cap P_{\ell_{|\Phi|-1}}[\mathcal{T}((\ell_{|\Phi|}, \ell_{|\Phi|-1}))]$$
if $|\Phi| \geq 3$ and $Q = P_{\ell_1}$ if $|\Phi| = 2$. Note that $V(Q) \neq \emptyset$ and hence if $|\Phi| \geq 3$, then by induction hypothesis $Q \subseteq P_{\ell_{|\Phi|-1}}$ and Q is c-colored for each $c \in \Phi \setminus \{\ell_{|\Phi|}\}$. If $|\Phi| = 2$, then $Q = P_{\ell_1} = P_{\ell_{|\Phi|-1}}$ and Q is by definition ℓ_1-colored. Thus,
$$\mathcal{T}(\sigma) = \{\alpha^{\ell_1,\ell_{|\Phi|}}_P(Q), \omega^{\ell_1,\ell_{|\Phi|}}_P(Q)\}$$

is well-defined and hence $\mathcal{T}(\sigma) \subseteq P_{\ell_{|\Phi|}}$. Moreover, by Observation 5.7, it holds that $\mathcal{T}(\sigma)$ is c-colored for each $c \in (\Phi \setminus \{\ell_{|\Phi|}\}) \cup \{\ell_{|\Phi|}\} = \Phi$. □

Note that Observation 5.17 states that, for each sequence $\sigma := (\ell_1, \ell_2, \dots, \ell_{|\Phi|})$, the set $\mathcal{T}(\sigma)$ describes a pair of vertices in $P_{\ell_{|\Phi|}}$. The next lemma states that for any sequence $\sigma' = (\ell_i, \ell_{i+1}, \dots, \ell_{|\Phi|})$ with $i \geq 1$ it holds that the subpath of $P_{\ell_{|\Phi|}}$ between the two vertices in $\mathcal{T}(\sigma)$ is a subpath of the one between the two vertices in $\mathcal{T}(\sigma')$, that is, if we add more entries to the front of σ', then we get smaller and smaller paths.

Lemma 5.18. *Let $\sigma := (\ell_1, \ell_2, \dots, \ell_{|\Phi|})$ be any permutation of any $\Phi \subseteq [k]$ with $|\Phi| \geq 2$. If $\mathcal{T}(\sigma) \neq \{\bot\}$, then $P_{\ell_{|\Phi|}}[\mathcal{T}(\sigma)] \subseteq P_{\ell_{|\Phi|}}[\mathcal{T}((\ell_2, \ell_3, \dots, \ell_{|\Phi|}))]$.*

Proof. We prove the statement by a case distinction over $|\Phi|$. If $|\Phi| = 2$, then the statement is trivial as $\mathcal{T}(\sigma) \subseteq P_{\ell_{|\Phi|}}$ by Observation 5.17 and by Definition 5.7 $\mathcal{T}((\ell_{|\Phi|})) = \{s_{\ell_{|\Phi|}}, t_{\ell_{|\Phi|}}\}$. If $|\Phi| \geq 3$, then note that $P_{\ell_{|\Phi|}}[\mathcal{T}(\sigma)]$ is by definition of α- and ω-vertices the maximal subpath P of $P_{\ell_{|\Phi|}}$ such that there is a subpath

$$Q \subseteq P_{\ell_{|\Phi|-1}}[\mathcal{T}((\ell_1, \ell_2, \dots, \ell_{|\Phi|-1}))] \text{ with } P =^{\{\ell_1, \ell_2, \dots, \ell_{|\Phi|}\}} Q.$$

Analogously, $P_{\ell_{|\Phi|}}[\mathcal{T}((\ell_2, \ell_3, \dots, \ell_{|\Phi|}))]$ is the maximal subpath P' of $P_{\ell_{|\Phi|}}$ such that there is a subpath

$$Q' \subseteq P_{\ell_{|\Phi|-1}}[\mathcal{T}((\ell_2, \ell_3, \dots, \ell_{|\Phi|-1}))] \text{ with } P' =^{\{\ell_2, \ell_3, \dots, \ell_{|\Phi|}\}} Q'.$$

Since $Q \subseteq Q'$ by Definition 5.7, it also holds that $P \subseteq P'$. □

The next lemma states that when "transporting" the labels via a permutation $\sigma = (\ell_1, \ell_2, \dots, \ell_{|\Phi|})$, then the intersecting subpath P in the target path $P_{\ell_{|\Phi|}}$ "agrees" in all coordinates in $\operatorname{set}(\sigma)$ with the subpath Q of $P_{\ell_{|\Phi|-1}}$ where the label is transported from, that is, $P =^{\operatorname{set}(\sigma)} Q$.

Lemma 5.19. *Let $\Phi \subseteq [k]$ with $|\Phi| \geq 2$. Let $\sigma := (\ell_1, \ell_2, \dots, \ell_{|\Phi|})$ be any permutation of Φ. If $\mathcal{T}(\sigma) \neq \{\bot\}$, then $P_{\ell_{|\Phi|}}[\mathcal{T}(\sigma)] =^{\Phi} Q'$ for some subpath Q' of $Q := P_{\ell_{|\Phi|-1}}[\mathcal{T}((\ell_1, \ell_2, \dots, \ell_{|\Phi|-1}))] \cap P_{\ell_{|\Phi|-1}}[\mathcal{T}((\ell_{|\Phi|}, \ell_{|\Phi|-1}))]$.*

Proof. We will again use induction over $|\Phi|$ to prove the claim. For $|\Phi| = 2$, the claim follows from Observation 5.7. For $|\Phi| \geq 3$, let $\sigma_{\text{start}} := (\ell_1, \ell_2, \dots, \ell_{|\Phi|-1})$

and $\sigma_{\text{end}} := (\ell_2, \ell_3, \ldots, \ell_{|\Phi|})$. By Lemma 5.18, $P_{\ell_{|\Phi|}}[\mathcal{T}(\sigma)] \subseteq P_{\ell_{|\Phi|}}[\mathcal{T}(\sigma_{\text{end}})]$ and hence there is by induction hypothesis a subpath

$$R' \subseteq P_{\ell_{|\Phi|-1}}[\mathcal{T}(\sigma_{\text{end}})] \cap P_{\ell_{|\Phi|-1}}[\mathcal{T}((\ell_{|\Phi|}, \ell_{|\Phi|-1}))]$$

with $P_{\ell_{|\Phi|}}[\mathcal{T}(\sigma)] =^{\text{set}(\sigma_{\text{end}})} R'$. Furthermore, by Definition 5.7 $\mathcal{T}((\ell_1, \ell_{|\Phi|})) \neq \perp$ and hence by induction hypothesis there is some subpath

$$Q' \subseteq Q \text{ with } P_{\ell_h}[\mathcal{T}(\sigma)] =^{\ell_1, \ell_h} Q'.$$

Note that $R' =^{\ell_{|\Phi|}} P_{\ell_{|\Phi|}}[\mathcal{T}(\sigma)] =^{\ell_{|\Phi|}} Q'$ and that R' and Q' are both subpaths of Q. Finally, since $Q \subseteq P_{\ell_{|\Phi|-1}}[\mathcal{T}((\ell_{|\Phi|}, \ell_{|\Phi|-1}))]$ is by Observation 5.17 $\ell_{|\Phi|}$-colored and $R' =^{\ell_{|\Phi|}} Q'$, it holds that $R' = Q'$. Thus, $Q' =^{\text{set}(\sigma)} P_{\ell_{|\Phi|}}[\mathcal{T}(\sigma)]$, which proves the claim. □

In Subsection 5.2.1, we defined marbles, that is, specific vertices of two paths P, Q such that when splitting P and Q at these vertices, then each resulting subpath P' of P and Q' of Q fulfill either $P' =^{b,c} Q'$ or P' and Q' are avoiding. In this subsection, we generalized the notion of marbles to more than two paths at the expense of restricting them to solution paths. We conclude this subsection with the notion of *marble paths*, the final link between marbles and crossing sets that will allow us to guess marbles and then compute shortest paths between them almost independently. By that, we mean that we will define labels for each subpath between marbles such that paths with different labels are avoiding and paths with the same labels have a common color. Afterwards, we will show in Section 5.3 how to compute disjoint paths between marble pairs with a common color.

Definition 5.8. An *i-marble path* T is a set of vertices such that $\{s_i, t_i\} \subseteq T$ and for each $u, v \in T$ the pair (u, v) is i-colored. A *segment* S of an i-marble path T is a subset of T containing two vertices denoted by start(S) and end(S) and all vertices $v \in T$ with start(S) $<^i v <^i$ end(S). A segment is *minimal* if it contains exactly two vertices, and it is *j-colored* if (start(S), end(S)) is j-colored. A path P *follows* S if P is i-colored, has end vertices start(S) and end(S), and $S \subseteq V(P)$. Two segments S and S' are *avoiding* if each path P that follows S and each path P' that follows S' are pairwise avoiding. Two marble paths are *avoiding* if all their segments are pairwise avoiding.

Before we state the main result of this section, we will prove a series of lemmata that involve minimal segments of marble paths. The first one states that adding more vertices to avoiding segments still results in avoiding segments.

Lemma 5.20. *Let S be a segment of an i-marble path and let U be a segment of a j-marble path such that S and U are avoiding. Let $S' \supseteq S$ and $U' \supseteq U$ be two segments with*

$$\begin{aligned} \text{start}(S') &= \text{start}(S), & \text{end}(S') &= \text{end}(S), \\ \text{start}(U') &= \text{start}(U), \text{ and} & \text{end}(U') &= \text{end}(U). \end{aligned}$$

The segments S' and U' are avoiding.

Proof. Assume towards a contradiction that S' and U' are not avoiding, that is, there are minimal subsegments S^* of S' and U^* of U' and paths P and Q such that P follows S^* and Q follows U^* and P and Q are not avoiding. Let S'' be the minimal segment in S with

$$\text{start}(S'') \leq^i \text{start}(S^*) < \text{end}(S^*) \leq \text{end}(S'').$$

Note that $S^* \subseteq S''$. Analogously, let U'' be the minimal segment in U with

$$\text{start}(U'') \leq^i \text{start}(U^*) < \text{end}(U^*) \leq \text{end}(U'').$$

Since P follows S^*, it is i-colored and contains all vertices in S^*. Hence it contains all vertices in $S'' \subseteq S^*$ and thus follows S''. Analogously, Q follows U''. Hence S'' and U'' are not avoiding and thus S and U are by definition not avoiding, a contradiction. □

The next two lemmata state that segments of marble paths P and Q defined by vertices in $\mathcal{M}$ are avoiding unless the ends of the segment are between the respective α- and ω-vertices. The first lemma states that if $\alpha_P^{a,b}(Q) = \bot$, then the two marble paths are completely avoiding.

Lemma 5.21. *Let (s_P, t_P) be an a-colored pair and let $\{s_Q, t_Q\}$ be a b-colored pair. Let P be an a-colored s_P-t_P-path and let Q be a b-colored s_Q-t_Q-path. If $\alpha_P^{a,b}(Q) = \bot$, then the marble paths $\mathcal{M}_P^{a,b}(Q)$ and $\mathcal{M}_Q^{a,b}(P)$ are avoiding.*

Proof. Note that since $\alpha_P^{a,b}(Q) = \bot$, it follows that $\{v \in P \mid v \in^{a,b} Q\} = \emptyset$. Assume towards a contradiction that there are segments S of $\mathcal{M}_P^{a,b}(Q)$ and S' of $\mathcal{M}_Q^{a,b}(P)$ that are not avoiding. If S and S' are not minimal, then by definition they contain minimal subsegments that are not avoiding. Hence we can assume without loss of generality that S and S' are minimal.

Let P' be an a-colored path that follows S and let Q' be a b-colored path that follows S' such that P' and Q' are not avoiding. Let further

$$P'' := P[s_P, s_{P'}] \bullet P' \bullet P[t_{P'}, t_P] \text{ and } Q'' := Q[s_Q, s_{Q'}] \bullet Q' \bullet Q[t_{Q'}, t_Q].$$

Note that P'' follows $\mathcal{M}_P^{a,b}(Q)$ and therefore $\mathcal{M}_P^{a,b}(Q) \subseteq P''$. Analogously, Q'' follows $\mathcal{M}_Q^{a,b}(P)$ and hence $\mathcal{M}_Q^{a,b}(P) \subseteq Q''$. By Proposition 5.14, it holds that

$$\{v \in P'' \mid v \in^{a,b} Q''\} = \{v \in P \mid v \in^{a,b} Q\} = \emptyset,$$

that is, P'' and Q'' are avoiding. Since P'' and Q'' are avoiding, so are all subpaths of P'' and Q''. Thus P' and Q' are avoiding, a contradiction. ☐

The next lemma deals with the case where $\alpha_P^{a,b}(Q) \neq \perp$. Recall that in this case we only consider segments that do not contain $\alpha_P^{a,b}(Q)$ or $\omega_P^{a,b}(Q)$.

Lemma 5.22. *Let (s_P, t_P) be an a-colored pair and let (s_Q, t_Q) be a b-colored pair. Let P be an a-colored s_P-t_P-path and let Q be a b-colored s_Q-t_Q-path. If $\alpha_P^{a,b}(Q) \neq \perp$, then let S_1 and S_2 be segments of the marble path $\mathcal{M}_P^{a,b}(Q)$ with $\operatorname{start}(S_1) = s_P, \operatorname{end}(S_1) = \alpha_P^{a,b}(Q), \operatorname{start}(S_2) = \omega_P^{a,b}(Q)$, and $\operatorname{end}(S_2) = t_P$. Let further S' be a segment of the marble paths $\mathcal{M}_Q^{a,b}(P)$. Then S_1 and S' are avoiding and so are S_2 and S'.*

Proof. Note that $\alpha_P^{a,b}(Q) \neq \perp$, Observation 5.7 and Lemma 5.10 imply that

$$\{v \in P \mid v \in^{a,b} Q\} = P[\alpha_P^{a,b}(Q), \omega_P^{a,b}(Q)].$$

Assume towards a contradiction that S_1 and S' are not avoiding or S_2 and S' are not avoiding. Then there are paths P_1 that follows S_1, P_2 that follows S_2 and Q' that follows S' such that P_1 and Q' are not avoiding or P_2 and Q' are not avoiding. Hence there are vertices v in Q' and w in P_1 or P_2 that are inner vertices with $v =^{a,b} w$. Let

$$P^* := P_1 \bullet P[\alpha_P^{a,b}(Q), \omega_P^{a,b}(Q)] \bullet P_2 \text{ and } Q^* := Q[s_Q, s_{Q'}] \bullet Q' \bullet Q[t_{Q'}, t_Q].$$

Note that P^* is a-colored as each of its subpaths is a-colored. Moreover, P^* follows $\mathcal{M}_P^{a,b}(Q)$ as it contains $s_P, \alpha_P^{a,b}(Q), \omega_P^{a,b}(Q)$, and t_P. Analogously, Q^* follows $\mathcal{M}_Q^{a,b}(P)$. Proposition 5.14 then states that

$$\{v \in P^* \mid v \in^{a,b} Q'\} = \{v \in P \mid v \in^{a,b} Q\} = P[\alpha_P^{a,b}(Q), \omega_P^{a,b}(Q)].$$

Thus, $w \in P[\alpha_P^{a,b}(Q), \omega_P^{a,b}(Q)]$ which is a contradiction to the assumption that w is an interior vertex of the a-colored paths P_1 or P_2. ☐

The final lemma generalizes the two previous ones from $\mathcal{M}$ (comparison of two paths) to $\mathcal{C}$ (sequences of paths). Unfortunately, it contains a lot of rather tedious case distinctions. We remark that solving the respective cases is not particularly difficult or interesting.

Lemma 5.23. *Let $(G,(s_i,t_i)_{i\in[k]})$ be an instance of k-Disjoint Shortest Paths, let $\mathcal{P} := (P_i)_{i\in[k]}$ be a solution to this instance, and let $\Phi \subseteq [k]$. Let $\sigma := (\ell_1,\ell_2,\ldots,\ell_{|\Phi|})$ be a permutation of Φ, let $\sigma_{\text{start}} := (\ell_1,\ell_2,\ldots,\ell_{|\Phi|-1})$, and let $\mathcal{C}$ be the crossing set of $\mathcal{P}$. Let $g := \ell_1, i := \ell_{|\Phi|-1}$, and $j := \ell_{|\Phi|}$.*

(i) If $\mathcal{T}(\sigma) = \{\bot\}$ and $\mathcal{T}(\sigma_{\text{start}}) \neq \{\bot\}$, then the two marble paths

$$V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C} \text{ and } V(P_j) \cap \mathcal{C}$$

are avoiding.

(ii) If $\mathcal{T}(\sigma) = \{u,v\} \neq \{\bot\}$ with $u <^j v$, then the two marble paths

$$V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C} \text{ and } V(P_j[s_j,u]) \cap \mathcal{C}$$

are avoiding and so are

$$V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C} \text{ and } V(P_j[v,t_j]) \cap \mathcal{C}.$$

Proof. We will prove both claims by induction over $|\Phi|$.
Base case: Let $|\Phi| = 2$ and hence $g = i$ and $P_i[\mathcal{T}(\sigma_{\text{start}})] = P_i$.

(i) Since $\mathcal{T}(\sigma) = \{\bot\}$, it holds that $\alpha^{i,j}_{P_j}(P_i) = \bot$. By Definition 5.7, it holds that $\mathcal{M}^{i,j}_{P_i}(P_j) \subseteq V(P_i) \cap \mathcal{C}$ and $\mathcal{M}^{i,j}_{P_j}(P_i) \subseteq V(P_j) \cap \mathcal{C}$. By Lemma 5.21, $\mathcal{M}^{i,j}_{P_i}(P_j)$ and $\mathcal{M}^{i,j}_{P_j}(P_i)$ are avoiding. Lemma 5.20 states $V(P_i) \cap \mathcal{C}$ and $V(P_j) \cap \mathcal{C}$ are avoiding since

$$\begin{aligned} \text{start}(V(P_i) \cap \mathcal{C}) &= s_i = \text{start}(\mathcal{M}^{i,j}_{P_i}(P_j)), \\ \text{end}(V(P_i) \cap \mathcal{C}) &= t_i = \text{end}(\mathcal{M}^{i,j}_{P_i}(P_j)), \\ \text{start}(V(P_j) \cap \mathcal{C}) &= s_j = \text{start}(\mathcal{M}^{i,j}_{P_j}(P_i)), \text{ and} \\ \text{end}(V(P_j) \cap \mathcal{C}) &= t_j = \text{end}(\mathcal{M}^{i,j}_{P_j}(P_i)). \end{aligned}$$

(ii) Since $\sigma = (i,j)$, it holds that $u = \alpha^{i,j}_{P_j}(P_i)$ and $v = \omega^{i,j}_{P_j}(P_i)$. By Definition 5.7, it holds that $\mathcal{M}^{i,j}_{P_i}(P_j) \subseteq V(P_i) \cap \mathcal{C}$ and $\mathcal{M}^{i,j}_{P_j}(P_i) \subseteq V(P_j) \cap \mathcal{C}$. By Lemma 5.22, $\mathcal{M}^{i,j}_{P_i}(P_j)$ and $\{s_j, u\}$ are avoiding and so are $\mathcal{M}^{i,j}_{P_i}(P_j)$ and $\{v, t_j\}$. Thus the claim again follows from Lemma 5.20 and

$$\begin{aligned}
\text{start}(V(P_i) \cap \mathcal{C}) &= s_i = \text{start}(\mathcal{M}^{i,j}_{P_i}(P_j)),\\
\text{end}(V(P_i) \cap \mathcal{C}) &= t_i = \text{end}(\mathcal{M}^{i,j}_{P_i}(P_j)),\\
\text{start}(V(P_j[s_j,u]) \cap \mathcal{C}) &= s_j = \text{start}(\{s_j,u\}),\\
\text{end}(V(P_j[s_j,u]) \cap \mathcal{C}) &= u = \text{end}(\{s_j,u\}),\\
\text{start}(V(P_j[v,t_j]) \cap \mathcal{C}) &= v = \text{start}(\{v,t_j\}), \text{ and}\\
\text{end}(V(P_j[v,t_j]) \cap \mathcal{C}) &= t_j = \text{end}(\{v,t_j\}).
\end{aligned}$$

Induction step: Let $|\Phi| \geq 3$ and assume that the statement holds for all $\Phi' \subseteq [k]$ with $2 \leq |\Phi'| < |\Phi|$. Let $\sigma_{\text{end}} \coloneqq (\ell_2, \ell_3, \ldots, \ell_{|\Phi|})$ and $\sigma' \coloneqq (\ell_2, \ell_3, \ldots, \ell_{|\Phi|-1})$.

(i) Since $\mathcal{T}(\sigma) = \{\perp\}$ and $\mathcal{T}(\sigma_{\text{start}}) \neq \{\perp\}$, by Definition 5.7, there are three possible cases:

$$\begin{aligned}
\mathcal{T}(\sigma_{\text{end}}) &= \{\perp\},\\
V(Q) = V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap V(P_i[\mathcal{T}((j,i))]) &= \emptyset, \text{ or}\\
\alpha^{g,j}_{P_j[\mathcal{T}(\sigma_{\text{end}})]}(Q) &= \perp.
\end{aligned}$$

We will show that $V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C}$ and $V(P_j) \cap \mathcal{C}$ are avoiding in each of the three cases.

(1) We start with the case where $\mathcal{T}(\sigma_{\text{end}}) = \{\perp\}$. Since $\mathcal{T}(\sigma_{\text{start}}) \neq \{\perp\}$ it holds by Lemma 5.18 that

$$\emptyset \neq V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \subseteq V(P_i[\mathcal{T}(\sigma')])$$

and in particular, $\mathcal{T}(\sigma') \neq \{\perp\}$. Since $\mathcal{T}(\sigma_{\text{end}}) = \{\perp\}$, $\mathcal{T}(\sigma') \neq \{\perp\}$, and σ_{end} is the permutation of a set Φ' with $|\Phi'| < |\Phi|$, the induction hypothesis states that

$$V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C} \text{ and } V(P_j) \cap \mathcal{C}$$

are avoiding. By definition, each subsegment of $V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C}$ is also avoiding $V(P_j) \cap \mathcal{C}$ and since $V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \subseteq V(P_i[\mathcal{T}(\sigma')])$ and $\mathcal{T}(\sigma_{\text{start}}) \subseteq \mathcal{C}$, it holds that $V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C}$ is a subsegment of $V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C}$.

(2) We continue with the case where

$$V(Q) := V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap V(P_i[\mathcal{T}((j,i))]) = \emptyset.$$

We consider the sequence (j,i) and the two cases $\mathcal{T}((j,i)) = \{\perp\}$ and $\mathcal{T}((j,i)) \neq \{\perp\}$. Since $|\{j,i\}| = 2 < |\Phi|$, the induction hypothesis states that if $\mathcal{T}((j,i)) = \{\perp\}$, then

$$V(P_j) \cap \mathcal{C} \text{ and } V(P_i) \cap \mathcal{C} \text{ are avoiding}$$

and if $\mathcal{T}((j,i)) \neq \{\perp\}$, then

$V(P_j) \cap \mathcal{C}$ avoids both $V(P_i[s_i, \alpha^{i,j}_{P_i}(P_j)]) \cap \mathcal{C}$ and $V(P_i[\omega^{i,j}_{P_i}(P_j), t_i]) \cap \mathcal{C}$.

In the former case, since $V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C}$ is a segment of $V(P_i) \cap \mathcal{C}$, it holds by definition that $V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C}$ and $V(P_j) \cap \mathcal{C}$ are avoiding. In the latter case, since $V(Q) = \emptyset$, it holds that

$$V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \subseteq V(P_i[s_i, \alpha^{i,j}_{P_i}(P_j)]) \text{ or}$$
$$V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \subseteq V(P_i[\omega^{i,j}_{P_i}(P_j), t_j]).$$

Since the two cases are analogous, we assume without loss of generality the former, that is, $V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \subseteq V(P_i[s_i, \alpha^{i,j}_{P_i}(P_j)])$.

Since $V(P_i[s_i, \alpha^{i,j}_{P_i}(P_j)]) \cap \mathcal{C}$ and $V(P_j) \cap \mathcal{C}$ are avoiding and since

$$V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C} \subseteq V(P_i[s_i, \alpha^{i,j}_{P_i}(P_j)]) \cap \mathcal{C},$$

by definition $V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C}$ and $V(P_j) \cap \mathcal{C}$ are also avoiding.

(3) It remains to analyze the case where $\alpha^{g,j}_{P_j[\mathcal{T}(\sigma_{\text{end}})]}(Q) = \perp$. We assume that $\mathcal{T}(\sigma_{\text{end}}) \neq \{\perp\}$ and $V(Q) \neq \emptyset$ as we can otherwise use the proofs above. Assume towards a contradiction that $V(P_j) \cap \mathcal{C}$ and $V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C}$ are not avoiding. Then there are minimal segments $S_i \subseteq V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C}$ and $S_j \subseteq V(P_j) \cap \mathcal{C}$ that are not avoiding. We consider the two cases

$$S_i \subseteq V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap V(P_i[\mathcal{T}((j,i))]) = V(Q) \text{ and}$$
$$S_i \subseteq V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \setminus V(P_i[\mathcal{T}((j,i))]).$$

Note that $\mathcal{T}((j,i)) \subseteq \mathcal{C}$ and that S_i is minimal and hence this case distinction is complete. In the latter case, note that since S_i and S_j are not avoiding, there is a path R_i that follows S_i and a path R_j that follows S_j such that $\{v \in R_i \mid v \in^{i,j} R_j\} \setminus S_i \neq \emptyset$. Then, it holds by Observation 5.7 $\{v \in P_i \mid v \in^{i,j} P_j\} \subseteq V(P_i[\mathcal{T}((j,i))])$. Moreover, by Proposition 5.14 $\{v \in R_i \mid v \in^{i,j} R_j\} \subseteq^{i,j} V(P_i[\mathcal{T}((j,i))])$. Hence

$$\{v \in R_i \mid v \in^{i,j} R_j\} \subseteq^{a,b} S_i,$$

a contradiction.

If $S_i \subseteq V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap V(P_i[\mathcal{T}((j,i))]) = V(Q)$, then we distinguish between the two cases

$$S_j \subseteq V(P_j[\mathcal{T}(\sigma_{\text{end}})]) \cap \mathcal{C} \text{ and } S_j \setminus V(P_j[\mathcal{T}(\sigma_{\text{end}})]) \neq \emptyset.$$

In the former case, it holds by Lemma 5.21 that $\mathcal{M}^{g,j}_{P_j[\mathcal{T}(\sigma_{\text{end}})]}(Q)$ and $\mathcal{M}^{g,j}_{Q}(P_j[\mathcal{T}(\sigma_{\text{end}})])$ are avoiding. Since

$$\begin{aligned} \mathcal{M}^{g,j}_{P_j[\mathcal{T}(\sigma_{\text{end}})]}(Q) &\subseteq V(P_j[\mathcal{T}(\sigma_{\text{end}})]) \cap \mathcal{C} \text{ and} \\ \mathcal{M}^{g,j}_{Q}(P_j[\mathcal{T}(\sigma_{\text{end}})]) &\subseteq V(Q) \cap \mathcal{C}, \end{aligned}$$

it holds that S_i and S_j are avoiding, a contradiction.

Finally, it remains to analyze the case where $S_j \setminus V(P_j[\mathcal{T}(\sigma_{\text{end}})]) \neq \emptyset$. Since $\mathcal{T}(\sigma_{\text{start}}) \neq \{\bot\}$ it holds by Lemma 5.18 that

$$\emptyset \neq V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \subseteq V(P_i[\mathcal{T}(\sigma')]).$$

and in particular, $\mathcal{T}(\sigma') \neq \{\bot\}$. Since by assumption $\mathcal{T}(\sigma_{\text{end}}) \neq \{\bot\}$, the induction hypothesis states that

$$V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C} \text{ and } V(P_j[s_j, \text{start}(\mathcal{T}(\sigma_{\text{end}}))]) \cap \mathcal{C}$$

are avoiding and so are

$$V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C} \text{ and } V(P_j[\text{end}(\mathcal{T}(\sigma_{\text{end}})), t_j]) \cap \mathcal{C}.$$

Since, by Lemma 5.18,

$$S_i \subseteq V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C} \subseteq V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C}$$

and since

$$S_j \subseteq V(P_j[s_j, \mathrm{start}(\mathcal{T}(\sigma_{\mathrm{end}}))]) \cap \mathcal{C} \text{ or}$$
$$S_j \subseteq V(P_j[\mathrm{end}(\mathcal{T}(\sigma_{\mathrm{end}})), t_j]) \cap \mathcal{C}$$

it follows that S_i and S_j are avoiding, a contradiction.

(ii) In this case it holds that $\mathcal{T}(\sigma) = \{u, v\} \neq \{\bot\}$ with $u <^j v$ and it remains to show that

$$V(P_i[\mathcal{T}(\sigma_{\mathrm{start}})]) \cap \mathcal{C} \text{ and } V(P_j[s_j, u]) \cap \mathcal{C}$$

are avoiding and so are

$$V(P_i[\mathcal{T}(\sigma_{\mathrm{start}})]) \cap \mathcal{C} \text{ and } V(P_j[v, t_j]) \cap \mathcal{C}.$$

Since both cases are analogous, we will only show that $V(P_i[\mathcal{T}(\sigma_{\mathrm{start}})]) \cap \mathcal{C}$ and $V(P_j[s_j, u]) \cap \mathcal{C}$ are avoiding. To this end, assume towards a contradiction that there are minimal segments

$$S_i \subseteq V(P_i[\mathcal{T}(\sigma_{\mathrm{start}})]) \cap \mathcal{C} \text{ and } S_j \subseteq V(P_j[s_j, u]) \cap \mathcal{C}$$

that are not avoiding.

We consider the two cases $S_i \setminus V(P_i[\mathcal{T}((j,i))]) \neq \emptyset$ and $S_i \subseteq V(P_i[\mathcal{T}((j,i))])$. In the former case, note that if $\{w, x\} := \mathcal{T}((j,i)) \neq \{\bot\}$ with $w <^i x$, then it holds by Lemma 5.22 that $V(P_i[s_i, w]) \cap \mathcal{C}$ and $V(P_j) \cap \mathcal{C}$ are avoiding. If $\mathcal{T}((j,i)) = \{\bot\}$, then it holds by Lemma 5.21 that $V(P_i) \cap \mathcal{C}$ and $V(P_j) \cap \mathcal{C}$ are avoiding. Hence in both cases S_i and S_j are avoiding, a contradiction.

Now assume that $S_i \subseteq V(P_i[\mathcal{T}((j,i))])$. Since $S_i \subseteq V(P_i[\mathcal{T}(\sigma_{\mathrm{start}})])$, it holds by Lemma 5.18 that $\emptyset \neq V(P_i[\mathcal{T}(\sigma_{\mathrm{start}})]) \subseteq V(P_i[\mathcal{T}(\sigma')])$ and that

$$S_i \subseteq V(Q) = V(P_i[\mathcal{T}(\sigma_{\mathrm{start}})]) \cap V(P_i[\mathcal{T}((j,i))]).$$

Note that if in this case $\mathcal{T}(\sigma_{\mathrm{end}}) = \{\bot\}$, then by induction hypothesis $V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C}$ and $V(P_j) \cap \mathcal{C}$ are avoiding, and thus so are

$$V(P_i[\mathcal{T}(\sigma_{\mathrm{start}})]) \cap \mathcal{C} \supseteq S_i \text{ and } V(P_j) \cap \mathcal{C} \supseteq S_j,$$

a contradiction.

It remains to analyze the case where $\{y, z\} := \mathcal{T}(\sigma_{\text{end}}) \neq \{\bot\}$. We resolve this case with a final case distinction:

$$S_j \setminus V(P_j[\mathcal{T}(\sigma_{\text{end}})]) \neq \emptyset \text{ or } S_j \subseteq V(P_j[\mathcal{T}(\sigma_{\text{end}})]).$$

In the former case $S_j \subseteq V(P_j[s_j, y])$ or $S_j \subseteq V(P_j[z, t_j])$. By Lemma 5.18, it holds that $S_i \subseteq V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \subseteq V(P_i[\mathcal{T}(\sigma')])$. Since by Lemma 5.22

$$\begin{gathered} V(P_j[s_j, y]) \cap \mathcal{C} \text{ and } V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C} \text{ and} \\ V(P_j[z, t_j]) \cap \mathcal{C} \text{ and } V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C} \end{gathered}$$

are avoiding, we conclude that S_i and S_j are avoiding, a contradiction.

Finally, if $S_j \subseteq V(P_j[\mathcal{T}(\sigma_{\text{end}})])$, then it holds by induction hypothesis and Lemma 5.18 that

$$S_i \subseteq V(P_i[\mathcal{T}(\sigma_{\text{start}})]) \cap \mathcal{C} \subseteq V(P_i[\mathcal{T}(\sigma')]) \cap \mathcal{C} \text{ and } S_j \subseteq V(P_j[\mathcal{T}(\sigma_{\text{end}})]) \cap \mathcal{C}$$

are avoiding, a contradiction. □

We conclude this section with the definition of labels of segments and the proof that they guarantee that paths following two segments have either a common color or are avoiding. To this end, let $S = \{u, v\}$ be a segment of an i-marble pathwith $u <^i v$. The set of *labels* of S (labels$[S]$) is defined as

$$\{a \mid \exists \sigma := (\ell_1 = a, \ell_2, \ldots, \ell_{|\sigma|} = i).\ \{\alpha, \omega\} := \mathcal{T}(\sigma) \neq \{\bot\} \wedge \alpha \leq^i u <^i v \leq^i \omega\}.$$

Proposition 5.24. *Let $(G, (s_i, t_i)_{i \in [k]})$ be an instance of k-Disjoint Shortest Paths and let $\mathcal{P} = (P_i)_{i \in [k]}$ be a solution to this instance. Let $i, j \in [k]$ and let $T_i = V(P_i) \cap \mathcal{C}$ be an i-marble path and $T_j \subseteq V(P_j) \cap \mathcal{C}$ be a j-marble path. Let $S_i \subseteq T_i$ and $S_j \subseteq T_j$ be two minimal segments. If* labels$[S_i] \neq$ labels$[S_j]$, *then S_i and S_j are avoiding.*

Proof. We start with the case where $i \notin$ labels$[S_j]$. Then either $\mathcal{T}((i, j)) = \{\bot\}$ or $S_j \cap T_j[u, v] = \emptyset$, where $\{u, v\} := \mathcal{T}((i, j)) \neq \{\bot\}$. In both cases, S_j and S_i are avoiding by Lemma 5.23. The case where $j \notin$ labels$[S_i]$ is analogous.

It remains to consider the case where

$$i, j \in \text{labels}[S_i] \cap \text{labels}[S_j].$$

Let without loss of generality be $d \in$ labels$[S_i] \setminus$labels$[S_j]$. By definition of labels, there is a set $\Phi = \{\ell_1, \ell_2, \ldots, \ell_{|\Phi|}\}$ and a permutation $\sigma = (\ell_1, \ell_2, \ldots, \ell_{|\Phi|})$ of Φ

such that $\ell_1 = d$, $\ell_{|\Phi|} = i$, $\mathcal{T}(\sigma) = \{\alpha, \omega\} \neq \{\perp\}$, and $S_i \subseteq P_i[\alpha, \omega] \cap \mathcal{C}$. We consider the two cases $j \notin \Phi$ and $j \in \Phi$.

If $j \notin \Phi$, then let $\sigma' := (d = \ell_1, \ell_2, \ldots, \ell_{|\Phi|} = i, j)$ and we distinguish between the two cases $\mathcal{T}(\sigma') \neq \{\perp\}$ and $\mathcal{T}(\sigma') = \{\perp\}$. If $\mathcal{T}(\sigma') \neq \{\perp\}$, then by definition of labels and since $d \notin \text{labels}[S_j]$, it holds that $S_j \setminus V(P_j[\mathcal{T}(\sigma')]) \neq \emptyset$. Lemma 5.23 states that S_j and each minimal segment of $V(P_i[\mathcal{T}(\sigma)]) \cap \mathcal{C}$ are avoiding. Hence, S_i and S_j are avoiding as $S_i \subseteq V(P_i[\mathcal{T}(\sigma)]) \cap \mathcal{C}$. If $\mathcal{T}(\sigma') = \{\perp\}$, then, by Lemma 5.23, it holds that $V(P_i[\mathcal{T}(\sigma)]) \cap \mathcal{C}$ and $T_j := V(P_j) \cap \mathcal{C}$ are avoiding. Thus by definition S_i and S_j are avoiding.

It remains to consider the case where $j \in \Phi = \{\ell_1, \ell_2, \ldots, \ell_{|\Phi|}\}$. In this case let $x \in [2, |\Phi| - 1]$ such that $j = \ell_x$ and let $\sigma_i := (d = \ell_1, \ell_2, \ldots, \ell_i)$ for all $h \in [x, |\Phi|]$ ($x \leq h \leq |\Phi|$). Note that $\sigma_x := (d = \ell_1, \ell_2, \ldots, \ell_x = j)$ and $\sigma_{|\Phi|} = \sigma$. Since $S_i \subseteq V(P_i[\mathcal{T}(\sigma)]) \cap \mathcal{C}$, it follows that $\mathcal{T}(\sigma) \neq \{\perp\}$ and, by definition of $\mathcal{T}$, it holds for each $h \in [x, |\Phi|]$ that $\mathcal{T}(\sigma_h) \neq \{\perp\}$. Lemma 5.19 then states that for each $h \in [x, |\Phi|]$ and each subpath $Q_{|\Phi|}$ of $P_i[\mathcal{T}(\sigma_{|\Phi|})] = P_i[\mathcal{T}(\sigma)]$ there is some subpath Q_h of $P_{\ell_h}[\mathcal{T}(\sigma_h)]$ such that $Q_h =^{\text{set}(\sigma_h)} Q_{h-1}$. Let $Q_{|\Phi|}$ be such a path with $\{s_{Q_{|\Phi|}}, t_{Q_{|\Phi|}}\} = S_i$. Thus,

$$Q_{|\Phi|} =^j Q_{|\Phi|-1} =^j \ldots =^j Q_x$$

and in particular $\{\text{start}(S_i), \text{end}(S_i)\} \subseteq^j V(P_j[\mathcal{T}(\sigma_x)])$.

Since $S_i \subseteq V(P_i[\mathcal{T}(\sigma)]) \cap \mathcal{C}$, it holds by Lemma 5.18 that

$$S_i \subseteq V(P_i[\mathcal{T}(\sigma)]) \cap \mathcal{C} \subseteq V(P_i[\mathcal{T}((\ell_x, \ell_{x+1}, \ldots, \ell_{|\Phi|}))]) \cap \mathcal{C}.$$

Hence, it holds that $\{\text{start}(S_i), \text{end}(S_i)\}$ is j-colored and thus it holds for each path Q_i that follows S_i that $Q_i \subseteq^j V(P_j[\mathcal{T}(\sigma_x)])$. Since $d \in \text{labels}[S'_j]$ for each $S'_j \subseteq V(P_j[\mathcal{T}(\sigma_x)]) \cap \mathcal{C}$, $d \notin \text{labels}[S_j]$, and S_j is minimal, it follows that

$$S_j \cap (V(P_j[\mathcal{T}(\sigma_x)]) \cap \mathcal{C}) \subseteq \{\text{start}(S_i), \text{end}(S_i)\} \cap \mathcal{T}(\sigma_x).$$

Moreover, since P_j is strictly increasing in the j^{th} coordinate, it follows for each path Q_i that follows S_i and each path Q_j that follows S_j that

$$\overrightarrow{V(Q_i)}^j \cap \overrightarrow{V(Q_j)}^j \subseteq \overrightarrow{V(Q_i)}^j \cap \overrightarrow{V(P_j[\mathcal{T}(\sigma_x)])}^j \subseteq (\{\overrightarrow{s_{Q_i}}, \overrightarrow{t_{Q_i}}\} \cap \{\overrightarrow{s_{Q_j}}, \overrightarrow{t_{Q_j}}\})^j.$$

Hence, each such pair of paths is avoiding (no two inner vertices share the same vector) and thus it holds by definition that S_i and S_j are avoiding. □

5.3 An *XP*-Algorithm for k-Disjoint Shortest Paths

In this section, we present our main theorem, that is, an *XP*-algorithm for k-Disjoint Shortest Paths with respect to the number k of terminal pairs. In a nutshell, we first guess all marble paths T_i and the respective ends $\mathcal{T}$ corresponding to the crossing set $\mathcal{C}$ of some solution (if one exists). We then compute all minimal segments of each marble path T_i, compute their respective labels, and partition the segments such that all minimal segments in the same part of the partition are strictly monotone in a common coordinate and two minimal segments in distinct parts of the partition are avoiding. The crucial improvement over the algorithm by Lochet [Loc21] is that our partition is much smaller. Afterwards, we find via dynamic programming for all segments in one part of the partition disjoint paths that follow the respective segments.

To this end, we introduce c-layered DAGs and the problem p-Disjoint Paths on c-layered DAGs. For a graph G with vectors $\vec{v}$ for all $v \in V$ (as defined in Subsection 5.2.1), the *c-layered DAG D_c* of G is the directed graph $D_c = (V, A)$, where $A = \{(x, y) \mid \{x, y\} \in E(G) \wedge \ \vec{y}^{\,c} - \vec{x}^{\,c} = 1\}$. Notice that a path $P = (v_1, v_2, \dots, v_p)$ is c-colored if and only if $\overrightarrow{v_{i+1}}^{c} - \vec{v_i}^{\,c} = 1$ for all $i \in [p-1]$ or $\vec{v_i}^{\,c} - \overrightarrow{v_{i+1}}^{c} = 1$ for all $i \in [p-1]$. Let $P_m = (v_p, v_{p-1}, \dots, v_1)$ be the mirrored path of P. Then, P is c-colored if and only if P_m is and hence if and only if the directed path $(V(P), A(P))$ or the directed path $(V(P), A(P_m))$ is a path in D_c. Finally, observe that $A(P_m) = A^{-1}(P)$, that is, P_m and P have the same vertices but the edges are oppositely directed.

Observation 5.25. *A path P in G is c-colored if and only if $(V(P), A(P))$ or $(V(P), A^{-1}(P)$ is a path in the c-layered DAG D_c of G.*

We continue with a definition of p-Disjoint Paths on c-layered DAGs. Here, we are given a c-layered DAG D_c and a list $(s_i, t_i)_{i \in [p]}$ of (possibly intersecting) terminal pairs. We then ask whether there are pairwise internally vertex-disjoint s_i-t_i-path in D_c. Formally, it is defined as follows.

p-Disjoint Paths on c-layered DAGs
Input: A c-layered DAG D_c and p pairs $(s_i, t_i)_{i \in [k]}$ of vertices.
Question: Are there p internally vertex-disjoint paths P_i in D_c such that P_i is a shortest s_i-t_i-path for each $i \in [p]$?

With these definitions, we can state our algorithm. Algorithm 5.1 provides pseudo-code.

Algorithm 5.1: Our algorithm for k-Disjoint Shortest Paths.

```
function solve($G$, $(s_i, t_i)_{i\in[k]}$)
    foreach guess $(T_i)_{i\in[k]}$, Ends of the crossing set do
        /* We assume subsequently that the guesses are correct, that is, if
           there is a solution $\mathcal{P} := (P_i)_{i\in[k]}$, then $T_i = V(P_i) \cap \mathcal{C}$ for
           all $i \in [k]$ and Ends $= \mathcal{T}$.                                */
        foreach $i \in [k]$ do
            $\mathcal{P}_i \leftarrow \emptyset$ // $P_i$ contains all segments corresponding to $D_i$
        foreach minimal segment $S$ of some $T_i$ with $i \in [k]$ do
            marks$[S_i] \leftarrow \emptyset$
            foreach permutation $\sigma = (\ell_1, \ell_2, \dots, i)$
              with Ends$(\sigma) = \{\alpha, \omega\} \neq \{\perp\}$
              and $\alpha \leq^i$ start$(S_i) <^i$ end$(S_i) \leq^i \omega$ do
                marks$[S_i] \leftarrow$ marks$[S_i] \cup \mathrm{set}(\sigma)$
            $j \leftarrow \min \mathrm{marks}[S]$
            $x \leftarrow \arg\min\{\overrightarrow{v}^j \mid v \in \{\mathrm{start}(S), \mathrm{end}(S)\}\}$
            $y \leftarrow \arg\max\{\overrightarrow{v}^j \mid v \in \{\mathrm{start}(S), \mathrm{end}(S)\}\}$
            $\mathcal{P}_j = \mathcal{P}_j \cup \{(x, y)\}$
        foreach $j \in [k]$ do
            Order $\mathcal{P}_j = ((x_1, y_1), (x_2, y_2), \dots)$ such that $\overrightarrow{x_1}^j \leq \overrightarrow{x_2}^j \leq \dots$
        if all instances $(D_i, \mathcal{P}_i)$ of $|\mathcal{P}_i|$-Disjoint Paths on $i$-layered
          DAGs are yes-instances and the combined solutions form a
          solution of $k$-Disjoint Shortest Paths then
            return true
    return false
```

Fortune et al. [FHW80] showed that p-Disjoint Path on DAGs can be solved in $n^{O(p)}$ time. Since c-layered DAGs are DAGs, we could use their algorithm in Algorithm 5.1 and achieve a running time of $n^{O(k!)}$. However, to drop the Landau notation in the exponent, we show that p-Disjoint Paths on c-layered DAGs can be solved in $O(n^{p+1})$ time. Afterwards, we show that Algorithm 5.1 is correct and runs in $O(n^{16k+k!+k+1})$ time.

The idea behind the dynamic program for p-Disjoint Paths on c-layered DAGs is as follows. Given an i-layered DAG D_i, a number p, and a set of terminal pairs $(s_j, t_j), j \in [p]$, where $\overrightarrow{s_j}^i < \overrightarrow{t_j}^i$ and $\overrightarrow{s_j}^i \leq \overrightarrow{s_\ell}^i$ for all $j < \ell \in [p]$, the dynamic program is a table $T[x_1, x_2, \ldots, x_p] \in \{\text{true}, \text{false}\}$ that stores true roughly if the following two criteria are fulfilled.

i) All x_i, x_j are pairwise different or $x_i \in \{s_i, t_i\}$ and $x_j \in \{s_j, t_j\}$, and

ii) there is a set of internally vertex-disjoint s_j-x_j paths.

Thus, $T[t_1, t_2, \ldots, t_p] = \text{true}$ if and only if there is a set of internally vertex-disjoint s_j-t_j-paths.

Note that if $s_i =^c s_j$ and $t_i =^c t_j$ for all $i, j \in [p]$, then there is a fairly straightforward dynamic program for p-Disjoint Paths on c-layered DAGs. Store for increasing values of $d \in [\overrightarrow{s_i}^c, \overrightarrow{t_i}^c]$ and for each tuple $(x_1, x_2, \ldots, x_p)$ of vertices with $\overrightarrow{x_i}^c = d$ for all $i \in [p]$ whether there are pairwise disjoint paths from s_i to x_i (the paths may possibly share their end vertices s_i and/or t_i if $x_i = t_i$). The table corresponding to this dynamic program has $O(n \cdot n^p)$ table entries (at most n values for d and for each d there are at most n^p sequences of p vertices). Each table entry can be computed in $O(n^p)$ time by iterating over all table entries for $d-1$ and $(x'_1, x'_2, \ldots, x'_p)$ and checking whether $(x'_1, x_1), (x'_2, x_2), \ldots, (x'_p, x_p) \in A$. This would lead to an overall running time of $O(n^{2p+1})$ for p-Disjoint Paths on c-layered DAGs. Note further that ensuring that $s_i =^c s_j$ and $t_i =^c t_j$ is not difficult either. One can simply replace each s_i and t_i with new terminals and add paths of according lengths between the new and the old terminal vertices. An example of this roughly outlined construction is given in Figure 5.5. However, there is another dynamic program that is faster ($O(n^{p+1})$ time instead of $O(n^{2p+1})$ time) and that also works for general DAGs. Basically, instead of moving all x_i from one layer to the next in one step, we order them and move the x_i that is first in this ordering. This has the advantage that for computing one table entry, we only have to consider $O(n)$ table entries instead of $O(n^p)$.

Lemma 5.26. *An instance of p-Disjoint Paths on DAGs on a graph with n vertices can be solved in $O(n^{p+1})$ time.*

Proof. Let $D = (V, A)$ be a DAG and let $(s_i, t_i)_{i \in [p]}$ be a set of p terminal pairs. We define $V^{\text{end}} := \bigcup_{i \in [p]} \{s_i, t_i\}$ to be the set of all terminals. We also choose an arbitrary topological order of D and denote by $u \prec v$ that u comes before v

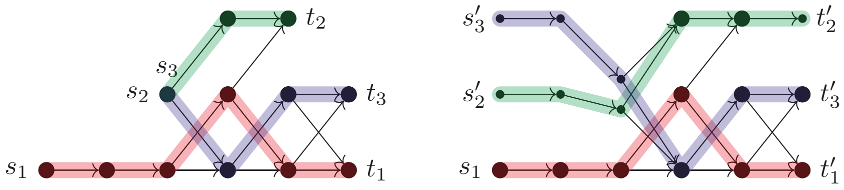


Figure 5.5: *Left-hand side:* An example of 3-Disjoint Paths on c-layered DAGs. The c-coordinate of vertices is illustrated by their horizontal position. The terminal pairs are (s_1, t_1), (s_2, t_2), and (s_3, t_3) and $s_2 = s_3$. A solution is highlighted. *Right-hand side:* An equivalent instance in which $s_i =^c s_j$ for all $i, j \in [p]$. The smaller vertices are the vertices that are newly introduced by the construction. The corresponding solution is again highlighted. Note that since $s_2 = s_3$, after adding s'_2 and s'_3, the solution is not internally vertex-disjoint if s_2 was not duplicated.

in this topological order. We assume without loss of generality that $s_i \preceq s_j$ for all $i < j \in [p]$. We further assume that $s_i \preceq t_i$ for all $i \in [p]$ as otherwise there can be no path from s_i to t_i and that $p \leq n$ as we can iterate over all pairs (s_i, t_i) and delete those that are connected by an arc $(s_i, t_i) \in A$. All remaining paths have at least one inner vertex that has to be from $V \setminus V^{\text{end}}$ and that has to be unique for each path. Hence, if there are at least $n + 1$ pairs remaining, then the instance has no solution.

We build a table $T[x_1, x_2, \dots, x_p] \in \{\text{true}, \text{false}\}$ that stores true if and only if the following three criteria are fulfilled.

i) $x_i \in V \setminus (V^{\text{end}} \setminus \{s_i, t_i\})$,

ii) $s_i \preceq x_i \preceq t_i$ for all $x_i \in \{s_i, t_i\}$, and

iii) there exist s_i-x_i-paths such that each inner vertex of each of these paths is in $V \setminus V^{\text{end}}$ and that each vertex in $V \setminus V^{\text{end}}$ is contained in at most one of these paths.

If the table is completely filled, then there is a set of internally vertex-disjoint shortest s_j-t_j-paths if and only if $T[t_1, t_2, \dots, t_p] = \text{true}$ as the first two requirements are trivially fulfilled. We initialize the table with $T[s_1, s_2, \dots, s_p] := \text{true}$ as internally vertex-disjoint s_i-s_i-paths trivially exist. Moreover, for each tuple $(x_1, \dots, x_p) \in V^p$ if $x_i \prec s_i$ or $t_i \prec x_i$ or $x_i \in V^{\text{end}} \setminus \{s_i, t_i\}$ for at least one $i \in [p]$, then we set $T[x_1, \dots, x_p] := \text{false}$. Note that there are n^p possible tuples and initializing each entry takes $O(n)$ time.

We next show how to compute the entries of T. To this end, for some tuple $(x_1, x_2, \ldots, x_p)$, let x_ℓ be a vertex such that $x_\ell \neq s_\ell$ and $x_i \preceq x_\ell$ for all x_i with $i \in [p]$ and $x_i \neq s_i$. Moreover, let

$$N^*(x_i) := \{v \mid (v, x_i) \in A \wedge v \in (V \setminus (V^{\text{end}} \setminus \{s_i\})) \setminus \{x_1, \ldots, x_p\}\}.$$

Finally, let

$$T[x_1, x_2, \ldots, x_p] := \bigvee_{x'_\ell \in N^*(x_\ell)} T[x_1, x_2, \ldots, x_{\ell-1}, x'_\ell, x_{\ell+1}, \ldots, x_p].$$

We now show by induction on the sum of positions in the topological order of all x_i that $T[x_1, x_2, \ldots, x_p] = \text{true}$ if and only if the three criteria are fulfilled. In the base case, $T[s_1, s_2, \ldots, s_p] = \text{true}$, or there is some x_i such that $x_i \prec s_i$ and therefore $T[x_1, x_2, \ldots, x_p] = \text{false}$. Note that there is no s_i-x_i-path in the latter case.

Now to show the statement for some table entry $T[x_1, x_2, \ldots, x_p]$, assume that the statement holds for all table entries $T[x'_1, x'_2, \ldots, x'_p]$ such that $x'_i \preceq x_i$ for all $i \in [p]$ and $x'_j \prec x_j$ for at least one $j \in [p]$. To this end, first assume that $T[x_1, x_2, \ldots, x_p] = \text{true}$. Since $T[x_1, x_2, \ldots, x_p] = \text{true}$, it was not set to false in the initialization and thus i) and ii) are satisfied. By construction, there is an $x'_\ell \in N^*(x_\ell)$ such that $T[x_1, x_2, \ldots, x_{\ell-1}, x'_\ell, x_{\ell+1}, \ldots, x_p] = \text{true}$. By induction hypothesis, there are internally vertex-disjoint s_ℓ-x'_ℓ- and s_j-x_j-paths for all $j \in [p] \setminus \{\ell\}$ such that $s_\ell \preceq x'_\ell \preceq t_\ell$ and $x'_\ell \in V \setminus (V^{\text{end}} \setminus \{s_\ell, t_\ell\})$. Since by definition of x_ℓ it holds that $x_i \prec x_\ell$ for all x_i with $i \in [p]$ and $x_i \neq s_i$, it holds that x_ℓ is not contained in any of the s_i-x_i-paths for $i \in [p]$. Hence the s_ℓ-x'_ℓ-path can be extended by the edge (x'_ℓ, x_ℓ) and the resulting path combined with the other s_i-x_i-paths satisfies iii).

To show the other direction assume that $x_1, x_2, \ldots, x_p$ satisfy i) to iii). Then consider the s_ℓ-x_ℓ-path and the predecessor x'_ℓ of x_ℓ. Note that x'_ℓ exists as otherwise $x_i = s_i$ for all $i \in [p]$ and hence we are in the base case. By construction, $x'_\ell \in N^*(x_\ell) \subseteq V \setminus (V^{\text{end}} \setminus \{s_i\})$. Note further that $x'_\ell \prec x_\ell \preceq t_\ell$, implying $x'_\ell \neq t_\ell$ and hence i) is also satisfied by x'_ℓ. Further, since there is an s_ℓ-x'_ℓ-path (a subpath of the s_ℓ-x_ℓ path), it holds that $s \preceq x'_\ell \prec x_\ell \preceq t_\ell$ and thus x'_ℓ also satisfies ii). Finally, iii) is also satisfied by the s_ℓ-x'_ℓ-subpath combined with the other s_i-x_i-paths. The induction hypothesis then states that $T[x_1, x_2, \ldots, x_{\ell-1}, x'_\ell, x_{\ell+1}, \ldots, x_p] = \text{true}$. Since $x'_\ell \in N^*(x_\ell)$, it holds that $T[x_1, x_2, \ldots, x_p] = \text{true}$. Thus, the statement holds for all table entries $T[x_1, x_2, \ldots, x_p]$.

It remains to analyze the running time. There are at most n^p possible table entries and computing one takes $O(n)$ time as V^{end}, ℓ, and $N^*(x_\ell)$ can be computed in $O(p+n) \subseteq O(n)$ time and iterating over at all neighbors of x_ℓ takes $O(n)$ time. Hence the overall running time is $O(n^{p+1})$. □

After showing how to solve the subproblems, it remains to show that Algorithm 5.1 is correct and to analyze its running time. We start with the analysis of the running time.

Lemma 5.27. *Algorithm 5.1 runs in $O(k \cdot n^{16k \cdot k! + k + 1})$ time.*

Proof. First, observe that there are at most $k \cdot k!$ different permutations of subsets of k objects as there are exactly $k!$ permutations of exactly k objects and each of these can be truncated at k positions to get any permutation of any smaller (non-empty) subset of objects. Second, observe that by Definition 5.7 there are at most eight vertices guessed for each sequence σ as if $\delta_P^{i,j}(Q) \neq \bot$, then $\alpha_P^{i,j}(Q) = \omega_P^{i,j}(Q) = \partial_P^{i,j}(Q) = \varpi_P^{i,j}(Q) = \bot$. Hence, at most $8k \cdot k!$ vertices need to be guessed, which requires at most $n^{8k \cdot k!}$ attempts.

Next we analyze the running time of each iteration of the main foreach-loop in Algorithm 5.1. Notice that by Definition 5.7, for each sequence σ there are at most four vertices on a marble path T_i and that each of these vertices increases the number of minimal segments S on T_i by at most one. Note that for each σ the set $\mathcal{C}^\sigma$ contains vertices from at most two paths. Thus, we create at most $8k \cdot k!$ new segments overall. Since we start with k marble paths, there are at most $8k \cdot k! + k$ minimal segments. Thus, there are at most $(8k \cdot k! + k) \cdot (k \cdot k!)$ iterations of the loop in Line 7, each of which takes constant time. Each iteration of Line 14 can be done in $O(n)$ time using bucket sort and hence the overall running time for all iterations is in $O(n \cdot k)$.

Next, there are k instances of p_i-Disjoint Paths on i-layered DAGs that are solved using Lemma 5.26, where $p_i \leq 8k \cdot k! + k$ for all $i \in [k]$. By Lemma 5.26 the running time for solving one instance is $O(n^{8k \cdot k! + k + 1})$ and the running time for solving all instances is hence $O(k \cdot n^{8k \cdot k! + k + 1})$. Lastly, we verify in Algorithm 5.1 that the solutions found can indeed be merged into one solution for k-Disjoint Shortest Paths. Note that we only stated the decision version of p-Disjoint Paths on c-layered DAGs but the actual solution can be found using a very similar algorithm where we do not only store true or false in the table T but also some set of disjoint paths corresponding to each table entry that stores true. Verifying a solution can for example be done in $O(k \cdot n)$ time by iterating over all solution paths and verify that between each pair of consecutive vertices there is an edge, that all paths are shortest paths, and that all paths are internally vertex-disjoint. This can be done by marking all inner vertices of each path and if some vertex is already marked once and visited again, then return false and otherwise return true. Thus the overall running time of Algorithm 5.1 is

$$O(n^{8k\cdot k!}\cdot((8k\cdot k!+k)\cdot(k\cdot k!)+n\cdot k+k\cdot n^{8k\cdot k!+k+1}+n\cdot k) \subseteq O(k\cdot n^{16k\cdot k!+k+1}). \quad \square$$

For the correctness of Algorithm 5.1, we need to show that each part of the partition of minimal segments can be solved independently. This follows from Proposition 5.24 together with the fact that Algorithm 5.1 exhaustively tries all possibilities for the crosssing set $\mathcal{C}$. Together with Lemma 5.27, this implies our main theorem.

Theorem 5.28. *k-Disjoint Shortest Paths is solvable in $O(k \cdot n^{16k\cdot k!+k+1})$ time.*

Proof. We use Algorithm 5.1 and focus on the correctness as the running time is already analyzed in Lemma 5.27. If Algorithm 5.1 returns true, then Line 16 is executed and a solution is verified. It remains to show that if there is some solution, then Algorithm 5.1 returns true. If there is some solution $\mathcal{P} = (P_i)_{i\in[k]}$, then let $\mathcal{C}$ be its crossing set (Definition 5.7). Then, there is some iteration of Line 2 where all guesses are correct, that is, Ends $= \mathcal{T}$ and $T_i = V(P_i) \cap \mathcal{C}$. We now consider this iteration of Line 2.

Observation 5.17 states that for each sequence σ and for each segment S with $\{\text{start}(S), \text{end}(S)\} = \text{Ends}(\sigma) = \mathcal{T}(\sigma)$ the pair $\{\text{start}(S), \text{end}(S)\}$ is c-colored for each $c \in \text{set}(\sigma)$. Hence the same also holds for each minimal segment $S' \subseteq S$. By Line 7, there is a solution where the shortest paths between the endpoints of each minimal segment S are strictly c-monotone for each $c \in \text{labels}[S]$. Note that $\text{labels}[S] = \text{marks}[S]$ in this iteration of Line 2. Hence each path following S is strictly c-increasing for each $c \in \text{marks}[S]$ and by Observation 5.25 this shortest path is contained in D_c. Hence we can find *some* solution for each minimal segment using Lemma 5.26 such that all paths for these minimal segments with the same marks are internally vertex-disjoint. Since $\text{marks}[S] = \text{labels}[S]$ for all minimal segments, by Proposition 5.24, all shortest paths between endpoints of minimal segments with different marks are internally vertex-disjoint. Hence, the result computed by Algorithm 5.1 is a solution to k-Disjoint Shortest Paths and thus the algorithm returns true. $\square$

5.4 Concluding Remarks

We provided an improved polynomial-time algorithm for k-Disjoint Shortest Paths for constant k. However, while the running time of Algorithm 5.1 can certainly be further improved by some case distinctions and a further refined analysis, the algorithm is still far from being practical. We believe that Algorithm 5.1 can be improved to run in $n^{2^{O(k)}}$ time. It is left open whether a running time of $n^{k^{O(1)}}$ is possible.

Concerning generalizations of k-Disjoint Shortest Paths, we believe that Algorithm 5.1 can be modified to not only work for unit edge lengths but also for positive integer lengths. However, the case of non-negative edge lengths seems much more difficult as edges with length zero result in overlapping vertices in our geometric representation. Finally, if there are no k disjoint shortest paths for some constant k, then computing in polynomial time disjoint paths with minimum length is still an open problem (for $k = 2$ Björklund and Husfeldt [BH19] provided an $O(n^{11})$-time randomized algorithm).

Part II

2-SAT Programming

Chapter 6

Tree Containment

In this chapter, we investigate a problem from computational biology. Concerning 2-SAT programming, we present a general k-SAT program that shows that a relevant special case is polynomial-time solvable as the resulting program only contains 2-SAT formulas. Concerning problem-specific aspects, we introduce a new variant of a well-known problem in computational biology. The new version models a certain uncertainty regarding the history of evolution. We then identify a relevant special case of this new variant and a natural parameter that models the amount of uncertainty. We conclude with an equivalence between the identified special case and k-SAT in the sense that there are reductions from and to k-SAT, where the value of k in both reductions matches the value of our identified parameter. This proves that the special case is polynomial-time solvable for $k \leq 2$ and *NP*-hard for $k \geq 3$.

With the dawn of molecular biology also came the realization that evolutionary trees, which have been widely adopted by biologists, are insufficient to describe certain processes that have been observed in nature. In the last decade, the idea of reticulate evolution, supporting gene flow from multiple parent species, arose [CCR13, TR11]. Reticulate evolution is described using "phylogenetic networks" (see the monographs by Gusfield [Gus14] and Huson et al. [HRS10] or the formal definitions in Section 6.1). A central question when dealing with phylogenetic networks is whether or not different phylogenetic networks provide consistent information. The corresponding problem is known as Tree Containment and it has been shown to be *NP*-hard [ISS10, Kan+08].

In real life, we cannot hope for perfectly precise evolutionary history. In particular, speciation events (a species splitting off another) occurring in rapid succession (only a few thousand years between speciation events) can often not be reliably placed in the order as they occurred. Incomplete information about a certain set of successive speciation events is called a soft polytomy and it is modeled by a non-binary vertex (a vertex with more than two parent species) in a phylogenetic network. We consider the information provided by two non-binary phylogenetic networks consistent if we can replace each non-binary vertex by some binary tree such that the resulting binary phylogenetic networks provide consistent information.

In Section 6.2, we present first structural results for Tree Containment with soft polytomies. In Section 6.3, we show that if one input network is a single-labeled phylogenetic tree and the other input network is a multi-labeled tree (for a definition, see Section 6.1), then Tree Containment is polynomial-time solvable if each label occurs at most twice in the multi-labeled phylogenetic tree and *NP*-complete otherwise. The polynomial-time algorithm is based on the results from Section 6.2 and 2-SAT programming.

6.1 Problem Definition and Related Work

A *phylogenetic network* on a set X of taxa is a rooted, single-source, directed, and acyclic graph in which all vertices have in-degree at most one or out-degree exactly one and each leaf v (a vertex with out-degree zero) is labeled with one taxon $x \in X$. We also say that v has label x. By default, no label occurs twice in a phylogenetic network, and we will make exceptions explicit by calling phylogenetic networks *multi-labeled* if a label can occur more than once. We say that it is *ℓ-labeled* if each label occurs at most ℓ times and if we want to emphasize that a phylogenetic network is not multi-labeled, then we call it *single-labeled*. Vertices with in-degree at least two (and out-degree one) are called *reticulations* and the other vertices are called *tree vertices*. A phylogenetic network without reticulations is called a *phylogenetic tree* and a phylogenetic network or tree is called binary if each vertex has in-degree and out-degree at most two. Figure 6.1 shows an example of a binary phylogenetic network (left-hand side) and a phylogenetic tree (right-hand side).

An important task in computational biology is to check whether two models of evolution are consistent. A relevant special case therein is whether a given phylogenetic network is consistent with an existing tree model or not [Gam+15]. A phylogenetic network N and a phylogenetic tree T are considered consistent if N *displays* T. For the definition of displaying, recall that subdividing an arc (u, v) in a directed graph refers to removing the arc (u, v) and replacing it

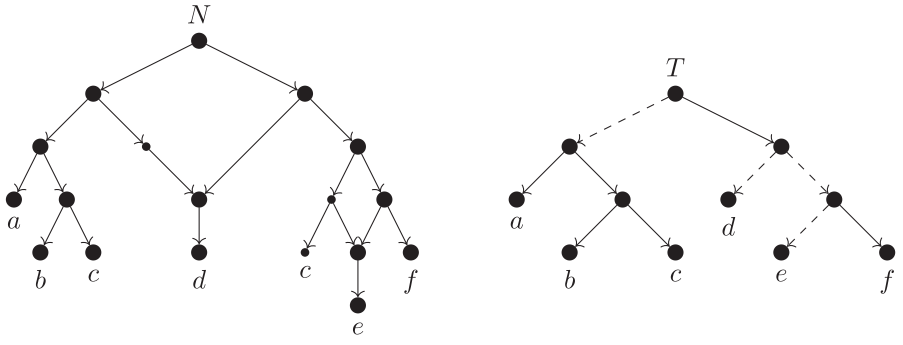


Figure 6.1: A 2-labeled phylogenetic network N (left-hand side) and a phylogenetic tree T (right-hand side). The respective topmost vertex is the only source and is called the root. The leaves are each labeled with one element of the set $\{a, b, c, d, e, f\}$. The parents of the leaves d and e in the left example are the reticulations in N and all other vertices are tree vertices. Removing the three smaller vertices (and all incident arcs) in N on the left-hand side and subdividing each dashed arc in T on the right-hand side once yields isomorphic[1] trees. Hence, N displays T.

by a new vertex w and two new arcs (u, w) and (w, v). A *subdivision* of a graph is the result of repeatedly subdividing arcs in it.

Definition 6.1. Let N be a (possibly multi-labeled) phylogenetic network and let T be a single-labeled phylogenetic tree. Then, N *firmly displays* T if a subdivision of N contains a subdivision of T as a subgraph such that leaf-labels are respected, that is, each leaf v in T with label x is mapped to a leaf with label x in N.

An example for Definition 6.1 is depicted in Figure 6.1. Based on this definition, Tree Containment is defined as follows.

Tree Containment
Input: A (possibly multi-labeled) phylogenetic network N and a single-labeled phylogenetic tree T.
Question: Does N firmly display T?

[1]In this chapter, *isomorphic* always refers to an isomorphism respecting leaf-labels, that is, the isomorphism must map a leaf with some label λ in N to a leaf with label λ in T.

Kanj et al. [Kan+08] showed that Tree Containment is *NP*-hard. Due to its importance in the analysis of evolutionary history, there have been several attempts to identify polynomial-time computable special cases [BS16, FKP15, Gam+15, GDZ17, Gun18, ISS10, Kan+08, Wel18] as well as moderately exponential-time algorithms [GLZ16, Wel18]. Since the definitions for the special cases are rather technical and the results are not relevant for this thesis, we do not present definitions here but only refer the reader to the works by Fakcharoenphol et al. [FKP15] and Weller [Wel18] for an overview.

Motivated by the concept of *soft polytomies*, that is, incomplete knowledge about the order of a limited set of speciation events, we consider a notion we call *soft displaying.* The goal is to allow any high-degree vertex to be replaced by any binary tree such that the resulting phylogenetic network firmly displays the resulting phylogenetic tree. To this end, we consider *arc contractions*. Contracting an arc (u, v) in a directed graph refers to the process of "merging" u and v (and all incident arcs). Formally, vertices u and v are removed and replaced by a new vertex w. For each vertex x other than u or v, if (x, u) or (x, v) existed in the original graph, then the new graph contains an arc (x, w) and if (u, x) or (v, x) existed in the original graph, then the new graph contains an arc (w, x). A *contraction* of a phylogenetic network is the result of repeatedly performing arc contractions in it. We call a binary phylogenetic network $B = (V_B, A_B)$ a *binary resolution* of a phylogenetic network $N = (V_N, A_N)$ if N is a contraction of B. An example of contractions and binary resolutions is given in Figure 6.2. We call a surjective function $\chi\colon V_B \to V_N$ a *contraction function* of B for N if contracting all arcs (uv) in B with $\chi(u) = \chi(v)$ results in a graph isomorphic to N. The notion of binary resolutions leads to the following definition of soft displaying.

Definition 6.2. Let N be a (possibly multi-labeled) phylogenetic network and let T be a single-labeled phylogenetic tree. Then, N *softly displays* T if there are binary resolutions N_B of N and T_B of T such that N_B firmly displays T_B.

Note that, since each binary resolution of a binary phylogenetic network N is a subdivision of N, it holds that the concepts of firm and soft displaying coincide for binary phylogenetic networks. The notion of soft displaying naturally leads to the following definition of Soft Tree Containment.

Soft Tree Containment
Input: A (possibly multi-labeled) phylogenetic network N and a single-labeled phylogenetic tree T.
Question: Does N softly display T?

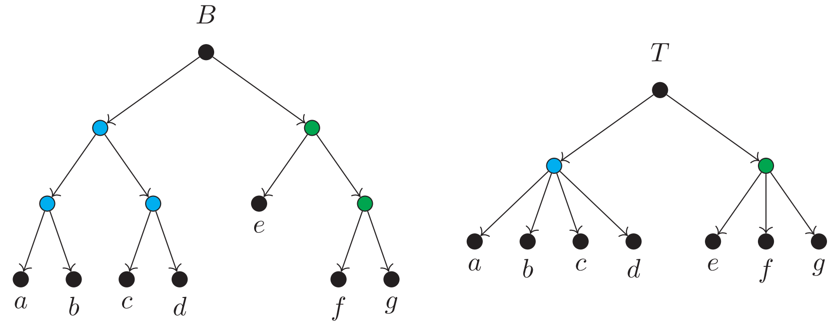


Figure 6.2: Two phylogenetic trees B (left-hand side) and T (right-hand side). The phylogenetic tree B is binary. Contracting the arc between the two green vertices in B yields the green vertex in T. Analogously, exhaustively contracting any arc between two blue vertices in B yields the blue vertex in T. Since the result of contracting these arcs in B is isomorphic to T, the phylogenetic tree B is a binary resolution of T.

An example of Soft Tree Containment is given in Figure 6.3. Throughout this chapter, we will mostly focus on Soft Tree Containment and for the sake of readability, we refer to soft displaying simply as "displaying". To the best of our knowledge, we are the first to study Soft Tree Containment. In this thesis, we focus on the special case where N is a multi-labeled phylogenetic tree. This has three main reasons. First, Tree Containment is known to be *NP*-hard even on binary phylogenetic networks and since Tree Containment and Soft Tree Containment coincide for binary phylogenetic networks, Soft Tree Containment is *NP*-hard on binary phylogenetic networks (that is, N is not restricted to being a phylogenetic tree). Conversely, Tree Containment is polynomial-time solvable when N is a phylogenetic tree [Gam+15] and hence, the computational complexity of Soft Tree Containment on phylogenetic trees remains unclear. Second, reticulation events are comparatively rare especially when considering phylogenies of animals and so chances are that the input consists of phylogenetic trees (or phylogenetic networks with few reticulations). Hence, Soft Tree Containment on phylogenetic trees is a relevant special case from a biological perspective. Third, each algorithm for Soft Tree Containment on phylogenetic networks has to decide on a subgraph of N that is a phylogenetic tree and then verify that this phylogenetic tree softly

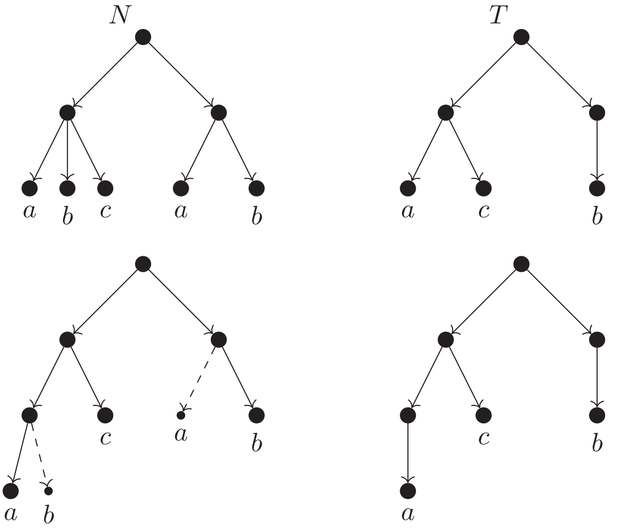


Figure 6.3: An example for Soft Tree Containment. In the top left-hand corner is a multi-labeled tree N and in the top right-hand corner is a single-labeled tree T. In the bottom right-hand corner is a subdivision of (a binary resolution of) T and in the bottom left-hand corner is (a subdivision of) a binary resolution of N. The subgraph in the bottom left-hand corner consisting of all vertices except for the two small vertices and all but the two dashed arcs is isomorphic to the phylogenetic tree in the bottom-right hand corner. This shows that N softly displays T.

displays T. Thus, Soft Tree Containment on phylogenetic trees is a relevant special case from an algorithmic perspective.

We conclude this section with some notation for the remainder of this chapter. In a single-labeled phylogenetic network, we use leaves and labels (taxa) interchangeably. A binary phylogenetic network B on three leaves a, b, and c is called a *triplet* and we denote it by $ab|c$ if c is a child of the root of B. In Figure 6.1, the subtree rooted in the parent of the leaf labeled with a is the triplet $bc|a$. We denote by N_v the subnetwork (or subtree) of N rooted in v, that is, the induced subgraph containing v and all its descendants. We denote the set of labels in a subnetwork N_v by $\mathcal{L}(N_v)$. Slightly abusing notation, we use n as the maximum number of vertices in N and T.

Recall that we use the notation $v <_D u$ to denote that a vertex v is a descendant of a vertex u in a directed acyclic graph (DAG) D. We use $v \leq_D u$ to denote that v is a descendant of u in D or $v = u$. Moreover, recall that the least common ancestor(s) (LCA) of a set V' of vertices is a set L of vertices such that each vertex in L is an ancestor of each vertex in V' and no descendant of a vertex in L is an ancestor of each vertex in V'. In trees, the LCA of any set of vertices is always a set containing a single vertex and for the sake of readability, we will assume that the LCA in a tree *is* a single vertex.

Let $N = (V, A)$ be a phylogenetic network. Recall that suppressing a vertex v with one incoming arc (u, v) and one outgoing arc (v, w) refers to the procedure of removing v and both incident arcs and adding the arc (u, w) to the graph (if it does not already exist). For any subset $U \subseteq V$ of vertices, we denote the result of removing all vertices v that do not have a descendant in U by $N|_U$, and $N||_U$ is the result of suppressing all degree-two vertices in $N|_U$. Such a phylogenetic network $N||_U$ can be computed in $O(|U|)$ time [Col+00]. Moreover, if N is a phylogenetic tree, then $N|_L$ is the smallest subtree of N containing the vertices in L and the root of N.

If N contains a subgraph S that is isomorphic to a tree T up to subdivision of arcs, then we simply say that N contains a subdivision of T. Slightly abusing notation, if an isomorphism maps a vertex v in T to a vertex u in S (and thus in N), then we do not distinguish between u and v but say that both vertices are the same. Thus, S consists of all vertices in T and some vertices of in- and out-degree one.

6.2 Single-labeled Trees

In this section, we will develop a characterization of when a single-labeled phylogenetic tree softly displays another single-labeled phylogenetic tree. To this end, all phylogenetic networks are single-labeled in this section. The characterization will then be used in Section 6.3 to design an algorithm for Soft Tree Containment when the input network N is a multi-labeled phylogenetic tree.

We start with a series of basic observations regarding the concept of displaying. First, note that a binary phylogenetic tree displays another binary phylogenetic tree if and only if they are isomorphic up to subdivision of arcs. Hence, if a phylogenetic tree T displays another phylogenetic tree T' on the same set of taxa, then there exist binary resolutions B of T and B' of T such that B displays B', that is, B and B' are isomorphic up to subdivision of arcs. Since isomorphism is a symmetric relation, T' then also displays T.

Observation 6.1. *A phylogenetic tree T displays a phylogenetic tree T' on the same label-set if and only if T' displays T.*

For binary trees and, in particular, triplets, the concept of firm displaying is well-researched and we will use the following characterization to develop a characterization for when a phylogenetic tree softly displays another phylogenetic tree.

Lemma 6.2 ([Dre+12, Chapter 9.1])**.** *Let B be a binary phylogenetic tree. Let $a, b, c \in \mathcal{L}(B)$ be three distinct labels. Then, B firmly displays the triplet $ab|c$ if and only if*

$$\mathrm{LCA}(\{a, b\}) <_B \mathrm{LCA}(\{b, c\}) = \mathrm{LCA}(\{a, c\}).$$

Indeed, B is uniquely identified (up to subdivision and suppression of degree-two vertices) by the set D of displayed triplets, that is, B is the only binary tree displaying the triplets in D.

Based on Lemma 6.2, we can now relate the two forms of displaying for triplets in non-binary trees. To this end, recall that in trees the LCA of a set of vertices is uniquely determined. Moreover, it is easy to verify that if it holds for three leaves a, b, and c in a tree T that $\mathrm{LCA}_T(\{a, b\}) <_T \mathrm{LCA}_T(\{a, c\})$, then $\mathrm{LCA}_T(\{a, c\}) = \mathrm{LCA}_T(\{b, c\})$. Lemma 6.2 and the definition of soft displaying then immediately imply the following.

Observation 6.3. *Let T be a tree and let $a, b, c \in \mathcal{L}(T)$. Then,*

(a) T firmly displays $ab|c$ if and only if

$$\mathrm{LCA}(\{a, b\}) <_T \mathrm{LCA}(\{a, c\}) = \mathrm{LCA}(\{b, c\}).$$

(b) T firmly displays $ac|b$ or $bc|a$ if and only if T does not softly display $ab|c$.

The next observation states that, in trees, an arc contraction does not change the ancestor relation. This is important as it allows us to reason about LCAs in binary resolutions.

Observation 6.4. *Let T be a tree and let T' be the result of contracting any arc in T. Let Y and Z be two sets of leaves common to T and T'. Then,* *(a)* $\mathrm{LCA}_T(Y) \leq_T \mathrm{LCA}_T(Z)$ *if and only if* $\mathrm{LCA}_{T'}(Y) \leq_{T'} \mathrm{LCA}_{T'}(Z)$ *and*

(b) if $\mathrm{LCA}_{T'}(Y) <_{T'} \mathrm{LCA}_{T'}(Z)$*, then* $\mathrm{LCA}_T(Y) <_T \mathrm{LCA}_T(Z)$*.*

Recall the example in Figure 6.2 and therein consider the contraction of the arc between the two green vertices in B. Observation 6.4 then states for $Y := \{f, g\}$ and $Z := \{a, f, g\}$ that $\mathrm{LCA}_B(\{f, g\}) \leq_B \mathrm{LCA}_B(\{a, f, g\})$ if and only if $\mathrm{LCA}_T(f, g) \leq_T \mathrm{LCA}_T(\{a, f, g\})$. Note that this is indeed the case as the LCA of $\{a, f, g\}$ is in both phylogenetic trees the root and the LCA of $\{f, g\}$ is the respective (lower) green vertex.

We now give a characterization of when a phylogenetic tree softly displays another phylogenetic tree. It is based on Lemma 6.2 and the following observation. Note that if in a tree B it holds that $\mathrm{LCA}(\{a, b\}) <_B \mathrm{LCA}(\{b, c\}) = \mathrm{LCA}(\{a, c\})$, then there is no vertex v such that $a, c \in \mathcal{L}(v)$ and $b \notin \mathcal{L}(v)$ as any ancestor of $\mathrm{LCA}(\{a, c\})$ is an ancestor of $\mathrm{LCA}(\{a, b\}) <_B \mathrm{LCA}(\{a, c\})$.

Lemma 6.5. *Let $N = (V_N, A_N)$ and $T = (V_T, A_T)$ be two phylogenetic trees on the same leaf-label set. Then, N softly displays T if and only if, for all $u \in V_T$ and $v \in V_N$, it holds that $\mathcal{L}(T_u) \subseteq \mathcal{L}(N_v)$, $\mathcal{L}(T_u) \supseteq \mathcal{L}(N_v)$, or $\mathcal{L}(T_u) \cap \mathcal{L}(N_v) = \emptyset$.*

Proof. We start by showing that if N displays T, then for all $u \in V_T$ and $v \in V_N$, it holds that $\mathcal{L}(T_u) \subseteq \mathcal{L}(N_v)$, $\mathcal{L}(T_u) \supseteq \mathcal{L}(N_v)$, or $\mathcal{L}(T_u) \cap \mathcal{L}(N_v) = \emptyset$. Assume towards a contradiction that N softly displays T but there are $u \in V_N$ and $v \in V_T$ such that

$$\mathcal{L}(N_u) \not\subseteq \mathcal{L}(T_v),\ \mathcal{L}(N_u) \not\supseteq \mathcal{L}(T_v),\ \text{and } \mathcal{L}(N_u) \cap \mathcal{L}(T_v) \neq \emptyset.$$

This is equivalent to the statement that there are three taxa x, y, and z such that

$$x \in \mathcal{L}(N_u) \setminus \mathcal{L}(T_v),\ y \in \mathcal{L}(N_u) \cap \mathcal{L}(T_v),\ \text{and } z \in \mathcal{L}(T_v) \setminus \mathcal{L}(N_u).$$

Since each label appears only once in N and T and N softly displays T, it holds that there are binary resolutions N^B of N and T^B of T such that N^B and T^B are isomorphic up to subdivision of arcs. Hence, there is a vertex u' in N^B with $\mathcal{L}(N^B_{u'}) = \mathcal{L}(N_u)$ and a vertex v' in T^B with $\mathcal{L}(T^B_{v'}) = \mathcal{L}(T_v)$. Since $x, y \in \mathcal{L}(N_u) = \mathcal{L}(N^B_{u'})$ and $z \notin \mathcal{L}(N_u) = \mathcal{L}(N^B_{u'})$, it holds that

$$\mathrm{LCA}(\{x, y\}) \leq_{N^B} u' <_{N^B} \mathrm{LCA}(\{y, z\}) = \mathrm{LCA}(\{x, z\}),$$

that is, N^B displays $xy|z$. Analogously, T^B displays $yz|x$. By Lemma 6.2, this contradicts the fact that T^B and N^B are isomorphic up to subdivision of arcs.

We continue with the other direction, that is, we show that if for all vertices $u \in V_T$ and $v \in V_N$ it holds that

$$\mathcal{L}(T_u) \subseteq \mathcal{L}(N_v),\ \mathcal{L}(T_u) \supseteq \mathcal{L}(N_v),\ \text{or } \mathcal{L}(T_u) \cap \mathcal{L}(N_v) = \emptyset,$$

then N displays T. Using Lemma 6.2, we will show how to construct binary trees B_N and B_T such that B_N is a binary resolution of N, B_T is a binary resolution of T, and both display all triplets that are firmly displayed by N or T. Since the constructions for both trees are analogous, we only focus on B_N here. Consider any vertex $v \in V_N$ that has out-degree at least three. Then, there are three labels a, b, and c such that $\mathrm{LCA}_N(\{a,b\}) = \mathrm{LCA}_N(\{a,c\}) = \mathrm{LCA}_N(\{b,c\})$. Let c_a, c_b, c_c be the three children of v in N such that $a \in \mathcal{L}(N_{c_a})$, $b \in \mathcal{L}(N_{c_b})$, and $c \in \mathcal{L}(N_{c_c})$. We now consider the two cases whether or not

$$\mathrm{LCA}_T(\{a,c\}) = \mathrm{LCA}_T(\{a,b\}) = \mathrm{LCA}_T(\{b,c\}).$$

If $\mathrm{LCA}_T(\{a,c\}) = \mathrm{LCA}_T(\{a,b\}) = \mathrm{LCA}_T(\{b,c\})$, then neither N nor T displays one of the triplets $ab|c$, $ac|b$, or $bc|a$. Hence we arbitrarily replace the arcs (v,c_b) and (v,c_c) by a new vertex w and new arcs (v,w), (w,c_b) and (w,c_c). Note that the resulting phylogenetic tree firmly displays all triplets that N firmly displayed and the triplet $bc|a$. Since this procedure reduces the out-degree of one vertex of out-degree at least three and does not introduce new vertices of out-degree at least three, we can repeat this procedure until no vertex has out-degree at least three any more, that is, the resulting phylogenetic tree is binary. Observe further that B_N is trivially a binary resolution of B_N and therefore N softly displays B_N by definition. The construction of B_T is analogous and whenever

$$\begin{aligned}\mathrm{LCA}_N(\{a,c\}) = \mathrm{LCA}_N(\{a,b\}) = \mathrm{LCA}_N(\{b,c\}) \text{ and}\\ \mathrm{LCA}_T(\{a,c\}) = \mathrm{LCA}_T(\{a,b\}) = \mathrm{LCA}_T(\{b,c\}),\end{aligned}$$

then we construct B_T to display the same triplet as B_N.

Note that since B_N and B_T are binary, they firmly display one of the following three possible triplets $ab|c$, $ac|b$, or $bc|a$ for each triple (a,b,c) of labels. By Lemma 6.2, B_N and B_T are isomorphic up to subdivision of arcs as binary trees are uniquely defined by their displayed triplets. Hence B_N is a subdivision of a binary resolution of both N and T and, as B_N is binary, N softly displays T by definition. $\square$

We conclude this section with a helpful lemma that lists some equivalent characterizations of soft displaying in relevant special cases. This lemma will be used to show hardness of Soft Tree Containment in Subsection 6.3.2.

Lemma 6.6. *Let T and T' be phylogenetic trees and let B be a binary phylogenetic tree, all on the same set X of labels.*

(a) T softly displays the leaf-triplet $ab|c$ if and only if

$$\mathrm{LCA}(\{a,b\}) \leq \mathrm{LCA}(\{b,c\}) = \mathrm{LCA}(\{a,c\}).$$

(b) T softly displays B if and only if T softly displays all triplets that B displays firmly.

(c) T softly displays a tree T' (and vice versa) if and only if there is a binary tree B on X that is softly displayed by both T and T'.

Proof. We prove the three statements one after another. To verify statement (a), note that, by definition, T softly displays $ab|c$ if and only if there is a binary resolution T_B of T displaying $ab|c$. By Lemma 6.2, T_B firmly displays $ab|c$ if and only if

$$\mathrm{LCA}_{T_B}(\{a,b\}) <_{T_B} \mathrm{LCA}_{T_B}(\{a,c\}) = \mathrm{LCA}_{T_B}(\{b,c\}).$$

Since T_B is binary, this is equivalent to

$$\mathrm{LCA}_{T_B}(\{a,b\}) \leq_{T_B} \mathrm{LCA}_{T_B}(\{a,c\}) = \mathrm{LCA}_{T_B}(\{b,c\}),$$

which by Observation 6.4 is equivalent to

$$\mathrm{LCA}_{T}(\{a,b\}) \leq_{T} \mathrm{LCA}_{T}(\{a,c\}) = \mathrm{LCA}_{T}(\{b,c\}).$$

We next prove statement (b). To this end, first assume towards a contradiction that T displays B but a triplet $ab|c$ that B displays firmly is not displayed softly by T. Then, $\{\mathrm{LCA}_T(\{a,b\}), \mathrm{LCA}_T(\{a,c\}), \mathrm{LCA}_T(\{b,c\})\}$ has a unique minimum x with respect to $<_T$ and it holds by statement (a) that $x \neq \mathrm{LCA}_T(\{a,b\})$ (as otherwise T displays $ab|c$). Without loss of generality, let $x = \mathrm{LCA}_T(\{a,c\})$. Since T has a binary resolution that is isomorphic to B up to subdivision of arcs, it holds that T is a contraction of a subdivision of B. Hence, Observation 6.4 states that $\mathrm{LCA}_B(\{a,c\}) <_{T_B} \mathrm{LCA}_B(\{a,b,c\})$ and thus B displays $ac|b$.

Note that a binary tree cannot display $ab|c$ and $ac|b$ and thus we reached a contradiction.

Now, assume towards a contradiction that $T = (V_T, A_T)$ does not softly display $B = (V_B, A_B)$ but displays all triplets that are firmly displayed by B. Since T does not display B, there are by Lemma 6.5 vertices $u \in V_T$ and $v \in V_B$ and labels x, y, and z such that $x \in \mathcal{L}(T_u) \setminus \mathcal{L}(B_v)$, $y \in \mathcal{L}(B_v) \setminus \mathcal{L}(T_u)$, and $z \in \mathcal{L}(T_u) \cap \mathcal{L}(B_v)$. Thus,

$$\begin{aligned}&\mathrm{LCA}_T(\{x,z\}) \leq_T u <_T \mathrm{LCA}_T(\{x,y,z\}) \text{ and}\\ &\mathrm{LCA}_B(\{y,z\}) \leq_B v <_B \mathrm{LCA}_B(\{x,y,z\}).\end{aligned}$$

By statement (a), T displays $xz|y$ and B displays $yz|x$. Since T displays all triplets that B displays firmly, T displays $yz|x$. Again by (a), we can conclude that $\mathrm{LCA}_T(\{y,z\}) \leq_T \mathrm{LCA}_T(\{x,z\}) \leq_T u$. Thus, $y \in \mathcal{L}(u)$, a contradiction.

It remains to show statement (c). By definition, T softly displays T' if and only if there are binary resolutions B and B' of T and T', respectively, such that B firmly displays B'. If such phylogenetic trees exist, then they are by Lemma 6.2 isomorphic up to subdivision of arcs. Thus, B is a binary resolution of a subdivision of T' and the statement follows. $\square$

6.3 Multi-labeled Trees and k-SAT

In this section, we study Soft Tree Containment for multi-labeled phylogenetic trees. We will show a strong connection between k-SAT and Soft Tree Containment on k-labeled phylogenetic trees in the sense that there is a polynomial-time reductions from k-SAT to Soft Tree Containment on k-labeled phylogenetic trees and a k-SAT program for Soft Tree Containment on k-labeled phylogenetic trees. This yields the dichotomy result that Soft Tree Containment on k-labeled phylogenetic trees is polynomial-time solvable if $k \leq 2$ and *NP*-hard if $k \geq 3$. We start with a characterization of when a multi-labeled phylogenetic tree softly displays a single-labeled phylogenetic tree T.

Lemma 6.7. *Let M be a multi-labeled phylogenetic tree and let T be a single-labeled phylogenetic tree on the same set X of labels. Then, M softly displays T if and only if M contains (as a subgraph) a single-labeled phylogenetic tree S on X that softly displays T.*

Proof. We will first show that if $M := (V_M, A_M)$ softly displays $T := (V_T, A_T)$, then M contains a single-labeled phylogenetic tree S that softly displays T. Note that, by definition, if M softly displays T, then there are binary resolutions $M_B := (V_B, A_B)$ of M and T_B of T and subdivisions M_B^S of M_B and T_B^S of T_B such that M_B^S contains T_B^S as a subgraph (respecting leaf labels). Let S_B^S be the subgraph of M_B^S that is isomorphic to T_B^S. Let S_B be the phylogenetic tree that is the result of reverting all subdivisions from M_B to M_B^S in S_B^S, that is, suppressing each vertex v that is contained in S_B^S but not in M_B. Note that S_B^S is a subdivision of S_B and S_B is a single-labeled subgraph of M_B. Let $\chi\colon V_B \to V_M$ be the contraction function of M_B for M, that is, the function mapping each vertex u in M_B to the vertex $\chi(u)$ in M that u is contracted to when forming M. Moreover, let S be the result of contracting each arc (u, v) in S_B with $\chi(u) = \chi(v)$. Note that for each vertex v in M_B it holds that all vertices u with $\chi(u) = \chi(v)$ contract to a single vertex in M and hence these vertices form, by definition of contracting functions, a weakly connected component in M_B. Further, since M_B is a tree and S_B is a subtree of M_B, it holds for each vertex v' in S_B that all vertices u' with $\chi(u') = \chi(v')$ form a weakly connected component in S_B. Thus, the phylogenetic tree S contains no two vertices u' and v' with $\chi(u') = \chi(v')$. Since M is the result of contracting each arc (u, v) with $\chi(u) = \chi(v)$ in M_B, and since S is the result of contracting each arc (u, v) with $\chi(u) = \chi(v)$ in S_B and since S_B is a subtree of M_B, it holds that S is a subtree of M. Concluding, S is a single-labeled subtree of M, S has a binary resolution S_B, S_B has a subdivision S_B^S, and S_B^S is by assumption isomorphic to T_B^S. Thus, S softly displays T by definition.

It remains to show that if M contains a single-labeled subtree S which softly displays T, then M softly displays T. If M contains a single-labeled subtree S that softly displays T, then there are by definition binary resolutions S_B and T_B of S and T, respectively, and subdivisions S_B^S of S_B and T_B^S of T_B such that S_B^S and T_B^S are isomorphic. We will show that M softly displays T, that is, there is a binary resolution M_B of M that has a subdivision M_B^S that contains T_B^S as a subgraph. First, to avoid ambiguity, we relabel each leaf that is not contained in S such that the resulting tree M' is a single-labeled tree on a set $X' \supseteq X$ of labels. This allows us to again refer to leaves of M' in terms of labels. Note that only labels for leaves not contained in S are different between M and M' and hence M' also contains S as a subgraph. Let M'_B be any binary resolution of M' that satisfies the following property. If for three labels $a, b, c \in X$ it holds that $\mathrm{LCA}(a, b) <_{S_B} \mathrm{LCA}(a, c) = \mathrm{LCA}(b, c)$, then $\mathrm{LCA}(a, b) <_{M'_B} \mathrm{LCA}(a, c) = \mathrm{LCA}(b, c)$. Note that M'_B contains a subdivision of S_B as a subtree. Hence, M'_B firmly displays T_B. Finally, let M_B be the multi-labeled phylogenetic tree resulting from replacing the labels in M'_B with their original labels from X. Since M and M' only differ in these labels, it holds that M_B is a binary resolution of M. Further, since S does not contain any of the leaves in which M_B and M'_B differ, it holds that M_B contains a subdivision of S_B as a subgraph. Thus, there is a binary resolution M_B of M and M_B contains a subdivision of S_B as a subgraph which firmly displays T, that is, M softly displays T. □

We will use the characterization shown in Lemma 6.7 to prove both sides of the dichotomy result in this chapter. In Subsection 6.3.1, we present a k-SAT program for Soft Tree Containment on k-labeled phylogenetic trees. This implies that Soft Tree Containment is polynomial-time solvable for 2-labeled phylogenetic trees. In Subsection 6.3.2, we complement this result with a reduction from k-SAT to Soft Tree Containment on k-labeled phylogenetic trees. This implies that Soft Tree Containment on k-labeled phylogenetic trees is *NP*-hard for each $k \geq 3$.

6.3.1 Reduction to k-SAT

In this subsection, we present a k-SAT program for Soft Tree Containment on k-labeled phylogenetic trees. The basic idea is a bottom-up approach that computes for each vertex u in the single-labeled phylogenetic tree T a set $M(u)$ of candidates. Each such candidate is a vertex v in the k-labeled phylogenetic tree N such that the subtree N_v of N rooted in v displays T_u and for no descendant w of v it holds that N_w displays T_u. We will later show that there are at most k such candidates for each vertex in T. Afterwards, we will show how to compute the set $M(u)$ for each vertex u in T in a bottom-up manner using k-SAT.

Note that if N displays T, then, by Lemma 6.7, N contains a single-labeled subtree S that displays T. We call S *canonical* for some vertex u in T if $\mathrm{LCA}_S(\mathcal{L}(T_u)) \in M(u)$ and *canonical* for T if it is canonical for all vertices in T. We start by showing that softly displaying is equivalent to having such a canonical subtree.

Lemma 6.8. *A k-labeled tree N softly displays a single-labeled tree T if and only if N has a canonical subtree for T.*

Proof. Let r be the root of T. If $N := (V_N, A_N)$ has a canonical subtree S for $T := (V_T, A_T)$, then, by definition, S contains a vertex v such that S_v displays $T_r = T$. Hence, N contains a single-labeled tree S that displays T and, by Lemma 6.7, this shows that N displays T.

It remains to show that if N displays T, then N contains a canonical subtree S for T. If N displays T, then N contains by Lemma 6.7 a single-labeled subtree S that displays T. Assume towards a contradiction that S is not canonical for T. Let $u \in V_T$ be a vertex for which S is not canonical but S is canonical for all ancestors of u in T. Note that $u \neq r$ as S displays $T = T_r$ by assumption. Let p be the parent of u in T. Since S is canonical for p, there is a vertex $y := \mathrm{LCA}_S(\mathcal{L}(T_p))$ in S such that S_y displays T_p. Let $S'_y := S_y|_{\mathcal{L}(T_p)}$, that is, S'_y is the subtree of S_y containing all leaves in T_p and no other. By Lemma 6.6(c), there is a binary single-labeled phylogenetic tree B on $\mathcal{L}(T_p)$ which is displayed by S'_y and T_p. By Lemma 6.6(b), S'_y displays each triplet which is firmly displayed by B. Let $x := \mathrm{LCA}_S(\mathcal{L}(T_u))$. Since S is not canonical for u, it holds that S_x does not display T_u or there is a descendant z of x such that S_z displays T_u. By definition of x, for no descendant z of x the subtree S_z can display T_u as for each such z there is a label $\ell \in \mathcal{L}(T_u) \setminus \mathcal{L}(S_z)$ and therefore no triplet containing ℓ can be displayed by S_z. Hence, S_x does not display T_u. Recall that there is a binary phylogenetic tree B which is displayed by S'_y and T_p. Let $B' := B|_{\mathcal{L}(T_u)}$ and let $ab|c$ be any triplet that B' displays firmly. Since B' is a subtree of B it holds that B firmly displays $ab|c$. Hence, S'_y and T_p softly display $ab|c$. If T_u does not display $ab|c$, then, by Observation 6.3(b), it firmly displays $ac|b$ or $bc|a$. Since T_u is a subtree of T_p, also T_p firmly displays $ac|b$ or $bc|a$. By Observation 6.3(b), T_p then does not display $ab|c$, a contradiction. Analogously, if S_x does not display $ab|c$, then S_y does not display $ab|c$, another contradiction. Thus, both S_x and T_u display all triplets that are displayed by B', and S_x therefore displays T_p by Lemma 6.6, yielding a final contradiction to the assumption that S is not canonical for u. □

As stated above, we compute $M(u)$ for each vertex u in T in a bottom-up fashion. We will now show that $|M(u)| \leq k$ for each $u \in V_T$.

Lemma 6.9. *Let N be a k-labeled phylogenetic tree and let $T := (V_T, A_T)$ be a single-labeled phylogenetic tree. Then, it holds for each $u \in V_T$ that $|M(u)| \leq k$.*

Proof. We prove the statement by induction over the height of a vertex u in T. If the height of u is 0, that is, u is a leaf, then $M(u)$ contains all leaves in N that have the same label as u. As N is k-labeled, each candidate set $M(u)$ for a

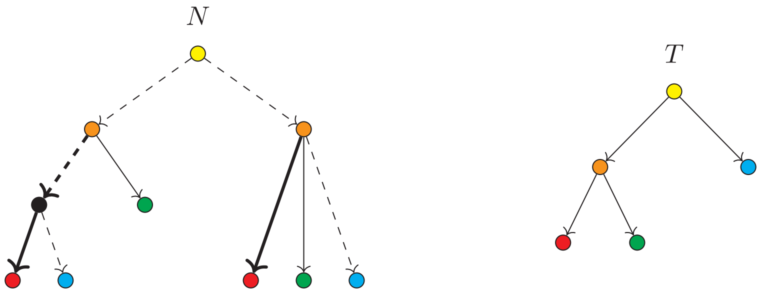


Figure 6.4: Two phylogenetic trees N (left-hand side) and T (right-hand side). The vertices in T are colored and for each vertex u in T all vertices in $M(u)$ in N are colored with the same color as u. The ascending paths of the two red vertices in N are drawn with bold arcs and the ascending paths of the two blue vertices are indicated by dashed arcs.

leaf u is of size at most k. If u is not a leaf, then let c be a child of u in T and assume that $|M(c)| \leq k$. Consider any vertex $v \in M(u)$. Since N_v displays T_u, there is by Lemma 6.8 a subtree S_v of N_v that is canonical for T_u. Hence, there is a vertex $w \in M(c)$ in S_v, that is, S_w displays T_c and w is a candidate for c. Note that v is the only ancestor of w in $M(u)$ as $M(u)$ only contains minima. Thus, any vertex in $M(u)$ has a unique ancestor in $M(c)$ and since $|M(c)| \leq k$, it holds that $|M(u)| \leq k$. □

Note that the proof of Lemma 6.9 also states that for each vertex u in T that is not a leaf, each child c of u in T, and each $w \in M(c)$, there is at most one ancestor v of w in N which is contained in $M(u)$. We call the unique v-w-path in N the *ascending path* of w with respect to c and we omit mentioning c if it is clear from the context. An example of ascending paths is given in Figure 6.4. We next present a crucial lemma about ascending paths which states that ascending paths with respect to two vertices c_1 and c_2 are arc-disjoint unless c_1 and c_2 are siblings in T. Afterwards, we present our k-SAT program for Soft Tree Containment on k-labeled phylogenetic trees using the notions of candidate sets and ascending paths.

Lemma 6.10. *Let N be a multi-labeled phylogenetic tree, let T be a single-labeled phylogenetic tree, and let N display T. Let S be a canonical subtree of N for T. Let u and v be two distinct vertices in T such that neither of them is the* *root of T and u and v are not siblings in T. Let $\mathrm{LCA}_S(\mathcal{L}(T_u))$ and $\mathrm{LCA}_S(\mathcal{L}(T_v))$ have ascending paths R and Q with respect to u and v, respectively. Then, R and Q are arc-disjoint.*

Proof. To prove the statement, we distinguish between the two cases where u and v are in an ancestor-descendant relation or not. If u and v are in an ancestor-descendant relation, then without loss of generality let $u <_T v$. Let p be the parent of u in T. Note that $p \leq_T v$ and hence $\mathrm{LCA}_S(\mathcal{L}(T_p)) \leq_S \mathrm{LCA}_S(\mathcal{L}(T_v))$. Thus, each vertex in the ascending path R of u is either v or a descendant of v in T. Since the ascending path Q of v only contains v and ancestors of v in T, it holds that R and Q share at most one vertex (v) and no arcs.

If u and v are not in an ancestor-descendant relation in T, then assume towards a contradiction that the ascending paths R and Q share an inner vertex z. Since z is an ancestor of both u and v in T, it holds that $\mathcal{L}(T_u) \cup \mathcal{L}(T_v) \subseteq \mathcal{L}(T_z)$. As u and v are not siblings in T, one of u and v has a parent p that is not in an ancestor-descendant relation with the other. Assume without loss of generality that p is the parent of u. Since v and p are not in an ancestor-descendant relation and since T is a single-labeled phylogenetic tree, it holds that

$$\mathcal{L}(T_p) \cap \mathcal{L}(T_z) \supseteq \mathcal{L}(T_u) \neq \emptyset \text{ and } \mathcal{L}(T_z) \setminus \mathcal{L}(T_p) \supseteq \mathcal{L}(T_v) \neq \emptyset.$$

Since S is canonical, it holds that $y := \mathrm{LCA}_S(\mathcal{L}(T_p)) \in M(p)$ and, thus, the ascending path R starts in y. As z is an inner vertex of R, it holds that $z <_S y$, implying

$$\mathcal{L}(T_p) \setminus \mathcal{L}(T_z) \neq \emptyset.$$

Concluding, it holds that

$$\mathcal{L}(T_p) \cap \mathcal{L}(T_z) \neq \emptyset, \ \mathcal{L}(T_z) \setminus \mathcal{L}(T_p) \neq \emptyset, \text{ and } \mathcal{L}(T_p) \setminus \mathcal{L}(T_z) \neq \emptyset$$

and, by Lemma 6.5, this contradicts the assumption that S softly display T. $\square$

We next present the idea behind the main result in this section. To this end, let r be the root of T. Clearly, N displays T if and only if $M(r) \neq \emptyset$. Hence, it remains to show how to compute $M(u)$ given $M(v)$ for all $v \neq u$ in T_u. We do so via a reduction to k-SAT that checks for each y in N whether $y \in M(u)$. Therein, we have a variable $x_{z \to c}$ for each vertex $c \neq u$ in T_u and $z \in M(c)$ that represents whether S_z displays T_c, where S is the canonical subtree of N_y for T_u. The formula then checks whether these choices are consistent, that is, if $x_{a \to w}$ and $x_{z \to v}$ are set to true and w is a descendant of v in T, then a is a descendant of z in N (or $a = z$). Finally, the formula checks whether these choices satisfy Lemma 6.10. After presenting the formula, we will prove that it is correct, that is, $\varphi_{y\to u}$ is satisfiable if and only if N_y displays T_u.

Construction 6.11. Construct $\varphi_{y\to u}$ as follows. For each $v \neq u$ in T_u and for each $z \in M(v)$, introduce a variable $x_{z\to v}$. Moreover for each $v \neq u$ in T_u

(1) add the clause $\bigvee_{z\in M(v)} x_{z\to v}$,

(2) for each pair $z_1, z_2 \in M(v)$ of vertices, add the clause $\neg x_{z_1\to v} \vee \neg x_{z_2\to v}$,

(3) if the parent p of v in T_u is not u, then for all $z \in M(v)$ and all $q \in M(p)$ with $z \not\leq_N q$, add the clause $\neg x_{z\to v} \vee \neg x_{q\to p}$, and

(4) for each $w \neq u$ in T_u that is not a sibling of v, each $z_1 \in M(v)$, and each $z_2 \in M(w)$ such that the ascending paths of z_1 and z_2 in N_x share an arc, add the clause $\neg x_{z_1\to v} \vee \neg x_{z_2\to w}$.

Note that the ascending path of z or q in (4) is not defined if v or w is a child of u in T_u as $M(u)$ is not defined. In this case we call the unique y-z-path or the unique y-q-path the ascending path as we test whether $y \in M(u)$. We next show that Construction 6.11 is correct. Since we use the construction to test whether $y \in M(u)$ and since $M(u)$ can, by definition, not contain two vertices that are in an ancestor-descendant relation, we assume that $\varphi_{z\to u}$ is not satisfiable for any descendant z of y in T.

Lemma 6.12. *Let u be a vertex in T and let y be a vertex in N such that for each descendant d of y in N it holds that $\varphi_{d\to u}$ is not satisfiable. Then, $\varphi_{y\to u}$ is satisfiable if and only if N_y displays T_u.*

Proof. We start by showing that if N_y displays T_u, then $\varphi_{y\to u}$ is satisfiable. To this end, let S be a canonical subtree of N_y that displays T_u. Note that S exists due to Lemma 6.8. Let β be a truth assignment for $\varphi_{y\to u}$ that sets each variable $x_{z\to v}$ to true if and only if $z = \mathrm{LCA}_S(\mathcal{L}(T_v))$. We will show that all clauses in Construction 6.11 are satisfied by this assignment. Note that for each $v \neq u$ in T it holds that S_z displays T_v and $z \in M(v)$ where $z := \mathrm{LCA}_S(\mathcal{L}(T_v))$. Hence each clause of type (1) is satisfied by β. Moreover, since the LCA in S is unique (as S is a tree), also all clauses of type (2) are satisfied by β.

Assume towards a contradiction that a clause of type (3) is not satisfied. Then, there is some v with parent p in T_u such that $y \not\leq_N z$ for some $y \in M(v)$

and $z \in M(p)$ and $\beta(x_{y\to v}) = \beta(x_{z\to p}) = \text{true}$. Since $\mathcal{L}(T_p) \supseteq \mathcal{L}(T_v)$, it holds that $y \leq_S z$. Moreover, since S is a subtree of N, it holds that $y \leq_N z$, contradicting $y \not\leq_N z$. Thus, all clauses of type (3) are satisfied.

Finally, if a clause of type (4) is not satisfied, then there are $x_{y\to v}$ and $x_{z\to w}$ such that

1. v and w are not siblings in T and neither of them is the root of T,
2. $\beta(x_{y\to v}) = \beta(x_{z\to w}) = \text{true}$, and
3. the ascending paths of $y = \text{LCA}_S(\mathcal{L}(T_v))$ and $z = \text{LCA}_S(\mathcal{L}(T_w))$ in N_y share an arc.

This contradicts Lemma 6.10 and therefore all clauses of type (4) are satisfied. Since each clause of $\varphi_{y\to u}$ is satisfied by β, the formula is satisfiable.

We next show that if $\varphi_{y\to u}$ is satisfiable, then N_y displays T_u. To this end, let β be a satisfying truth assignment for $\varphi_{y\to u}$. Let S be the subtree of N_y that contains y and all leaves z such that $\beta(x_{z\to v}) = \text{true}$ for some leaf v in T_u (and no other leaves except for possibly y). We will show that S is canonical for T_u. To this end, we first show that S contains each vertex z such that $\beta(x_{z\to v}) = \text{true}$ for some vertex v in T_u. Note that $\varphi_{y\to u}$ contains for each $v \neq u$ in T_u at most one vertex z such that $\beta(x_{z\to v}) = \text{true}$ as otherwise the respective clause of type (2) was not satisfied by β. It also contains at least one such vertex as otherwise the clause of type (1) was not satisfied. For the sake of readability, we will denote this unique vertex z by $\psi(v)$ for each vertex v. As a special case, we define $\psi(u) := y$. We will show by induction over the height of v that $\psi(v)$ is contained in S and that $S_{\psi(v)}$ displays T_v. The height of a vertex v in a tree is the maximum distance between v and a descendant of v. If v is a leaf, then $\psi(v)$ is by definition contained in S, and $S_{\psi(v)}$ displays T_v. If v is not a leaf, then let c be a child of v in T_u. Since c has smaller height than v in T_u, it holds by induction hypothesis that $\psi(c)$ is contained in S. If $\psi(v)$ was not contained in S, then $\psi(v)$ is not an ancestor of $\psi(c)$. This, however, contradicts the clause of type (3). Hence, each vertex $\psi(v)$ for some vertex $v \neq u$ in T_u is contained in S. It remains to show that $S_{\psi(v)}$ displays T_v. Assume towards a contradiction that $S_{\psi(v)}$ does not display T_v. By Lemma 6.5, there are vertices w in T_v and q in $S_{\psi(v)}$ and leaves

$$a \in \mathcal{L}(S_q) \setminus \mathcal{L}(T_w),\ b \in \mathcal{L}(T_w) \setminus \mathcal{L}(S_q),\ \text{and } c \in \mathcal{L}(T_w) \cap \mathcal{L}(S_q).$$

On the one hand, note that $a <_S q$ and $c <_S q$ and therefore there is a highest ancestor α of a in T with $\psi(\alpha) \leq_S q$ and a highest ancestor γ of c in T

with $\psi(\gamma) \leq_S q$. By the definitions of α and γ, there are parents p_α and p_γ of α and γ, respectively, such that $\psi(p_\alpha) \not\leq_S q$ and $\psi(p_\gamma) \not\leq_S q$. Hence, the ascending paths of $\psi(\alpha)$ and $\psi(\gamma)$, respectively, share q as an inner vertex and the arc (p_q, q) where p_q is the parent of q in S. Note that q has a parent as there is a leaf with label b that is not contained in S_q but in $S_{\psi(v)}$. On the other hand, note that $\alpha <_T w$ and $\gamma \not\leq_T w$, implying that α and γ are not siblings in T, contradicting the assumption that all clauses of type (4) are satisfied by β. $\square$

We next show our main result in this chapter, that is, a k-SAT program for Soft Tree Containment on k-labeled graphs for each $k \geq 2$. We mention that the program resulting from Soft Tree Containment on single-labeled phylogenetic trees contains clauses with two literals (the clauses of types (3) and (4) in Construction 6.11). Since 2-SAT formulas are linear-time solvable, the following result proves that Soft Tree Containment on single-labeled phylogenetic trees is polynomial-time solvable. In the paper on which this chapter is based, we also present a linear-time algorithm for Soft Tree Containment on single-labeled phylogenetic trees [BMW18].

Theorem 6.13. *For each $k \geq 2$, one can decide in $O(n^5 \cdot k^2)$ time whether a k-labeled phylogenetic tree N softly displays a single-labeled phylogenetic tree T using $O(n^2)$ queries of size $O(n^2 \cdot k^2)$ to k-SAT.*

Proof. The algorithm computes for each vertex u in T at most k vertices $M(u)$ such that for each $v \in M(u)$ the subtree N_v displays T_u and for no descendant w of v it holds that N_w displays T_u. It computes this set $M(u)$ bottom-up for each vertex u in T. The pseudo-code is given in Algorithm 6.1. All possible candidates for vertices in $M(u)$ that are found by the algorithm are compared in Line 16 and all non-minima are removed. Hence, the set $M(u)$ computed by the algorithm only contains minima. Hence, it remains to show that if for a vertex v in N it holds for no descendant w of v that N_w displays T_u, then $v \in M(u)$ if and only if N_v displays T_u. Let v be such a vertex. Note that since for no descendant of w of v it holds that N_w displays T_u, it holds by Lemma 6.12 that $\varphi_{w \to u}$ is not satisfiable for any descendant w of v in N. Hence, Lemma 6.12 states that $\varphi_{v \to u}$ is satisfiable if and only if N_v displays T_u. Thus, it remains to show that $v \in M(u)$ if and only if $\varphi_{v \to u}$ is satisfiable. To this end, note that since N_v displays T_u it also displays T_c for any descendant c of u in T. Let c be the child of u chosen in Line 6 and let $v' \in M(c)$ be a descendant of v (or $v' = v$). We now consider the iteration of Line 7 where $w = v'$. If $v' = v$, then the algorithm adds v to $M(u)$. If $v' \neq v$, then note that v' is a descendant

Algorithm 6.1: A k-SAT program for Soft Tree Containment on k-labeled phylogenetic trees.

```
Input: A $k$-labeled phylogenetic tree $N$ and a single-labeled
       phylogenetic tree $T$.
Output: true if $N$ displays $T$ and false otherwise.
$r \leftarrow$ root of $T$
foreach vertex $u$ in $T$ do                         // in a bottom-up manner
    $M(u) \leftarrow \emptyset$
    if $u$ is a leaf in $T$ then $M(u) \leftarrow \{v \in N \mid \mathcal{L}(v) = \mathcal{L}(u)\}$
    else
        $c \leftarrow$ any child of $u$ in $T$              // $c$ can be chosen arbitrarily
        foreach $w \in M(u)$ do
            $w' \leftarrow w$
            while $w' \neq \bot$ do
                construct $\varphi_{w' \to u}$
                if $\varphi_{w' \to u}$ is satisfiable then
                    $M(u) \leftarrow M(u) \cup \{w'\}$
                    $w' \leftarrow \bot$
                else
                    $w' \leftarrow$ parent of $w'$ in $T$ // If $w' = r$, then $w' \leftarrow \bot$
        if $\exists a, b \in M(u).\ a \leq_N b$ then remove $b$ from $M(u)$
if $M(r) \neq \emptyset$ then return true
else return false
```

of v and hence $\varphi_{v' \to u}$ is not satisfiable. The algorithm then iteratively tries each ancestor v^* of v' and checks whether $\varphi_{v^* \to u}$ is satisfiable. The formula $\varphi_{v^* \to u}$ is not satisfiable for each descendant v^* of v and hence eventually $\varphi_{v \to u}$ is tested. By assumption, $\varphi_{v \to u}$ is satisfiable and hence v is added to $M(u)$. Thus, the set $M(u)$ is computed correctly by Algorithm 6.1 for each vertex u in T. Finally, observe that N displays T if and only if $M(r) \neq \emptyset$ where r is the root of T.

It remains to analyze the number and sizes of the constructed formulas and the running time of the algorithm. Note that all clauses of type (1) are of size at most k and all other clauses are of size at most 2. Hence for each $k \geq 2$ the resulting formulas are k-SAT formulas. We first analyze the size of each formula.

Note that there are $O(n)$ clauses of type (1), $O(n \cdot k^2)$ clauses of type (2) and (3), and $O(n^2 \cdot k^2)$ clauses of type (4). Since only clauses of type (1) are not of constant size, each formula is of size $O(n^2 \cdot k^2)$.

Note that we construct at most one formula $\varphi_{v \to u}$ for each pair (v, u) of vertices where v is a vertex of N and u is a vertex in T. Hence, there are at most n^2 such formulas. It remains to analyze the running time of the algorithm (excluding the steps to solve the k-SAT formulas). The running time is dominated by the time to construct all formulas. Since we construct $O(n^2)$ formulas of size at most $O(n^2 \cdot k^2)$, it remains to analyze the running time to construct each clause. Clauses of type (1) and (2) take constant time per literal. Clauses of type (3) and (4) take $O(n)$ time to construct. Thus, the overall running time is $O(n^2 \cdot (n^2 \cdot k^2) \cdot n) = O(n^5 \cdot k^2)$. □

A direct consequence of Theorem 6.13 is that Soft Tree Containment on 2-labeled phylogenetic trees can be solved in $O(n^5)$ time. This is a somewhat surprising application of 2-SAT programming as it is not apparent that the difference between 2-labeled phylogenetic trees and 3-labeled phylogenetic trees and the difference between 3-labeled phylogenetic trees and 4-labeled phylogenetic trees should be very dissimilar.

Corollary 6.14. *It can be verified in $O(n^5)$ time whether a 2-labeled phylogenetic tree N softly display a single-labeled phylogenetic tree T.*

We remark that this running time is not optimized and a more careful analysis using the amortized running time leads to a cubic running time [BMW18].

6.3.2 Reduction from k-SAT

In this subsection, we supplement the result from the previous subsection in the sense that we show that k-SAT reduces to Soft Tree Containment on k-labeled trees. As a consequence, Soft Tree Containment is *NP*-hard even when restricted to 3-labeled phylogenetic trees. To this end, we make a slight detour and first show a reduction from k-SAT to a rather technically looking version of Independent Set that will turn out to be equivalent to a very natural variant of Colorful Independent Set. From this variant of Colorful Independent Set, we will then show a reduction to Soft Tree Containment on k-labeled trees.

The mentioned variant of Independent Set is based on the notion of $A \bowtie B$ graphs. Therein, A and B are graph classes and a graph $G = (V, E)$ is in $A \bowtie B$

if its edge set E can be partitioned into two sets E_1 and E_2 of edges such that $G_1 \coloneqq (V, E_1)$ is in graph class A and $G_2 \coloneqq (V, E_2)$ is in B [BBN19]. We are interested in the case where A is the disjoint union of P_3's and B is the disjoint union of cliques of size at most k. Disjoint unions of cliques are also known as cluster graphs. This leads to the following special case of Independent Set.

$P_3 \bowtie$ Cluster Independent Set
Input: An integer ℓ and a graph $G \coloneqq (V, E)$ where $E = E_1 \uplus E_2$ such that $G_1 \coloneqq (V, E_1)$ is a collection of disjoint P_3's and $G_2 \coloneqq (V, E_2)$ is a cluster graph in which each clique has size at most k.
Question: Does G contain an independent set of size ℓ?

Van Bevern et al. [Bev+15] showed via a reduction from 3-SAT that Independent Set is *NP*-hard on $A \bowtie B$ graphs[1] unless A and B only contain cluster graphs. We modify their reduction to be able to reduce from k-SAT. The basic idea is to represent each clause by a clique and each variable by a cycle of even length. The largest independent set can contain at most half of the vertices in each cycle and at most one vertex from each clique and it contains that many vertices if and only if the k-SAT formula is satisfiable. In the following, we denote the number of literals in a clause C by $|C|$. Note that we can assume without loss of generality that each variable occurs at most once in each clause as otherwise the clause is either trivially satisfied (if one occurrence is positive and the other negative) or one of the literals can be removed (if both occurrences are positive or both are negative). Moreover, we assume that each variable occurs at least twice in the formula as we can otherwise always satisfy the clause in which the variable occurs.

Construction 6.15. Consider an instance φ of k-SAT. Let φ have n variables $x_1, x_2, \dots, x_n$ and m clauses $C_1, C_2, \dots, C_m$ such that each variable occurs at least twice in φ and at most once in each clause. For each variable x_i let J_i be the list of indices of clauses that contain x_i or $\neg x_i$ and let $J_i[\ell]$ denote the ℓ^{th} element of this list. Construct a graph $G \coloneqq (V, E_1 \uplus E_2)$ as follows. For each variable x_i construct a cycle V_i of $2|J_i|$ vertices $u_i^1, \overline{u}_i^1, u_i^2, \overline{u}_i^2, \dots, u_i^{|J_i|}, \overline{u}_i^{|J_i|}$ such that $\overline{u}_i^k$ is adjacent to u_i^k and u_i^{k+1} for each $k \in [|J_i| - 1]$ and u_i^1 and $\overline{u}_i^{|J_i|}$ are adjacent. We call V_i a variable gadget. For each clause C_j

[1]We remark that A and B have to be closed under disjoint union and taking an induced subgraph. Moreover, at least one graph in A and one graph in B has to contain an edge.

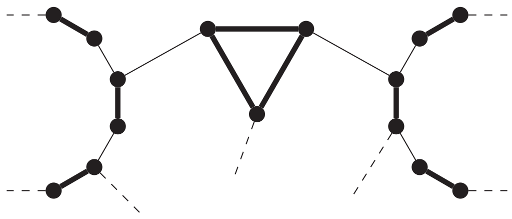


Figure 6.5: Illustration of a small extract of the resulting graph of Construction 6.15. The triangle in the middle is the clause gadget for a clause of size three and the two cycles left and right are variable gadgets corresponding to variables that occur in this clause. The thin edges are contained in E_1 and the bold edges are contained in E_2.

that contains variables $x_{a_1}, x_{a_2}, \ldots, x_{a_{|C_j|}}$ construct a clique that contains vertices $w_j^{a_1}, w_j^{a_2}, \ldots, w_j^{a_{|C_j|}}$. We call this clique a clause gadget. For each variable x_i and each $\ell \in [|J_i|]$, connect $w^i_{J_i[\ell]}$ to $\overline{u}_i^\ell$ if $C_{J_i[\ell]}$ contains x_i and to u_i^ℓ if $C_{J_i[\ell]}$ contains $\neg\, x_i$. The edge set E_1 consists of all edges between two vertices v and w where v is contained in a vertex gadget and w is contained in a clause gadget. Moreover, E_1 contains the edge $\{u_i^k, \overline{u}_i^k\}$ for each variable gadget V_i and each $k \in [|J_i|]$. The edge set E_2 contains all constructed edges that are not contained in E_1.

See Figure 6.5 for an illustration of Construction 6.15. We show that the graph $G_1 := (V, E_1)$ consists only of disjoint P_3's. Note that E_1 contains all edges $\{u_i^k, \overline{u}_i^k\}$ and exactly one of the two vertices in $\{u_i^k, \overline{u}_i^k\}$ is adjacent to a vertex in a clause gadget in G_1. Since each vertex in a clause gadget has degree exactly one in G_1, this proves that G_1 only consist of disjoint P_3's. Next, observe that $G_2 := (V, E_2)$ consists of disjoint cliques of size at most k. In each variable gadget it contains every other edge, that is, a matching (disjoint cliques of size two) and it contains all edges between vertices in clause gadgets which are by definition of size at most k. We next show that Construction 6.15 is correct.

Lemma 6.16. *Let φ be an instance of k-SAT in which each variable occurs at least twice in φ and at most once in each clause. Then, φ is satisfiable if and only if the graph $G := (V, E_1 \uplus E_2)$ resulting from Construction 6.15 has an independent set of size ℓ where ℓ is the number of cliques in $G_2 := (V, E_2)$.*

Proof. We start by showing that if G contains an independent set of size ℓ, then φ is satisfiable. To this end, let I be an independent set of size ℓ in G. Note that I contains exactly one vertex from each clique in G_2 and therefore for each variable gadget V_i it either contains u_i^1 or $\overline{u}_i^1$. By construction of V_i, it holds that if $u_i^1 \in I$, then $u_i^\ell \in I$ for all $\ell \in [|J_i|]$. Analogously, if $\overline{u}_i^1 \in I$, then $\overline{u}_i^\ell \in I$ for all $\ell \in [|J_i|]$. We now describe how to construct a satisfying truth assignment for φ. For each variable x_i, we set $\beta(x_i) :=$ true if $u_i^1 \in I$ and $\beta(x_i) :=$ false if $\overline{u}_i^1 \in I$. It remains to show that this truth assignment β satisfies all clauses in φ. To this end, consider any clause C_j. Since I contains exactly one vertex from each clause gadget (each such gadget induces a clique in G_2), it holds that $w_j^i \in I$ for some $i \in [|C_j|]$. By construction, the variable x_i occurs in C_j (exactly once). If C_j contains the literal $\neg x_i$, then w_j^i is adjacent to u_i^h for some $h \in [|J_i|]$ and, since I is an independent set, I does not contain u_i^h. Thus, $\overline{u}_i^1 \in I$ and therefore $\beta(x_i) =$ false and C_j is satisfied by β. If C_j contains the literal x_i, then w_j^i is adjacent to $\overline{u}_i^h$ for some $h \in [|J_i|]$ and analogously $\overline{u}_i^1 \in I$. Thus, C_j is satisfied by β as $\beta(x_i) =$ true. Since each clause is satisfied by β, this concludes the first direction of the proof.

It remains to show that if φ is satisfiable, then G contains an independent set of size k. Let β be a satisfying truth assignment for φ. We construct an independent set I of size ℓ for G as follows. For each variable x_i, if $\beta(x_i) =$ true, then I contains all vertices u_i^h for $h \in [|J_i|]$ and if $\beta(x_i) =$ false, then I contains all vertices $\overline{u}_i^h$ for $h \in [|J_i|]$. For each clause C_j, let x_i be a variable that satisfies C_j under assignment β and let I contain w_j^i. Observe that I is of size ℓ as it contains exactly one vertex of each clique in G_2. It remains to show that I is indeed an independent set. Assume towards a contradiction that I was not an independent set. Then it contains two adjacent vertices. Note that it does not contain two adjacent vertices from variable gadgets as it contains every second vertex from the respective cycle. It does not contain two adjacent vertices from clause gadgets either as it contains exactly one vertex from each clause gadget and vertices from different clause gadgets are not adjacent in G. Hence, I contains a vertex w_j^i from a clause gadget and a vertex v from a variable gadget such that v and w_j^i are adjacent. If $w_j^i \in I$, then x_i satisfies C_j by construction, that is, $\beta(x_i) =$ true if C_j contains the literal x_i and $\beta(x_i) =$ false if C_j contains the literal $\neg x_i$. We distinguish between the two cases where C_j contains the literal x_i or the literal $\neg x_i$. If C_j contains the literal x_i, then by construction w_j^i is only adjacent to vertices in the clause gadget for C_j and to $\overline{u}_i^h$ for some $h \in [|J_i|]$. Since $\beta(x_i) =$ true, it holds that $u_i^h \in I$ and $\overline{u}_i^h \notin I$, a

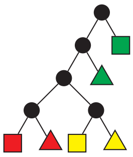


Figure 6.6: A gadget for a P_3 in Construction 6.17 where the inner vertex has a green color (the two upper leaves) and the end vertices have red (bottom left) and yellow color (bottom right), respectively. The triangles and squares represent leaves of different labels. There are six different labels in this phylogenetic tree (which are represented by a red square, a red triangle, a yellow square, a yellow triangle, a green square, and a green triangle, respectively).

contradiction to the assumption that w_j^i has a neighbor in I. If C_j contains $\neg x_i$, then by construction w_j^i is only adjacent to vertices in the clause gadget for C_j and to u_i^h for some $h \in [|J_i|]$. Since $\beta(x_i) = \text{false}$, it holds that $u_i^h \notin I$, which is again a contradiction to the assumption that w_j^i has a neighbor in I. Thus, I is an independent set which concludes the proof. □

Note that, by construction, the independent set has to contain exactly one vertex from each clique in G_2. This is equivalent to giving each vertex in G_1 a color that represents in which clique in G_2 the vertex is contained and asking for a colorful independent set, that is, an independent set which contains exactly one vertex of each color. Hence, Lemma 6.16 implies that k-SAT reduces to Colorful Independent Set on disjoint P_3's where each color appears at most k times and no P_3 contains two vertices of the same color. We next reduce this variant of Colorful Independent Set to Soft Tree Containment on k-labeled trees. The basic idea is to construct a gadget as shown in Figure 6.6 for each P_3 in the input graph and connect all of these gadgets by an arbitrary binary tree whose leaves are the roots of the respective gadgets. By Lemma 6.8, if this phylogenetic tree N displays T, then it contains a single-labeled phylogenetic tree S that displays T. We will show that S can contain either a leaf that represents the inner vertex in the respective P_3 or only leaves that represent end vertices of the respective P_3. Hence, we can use S to construct an independent set.

Construction 6.17. Given a vertex-colored collection $G := (V, E)$ of P_3's where each color occurs at most k times, we construct a k-labeled phylogenetic

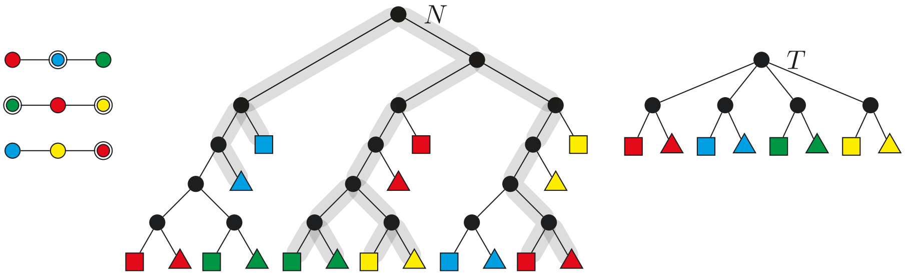


Figure 6.7: Illustration of Construction 6.17.
Left: The initial instance of Colorful Independent Set on disjoint P_3's with 4 colors (red, blue, green, and yellow) where each color occurs at most thrice. The encircled vertices represent a solution.
Right: The single-labeled phylogenetic tree T resulting from Construction 6.17.
Middle: The binary 3-labeled phylogenetic tree N resulting from Construction 6.17. The highlighted edges represent the single-labeled subtree S of N that displays T and that corresponds to the marked solution on the left-hand side.

tree N and a single-labeled phylogenetic tree T as follows. Both phylogenetic networks contain two different labels i_1 and i_2 for each color i in G.

Construct T by first creating a star that has exactly one leaf of each color occurring in G. Then, for each leaf x with color i, adding two new leaves labeled with i_1 and i_2, respectively. Since x is not a leaf any more, its label is removed.

The k-labeled phylogenetic tree N is constructed as follows. We start with a gadget as shown in Figure 6.6 for each $P_3 = (u, v, w)$ in the input graph where red, green, and yellow denote the colors of u, v, and w, respectively. Therein, a triangle of color i represents a leaf labeled with i_1 and a square of color i represents a leaf labeled with i_2. Finally, add an arbitrary binary tree that has a leaf for each P_3 in G and identify each such leaf with the root of the respective constructed gadgets.

An example of Construction 6.17 is given in Figure 6.7. We conclude this subsection with the proof that k-SAT reduces to Soft Tree Containment on k-labeled phylogenetic trees and a simple corollary that states NP-hardness.

Proposition 6.18. *k-SAT reduces for each k to Soft Tree Containment on binary k-labeled phylogenetic trees.*

Proof. Note that k-SAT reduces by Lemma 6.16 to Colorful Independent Set on disjoint P_3's where each color appears at most k times. We then apply Construction 6.17 to the constructed instance of Colorful Independent Set. Since the resulting phylogenetic tree from Construction 6.17 is k-labeled and binary, it remains to show that this construction is correct, that is, N displays T if and only if the given collection $G := (V, E)$ of P_3's has a colorful independent.

We first show that if N displays T, then there is a colorful independent set in G. If N displays T, then, by Lemma 6.8, N contains a single-labeled phylogenetic tree S that displays T. By Observation 6.1, this is equivalent to T displaying S. Let Q be the set of vertices in G such that for each vertex $v \in Q$ of color c, S contains a vertex of color c_1 in the gadget for the respective P_3 that v is in. Since S displays T, it contains a leaf with label c_1 and a leaf with label c_2 for each color c. Moreover, since S is single-labeled it contains exactly one vertex with label c_1 for each color c and therefore Q is colorful, that is, it contains exactly one vertex of each of the colors in G. Hence, it remains to show that Q is an independent set in G. Assume towards a contradiction that Q is not an independent set, that is, it contains two adjacent vertices u and w. Let without loss of generality u be an inner vertex in a P_3 and let c and d be the colors of u and w respectively. Since $u, w \in Q$, it holds that S contains the leaf with label c_1 and the leaf with label d_1 in the same gadget. By construction, S displays the triplet $c_1d_1|c_2$ and T displays the triplet $c_1c_2|d_1$ firmly. By Lemma 6.6(b), this contradicts the fact that T displays S.

We conclude the proof by showing that if G contains a colorful independent set I, then N displays T. To this end, let I be a colorful independent set in G. We will show that there is a single-labeled subtree S of N that displays T. This implies by Lemma 6.8 that N displays T. For each vertex $v \in I$ of color c, let S contain the two leaves with labels c_1 and c_2 in the gadget for the respective P_3 that v is in. Since I is colorful, S contains exactly one leaf of each label and it therefore remains to show that S displays T.

Assume towards a contradiction that S does not display T. This is, by Observation 6.1, equivalent to T not displaying S. In this case, there is, by Lemma 6.6, a triplet $xy|z$ that is firmly displayed by S but not softly displayed by T. By Observation 6.3(b), T then displays one of the triplets $xz|y$ or $yz|x$ firmly. Let T without loss of generality display $xz|y$ firmly. By construction of T, it holds that $y := c_1$ and $z := c_2$ (or $y := c_2$ and $z := c_1$) for some color c. By construction of S, it holds that the two leaves labeled with c_1 and c_2 in S are in the same gadget. Hence, c_1 and c_2 correspond to an inner vertex in the respective P_3 as otherwise there is no label x such that S displays the triplet $c_1x|c_2$ (or $c_2x|c_1$). By construction, I contains the inner vertex v of color c in the respective P_3. Moreover, it holds that the leaf with label x is also contained in the same gadget and thus I contains one of the two end vertices in the same P_3 as v, a contradiction to the fact that I is an independent set. ☐

Since k-SAT is *NP*-hard for each $k \geq 3$, it holds that Soft Tree Containment is *NP*-hard on binary k-labeled phylogenetic trees and, in particular, when restricted to 3-labeled phylogenetic trees.

Corollary 6.19. *Soft Tree Containment is NP-hard, even if the input network N is a binary 3-labeled phylogenetic tree.*

6.4 Concluding Remarks

We initiated research into a practically relevant variant of Tree Containment handling soft polytomies. We again defer the discussion on 2-SAT programming as a technique to the concluding chapter of this thesis and focus on Soft Tree Containment here. We laid the mathematical foundations to dealing with soft polytomies and showed the dichotomy result that Soft Tree Containment on k-labeled phylogenetic trees is polynomial-time solvable if $k \leq 2$ and *NP*-hard if $k \geq 3$. Further improving the running time of the polynomial-time algorithm for 2-labeled phylogenetic trees (e. g. within the context of *FPT in P* as done in Chapter 3) and empirically evaluating it on real-world data sets are clear avenues for further research.

Motivated by our hardness result, the search for parameterized or approximation algorithms is another logical next step. Previous work for Tree Containment [GLZ16, Wel18] might lend promising ideas and parameterizations to this effort.

Chapter 7

Reachable Objects

In this chapter, we will investigate a problem from the widely-studied field of resource allocation under preferences, having applications in areas such as artificial intelligence and economics. Conceptually, we will develop a 2-SAT program where the truth assignment of a variable does *not* represent picking some element into a solution or not. It rather represents which of two elements is picked into a solution. These types of 2-SAT programs are so far very rare in the literature. We mention that the 2-SAT program we develop in this chapter does not meaningfully generalize to a k-SAT program and therefore the 2-SAT program is not a special case of a reduction to k-SAT. In Chapter 8, we will analyze the structure of the problem we study here and observe which structural elements enable 2-SAT programming. This will lead us to a rule of when 2-SAT programming can be a promising tool for solving algorithmic problems.

Regarding resource allocation under preferences, we will investigate the Reachable Object problem which generalizes the well-known Housing Market problem [SS74]. In Reachable Object, agents are organized in a graph and two agents can only swap resources if they share an edge in the graph. This restriction models the situation where not all agents are able to communicate and swap with each other. We start with a dichotomy result regarding the number of objects each agent prefers over its initially held object and continue with investigating the special case where each agent has at most two neighbors in the graph. Using 2-SAT programming, we will show that this special case is polynomial-time solvable. The problem remains *NP*-hard for the case where each agent has at most four neighbors in the graph [SW18].

Resource allocation under preferences is a major topic in society and technology [Wal15]. It has also proven to be a key issue in a world of limited resources and allocating indivisible resources is well-studied in the context of multiagent systems [BCM16]. It has numerous applications e. g. in contexts of food-banks, when sharing charitable donations between cities or communities, or when allocating physical to virtual resources in virtualization technologies [BKN18]. There are several versions studied in the literature that try to optimize for different criteria such as Pareto optimality, fairness, or social welfare [Abr+05, Rot82, SU10].

In the field of resource allocation under preferences, one is interested in distributing a set of (divisible or indivisible) objects among a set of agents who value the objects differently. We focus entirely on indivisible objects here and consider the special case where each agent initially holds exactly one object. While a large body of research in the literature takes a *centralized* approach that globally controls and reallocates an object to each agent, we pursue a *decentralized* strategy where any pair of agents may locally *swap* objects as long as this leads to an improvement for both of them, that is, they both value the object they get over the one they give away [DBC15]. We are then interested whether there is a sequence of such *rational trades* that leads to a situation where a given agent obtains a given object. Other examples of recently studied problems regarding allocations of indivisible resources under social network constraints are envy-free allocations [Bey+19, BKN18], Pareto-optimal allocations [IP19], and stable matchings [ABH17, AV09].

The main contribution of this chapter is a polynomial-time algorithm for Reachable Object on cycles and the following dichotomy result. If each agent prefers at most two other objects over the object it initially holds, then the problem is linear-time solvable. If some agents prefer more than two objects over their initially held object, then the problem is *NP*-hard. The polynomial-time reduction in the hardness result also shows that the problem remains *NP*-hard if the underlying graph is a clique, that is, all agents can pairwise swap with one another. It might be tempting to think that the hardness then stems from the density of the underlying graph as cycles are very sparse. This assumption is, however, false as the problem is known to be *NP*-hard even when the input graph is a tree [GLW17].

Section 7.2 is dedicated to the dichotomy result and in Section 7.3 we will present our 2-SAT-programming-based polynomial-time algorithm for Reachable Object on cycles. Me mention that the positive result in the dichotomy part is based on dynamic programming.

7.1 Problem Definition and Related Work

Let $V := \{1, 2, \ldots, n\}$ be a set of n agents and let $X := \{x_1, x_2, \ldots, x_n\}$ be a set of n objects. Each agent $i \in V$ has a *preference list* over the objects in X, which is a strict linear order on X. This list is denoted as $\succ_i$ and we omit the subscript i if the agent is clear from the context. For two objects x_j, x_ℓ, the notation $x_j \succ_i x_\ell$ means that agent i *prefers* x_j *over* x_ℓ. A *preference profile* $\mathcal{P}$ is a collection $(\succ_i)_{i \in V}$ of preference lists of the agents in V. An *assignment* is a bijection $\sigma\colon V \to X$, where each agent i is assigned exactly one object $\sigma(i) \in X$. Since assignments are bijections, we will also use $\sigma^{-1}(x_i)$ to denote the agent that holds x_i in assignment σ.

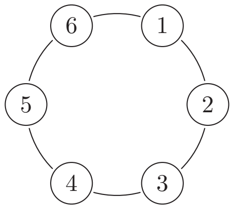
Let $G := (V, E)$ be a graph where the set V of agents is the set of vertices. An edge in this graph models that two agents know and trust each other enough to swap objects. We say that an assignment σ admits *a rational trade for two agents i and j*, denoted as $\tau = \{(i, \sigma(i)), (j, \sigma(j))\}$, if the vertices corresponding to i and j are adjacent in the graph ($\{i, j\} \in E$) and each of the two agents prefers the other's assigned object over its own object ($\sigma(j) \succ_i \sigma(i)$ and $\sigma(i) \succ_j \sigma(j)$). After performing the swap specified by τ, agent i holds object $\sigma(j)$, agent j holds object $\sigma(i)$, and the other agents keep their objects. To describe this move, we say that *objects $\sigma(i)$ and $\sigma(j)$ are swapped over edge* $\{i, j\}$. Sometimes, we also say that object $\sigma(i)$ (or $\sigma(j)$) *passes through* edge $\{i, j\}$ or *moves* from agent i to j.

A *sequence of swaps* is a sequence $(\sigma_0, \sigma_1, \ldots, \sigma_t)$ of assignments where for each index $k \in \{0, 1, \ldots, t-1\}$ there are two agents $i, j \in V$ for which σ_k admits a swap $\tau = \{(i, \sigma_k(i)), (j, \sigma_k(j))$ such that

1. $\sigma_{k+1}(i) = \sigma_k(j)$,
2. $\sigma_{k+1}(j) = \sigma_k(i)$, and
3. $\sigma_{k+1}(z) = \sigma_k(z)$ for each remaining agent $z \in V \setminus \{i, j\}$.

We call an *assignment σ' reachable from another assignment σ* if there is a sequence $(\sigma_0, \sigma_1, \ldots, \sigma_t)$ of swaps such that $\sigma_0 = \sigma$ and $\sigma_t = \sigma'$. We say that *an object $x \in X$ is reachable for an agent i from a given initial assignment σ_0* if there is an assignment σ which is reachable from σ_0 with $\sigma(i) = x$.

With these definitions at hand, we can now define the problem Reachable Object introduced by Gourvès et al. [GLW17] which we study in this chapter.



$1 : x_3 \succ x_4 \succ \boxed{x_1}$ $2 : x_1 \succ x_3 \succ x_4 \succ \boxed{x_2}$
$3 : x_1 \succ x_2 \succ x_4 \succ \boxed{x_3}$ $4 : x_5 \succ x_3 \succ \boxed{x_4}$
$5 : x_6 \succ x_3 \succ \boxed{x_5}$ $6 : x_4 \succ x_3 \succ \boxed{x_6}$

Figure 7.1: An example of Reachable Object. The six agents and the graph of agents are depicted on the left-hand side. The preference lists are depicted on the right and the initial assignment σ_0 is illustrated by the boxes in the preference lists (each agent initially holds the object that is drawn in a box in the agent's preference list). Since no agent will agree on receiving an object in any swap that it does not prefer over its initially held object, these objects are for the sake of readability not depicted in the preference lists. The agent I is agent 1 and x is the object x_3. If the underlying graph was complete, then object x_3 would be reachable for agent 1 within one swap. However, if the graph is a cycle as shown, then to reach agent 1 object x_3 has to be swapped along $\{2,3\}$ with object x_2 first and then along $\{1,2\}$ with object x_1. Note that at both edges both incident agents agree to the swap as agent 3 prefers x_2 over x_3 and agent 2 prefers x_3 over x_2 and x_1 over x_3. Finally, agent 1 prefers x_3 over x_1.

Reachable Object
Input: A set V of agents, a set X of objects with $|X| = |V|$, a preference profile $\mathcal{P}$, an initial assignment σ_0, a graph $G := (V, E)$, an agent $I \in V$, and an object $x \in X$.
Question: Is x reachable for I from σ_0?

An example of Reachable Object is given in Figure 7.1. Note that an agent i that gives away a certain object x_j during a sequence of swaps, obtains an object it prefers over x_j and hence agent i will not accept object x_j in the future.

Observation 7.1. *Let $\phi := (\sigma_0, \sigma_1, \dots, \sigma_s)$ be a sequence of swaps, let i be an agent and let x_j be an object. If $\sigma_r(i) = x_j$ and $\sigma_{r+1}(i) \neq x_j$ for some $r \in [s-1]$, then $\sigma_{r'}(i) \neq x_j$ for all $r' > r$.*

Concerning related work, Gourvès et al. [GLW17] introduced Reachable Object and showed that it is *NP*-hard on trees. Moreover, they showed polynomial-time solvability on stars and for a special case on paths, namely when testing whether an object is reachable for an agent positioned on an end vertex of the path. Huang and Xiao [HX20] generalized this special case and showed that Reachable Object on paths is polynomial-time solvable independently of where the target agent I is located on the path. They also considered a version where agents can value different objects equally (that is, the preference lists are not strict) and showed that in this case Reachable Object remains *NP*-hard on paths. Saffidine and Wilczynski [SW18] studied the parameterized complexity of Reachable Object with respect to parameters such as the maximum degree of the input graph or the overall number of swaps allowed in a sequence. They showed that Reachable Object remains *NP*-hard even on graphs with maximum degree at most four. Further, they showed that Reachable Object is *W[1]*-hard when parameterized by the length of the minimum sequence of swaps that leads to agent I obtaining object x. Finally, Reachable Object is *NP*-complete on generalized caterpillars where each hair has length at most two and only one vertex has degree larger than two [Ben+19a].

7.2 Length of Preference Lists

In this section, we will show a complexity dichotomy result with regard to the maximum length of a preference list. Notice that each agent initially holds one object and it will never obtain any object that it does not prefer over its initially held object. Thus, we describe the preference list of an agent only up to its initially held object. The length of the preference list of an agent is then defined as the number of objects the agent likes at least as much as its initially held object and the maximum length of a preference list is the length of a longest preference list of any agent.

The parameter maximum length of a preference list is mainly motivated by the following two scenarios. In many applications each agent only knows some of the objects (e. g. potential buyers usually only visited five to ten houses and do not like all of them or when ranking movies each participant has only seen some of the available movies) and in other applications even when all alternatives are known only a few of them are appealing (e. g. when applying for a job or when choosing food). Notably, Saffidine and Wilczynski [SW18] suggested to study Reachable Object with restrictions on the preference lists.

We will show in Subsection 7.2.1 that instances in which the maximum length of a preference list is at most three can be solved in linear time. We complement this result in Subsection 7.2.2 by showing that Reachable Object is *NP*-hard even if restricted to cases where the maximum length of a preference list is at most four and where the underlying graph is a clique.

7.2.1 Maximum Length at Most Three

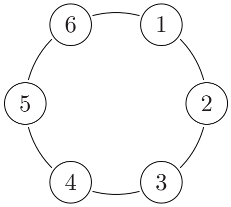
In this subsection, we provide a linear-time algorithm for Reachable Object when the maximum length of a preference list is at most three. The main idea is to reduce Reachable Object to computing an s-t-path in a directed graph. Throughout this subsection, we assume that each agent i initially holds object x_i. Consider all agents that hold the given target object x during a sequence of swaps that leads to agent I obtaining object x. All those agents except for the agent a that initially holds x and agent I must swap their initially held object to receive x and then receive their most preferred object for giving x away. We call those agents *x-forwarder*. Concerning agent I, it might swap its initially held object x_I in order to receive x or it might first receive an object x_w and then swap x_w away in order to receive x. Note that in this case the preference list of agent I is $x \succ x_w \succ \boxed{x_I}$. Since each preference list is of length at most three, the object x_w is unique. Note that all agents that hold object x_w in the mentioned sequence of swaps except for agents I and w must swap their initially held object to receive x_w and then receive their most preferred object for giving x_w away. Analogously to x-forwarder, we call such agents x_w-forwarder. Hence, we basically just consider the case distinction whether agent I is a w-forwarder or not and which objects agent a and w receive in exchange for their initially held objects. We remark that there is a special case where a is an x_w-forwarder and agent w is an x-forwarder. Figure 7.2 gives an example of Reachable Object where the maximum length of preference lists is at most three.

Let $(\sigma_0, \sigma_1, \ldots, \sigma_t)$ be a sequence of swaps. To ease the reasoning, we define τ_i to be the swap that transforms σ_{i-1} into σ_i . Formally, $\tau_i = \{(j, x_p), (k, x_q)\}$ such that

1. $\sigma_{i-1}(j) = \sigma_i(k) = x_p$, and

2. $\sigma_{i-1}(k) = \sigma_i(j) = x_q$.

Using this notation, we first prove a property which allows us to exclusively focus on the objects w and x. Roughly speaking, any solution can be partitioned into two sequences of swaps. In the first sequence, the object x_w, which agent I swaps in exchange for object x, is swapped between each two consecutive



$1 : x_3 \succ x_4 \succ \boxed{x_1}$ $2 : x_1 \succ x_4 \succ \boxed{x_2}$
$3 : x_2 \succ x_4 \succ \boxed{x_3}$ $4 : x_5 \succ x_3 \succ \boxed{x_4}$
$5 : x_6 \succ x_3 \succ \boxed{x_5}$ $6 : x_4 \succ x_3 \succ \boxed{x_6}$

Figure 7.2: An example of Reachable Object that is a slight modification of the example in Figure 7.1. Initially held objects are again drawn in boxes and the question is still whether x_3 is reachable for agent 1. Then, our algorithm finds the following swap sequence for object x_3 to reach agent 1: $4 \leftrightarrow 3$, $3 \leftrightarrow 2$, $2 \leftrightarrow 1$, $4 \leftrightarrow 5$, $5 \leftrightarrow 6$, $6 \leftrightarrow 1$, where "$i \leftrightarrow j$" means that agents i and j swap the objects they currently hold. In this example each agent in $\{1, 2, 3\}$ is an x_4-forwarder and each agent in $\{4, 5, 6\}$ is an x_3-forwarder.

assignments. In the second sequence, object x is swapped between each two consecutive assignments. More specifically, the following lemma states that the sequence of swaps resulting from performing all swaps that involve x_w and no other swaps leads to agent I obtaining x_w.

Lemma 7.2. *Let*

$$(V := \{1, 2, \dots, n\}, X := \{x_1, x_2, \dots, x_n\}, \mathcal{P}, \sigma_0, G := (V, E), I, x)$$

be an instance of Reachable Object *where $\sigma_0(i) = x_i$ for all $i \in [n]$ and where the* maximum length of preference lists *is at most three. Let $\phi := (\sigma_0, \sigma_1, \dots, \sigma_t)$ be a sequence of swaps such that $\sigma_t(I) = x$. Consider two objects x_p and x_q such that there is a swap τ_r with $\tau_r = \{(I, x_p), (j, x_q)\}$, that is, agent I obtains object x_q in exchange for x_p during ϕ. Let $T = \{\tau_i \mid \tau_i = \{(j, x'_p), (k, x_q) \wedge i \le r\}\}$ be the set of all swaps between assignments in ϕ up to assignment σ_r that involve swapping x_q. We denote the elements of T by $\tau'_1, \tau'_2, \dots, \tau'_{|T|}$ such that swap τ'_i occurs before swap τ'_j in ϕ for each $i < j$. Let $\phi_{\text{start}} := (\sigma'_0, \sigma'_1, \dots, \sigma'_s)$ be the sequence of assignments such that $\sigma'_0 := \sigma_0$ and σ'_i is the result of performing swap τ'_i in assignment τ'_{i-1}. Let $\tau'_i := \{(a_{i-1}, x_q), (a_i, x_{b_i})\}$ for each $i \in [s]$. Then,*

(i) $\tau'_s = \tau_r$,

(ii) $a_0 = q$ *and agent q prefers x_{b_1} over x_q,*

(iii) *$a_s = I$ and agent I prefers x_q over its initially held object x_I,*

(iv) *for each $z \in [s-1]$ agent a_z has preference list $x_{b_{z+1}} \succ x_q \succ \boxed{x_{b_z}}$ and is an x_q-forwarder,*

(v) *if agent I prefers x over x_q, then no agent a_z with $z \in [s-1]$ prefers object x over its initially held object, and*

(vi) *if agent I initially holds x_p, then ϕ_{start} is a sequence of swaps such that $\sigma'_s(I) = x_q$.*

Proof. We prove the individual statements one after another and we start with statement (i). To this end, note that by definition, the last swap τ_r between σ_{r-1} and σ_r contains (j, x_q) for some agent j. Since the swaps between consecutive assignments in ϕ' are exactly those that involve swapping object x_q, it follows that the swap between σ'_{s-1} and σ'_s must be τ_r and thus $\tau'_s = \tau_r$ and statement (i) holds.

We continue with statement (ii). By definition, $\tau'_1 := \{(a_0, x_q), (a_1, x_{b_1})\}$ and agent q initially holds object x_q. By definition of rational swaps it holds that agent a_0 initially holds object x_q and prefers x_{b_1} over x_q. Since each object is unique, sind each object is only held by one agent at a time, and since both agents a_0 and q initially hold x_q, it holds that $a_0 = q$ and thus statement (ii) holds.

To show statement (iii), recall that $\tau_r = \tau'_s := \{(a_{s-1}, x_q), (a_s, x_{b_1})\}$ and thus $a_s = I$ and since τ is a rational swap, agent a_s has to prefer x_q over x_p. Since agent a_s holds x_p during ϕ, it holds that $x_p = x_1$ or that agent a_s prefers x_p over x_I. In both cases it prefers x over x_I and thus statement (iii) holds.

We next prove statement (iv). Assume towards a contradiction that there is a minimum $z \in [s-1]$ such that a_z does not have preference list $x_{b_{z+1}} \succ x_q \succ \boxed{x_{b_z}}$ or that it is not an x_q-forwarder. Observe that if $z = 1$, then by statement (ii) $a_{z-1} = a_0 = q$ and agent a_{z-1} initially holds x_q and otherwise, since z is minimum, it holds that after swap τ'_{z-1} agent a_{z-1} holds object x_q. Since $\tau'_z = \tau_k$ for some k, it holds that $x_q \succ_{a_z} \boxed{x_{b_z}}$ and agent a_z swaps x_{b_z} for x_q away. By definition of τ'_{z+1}, agents a_z and a_{z+1} then swap x_q and $x_{b_{z+1}}$ and thus $x_{b_{z+1}} \succ_{a_z} x_q$ and a_z is an x_q-forwarder, a contradiction.

We next show statement (v). Note that if agent I prefers object x over x_q, then $x \neq x_q$. Assume towards a contradiction that some agent a_z with $z \in [s]$ prefers x over x_q. Note that in this case $x_q \neq x_{b_z}$ as x_{b_z} is the object that a_z

initially holds. Then by statement (iv), it holds that a_z only prefers x_q and $x_{b_{z+1}}$ over x_{b_z} and that a_z obtains $x_{b_{z+1}}$ during ϕ before agent I obtains object x. Since $x_q \neq x$, it holds that $x = x_{b_{z+1}}$ and since a_z holds its most preferred object x, it will never trade this object away. Thus, object x cannot be obtained by agent I during ϕ, a contradiction.

Finally, to show statement (vi), notice that by statement (i) and the definition of τ, it holds that $\sigma'_s(I) = x_q$ and hence it remains to show that ϕ_{start} is a sequence of swaps. Assume towards a contradiction that ϕ_{start} is not a sequence of swaps, that is, there are two consecutive assignments σ'_{i-1} and σ'_i such that τ'_i is not a rational swap. Since by definition $\tau'_i = \tau_k$ for some k, it holds that if $\tau'_i := \{(a_{i-1}, x_q), (a_i, x_{b_i})\}$ is possible in the sense that a_{i-1} holds x_q and agent a_i holds x_{b_i}, then τ'_i is a rational swap. Assume without loss of generality that τ'_i is the first swap between consecutive assignments in ϕ_{start} that is not possible. Then, $i = 1$ or τ'_{i-1} is possible. By statement (ii), in both cases agent a_{i-1} holds x_q in σ'_i. If $i < s$, then by statement (iv) agent a_i can only trade x_{b_i} away in order to obtain x_q in any trade τ_k between two assignments σ_{k-1} and σ_k in ϕ. Since $\tau'_i = \tau_k$ for some $k \in [r]$, it holds that τ'_i is possible, a contradiction. If $i = s$, then by statement (iii) agent a_s is agent I which initially holds by assumption $x_p = x_{b_i}$ and hence τ'_i is possible, a contradiction. □

We next present the main algorithm of this subsection and prove that it solves Reachable Object when the maximum length of preference lists is at most three. Pseudo-code is given in Algorithm 7.1. The idea therein is to model possible swaps that involve x as arcs in a directed graph. Each arc (i, j) in this graph represents the fact that if agent i obtains object x, then it can swap it to agent j in exchange for object x_j. A directed path from the agent that initially holds x to agent I then corresponds to a sequence of swaps such that agent I obtains object x in the end. We then consider the third object $x_w \notin \{x_I, x\}$ which appears in the preference list of I and build a similar directed graph for x_w. The directed paths from agent w to agent I in it again correspond to sequences of swaps such that agent I obtains object x_w.

Proposition 7.3. *Reachable Object can be solved in linear time when the maximum length of preference lists is at most three.*

Proof. Let

$$(V := \{1, 2, \ldots, n\}, X := \{x_1, x_2, \ldots, x_n\}, \mathcal{P}, \sigma_0, G := (V, E), I, x)$$

Algorithm 7.1: Algorithm for Reachable Object when the maximum length of preference lists is at most three.

```
Input  : A set $V$ of agents, preference lists $(\succ_i)_{i \in V}$ of length at most
         three, and a graph $(V, E)$.
Output : true if agent $I$ can receive object $x$ that is initially held by
         agent $a$ and false otherwise.
1  $F \leftarrow \{(i,j) \mid \{i,j\} \in E \wedge x_j \succ_i x \wedge x \succ_j x_j\}$
             // If agent $i$ obtains $x$, then it can swap it to agent $j$ for $x_j$.
2  $D \leftarrow (V, F)$
3  if $D$ admits a directed path $P$ from $a$ to $I$ then return yes
4  if $x \succ_I x_w \succ_I \boxed{x_I}$ for some $x_w \neq x$ then
5      $F_1 \leftarrow \{(i,j) \mid \{i,j\} \in E \wedge x_j \succ_i x_w \wedge x_w \succ_j x_j\}$
6      $F_2 \leftarrow \{(i,j) \mid j \neq I \wedge \{i,j\} \in E \wedge x_j \succ_i x \wedge x \succ_j x_j\}$
7      $F_3 \leftarrow \{(i,I) \mid \{i,I\} \in E \wedge x_w \succ_i x\}$
             // If $i$ obtains object $x$, then it swaps it to agent $I$ for $x_w$.
8      $D_1 \leftarrow (V, F_1)$
9      $D_2 \leftarrow (V, F_2 \cup F_3)$
10     if $D_1$ admits a directed path from $w$ to $I$ and $(w,a) \in F_2$ then
                    // Object $x$ is held by agent $w$ after the first swap
11         if $D_2$ admits a directed path from $w$ to $I$ then return true
12     if $D_1 - \{a\}$ admits a directed path from $w$ to $I$ then
13         if $D_2$ admits a directed path from $a$ to $I$ then return true
14 return false
```

be an instance of Reachable Object where $\sigma_0(i) = x_i$ for all $i \in [n]$ and where the maximum length of preference lists is at most three. Let a be the agent that initially holds object x. We use Algorithm 7.1 to prove this proposition. We start with showing that if object x is reachable for agent I, then Algorithm 7.1 returns true. To this end, assume that there exists a sequence $\phi = (\sigma_0, \sigma_1, \ldots, \sigma_t)$ of swaps such that $\sigma_t(I) = x$. We assume without loss of generality that $\sigma_{t-1}(I) := x_b \neq x$. We then distinguish between the two cases $x_b = x_I$ and $x_b \neq x_I$. If $x_b = x_I$, then using $x_p = x_I$ and $x_q = x$, the sequence $\phi' = (\sigma'_0, \sigma'_1, \ldots, \sigma'_s)$ as defined in Lemma 7.2 is a sequence of swaps such that $\sigma'_0 = \sigma_0$ and $\sigma'_s(I) = x$. By Lemma 7.2(ii) to (iv), graph D as constructed in Line 2 must contain a path from a to I. Thus, Algorithm 7.1 returns true in Line 3.

If $x_b \neq x_I$, then the preference list of agent I is $x_n \succ x_w \succ \boxed{x_I}$ and $x_b = x_w$. Moreover, agent I obtains x_w during ϕ and thus there are σ_{r-1} and σ_r such that $\tau_r = \{\{I, x_I\}, \{k, x_w\}\}$ for some agent k. By Lemma 7.2 (using $x_p = x_I$ and $x_q = x_w$), the sequence $\phi' = (\sigma'_0, \sigma'_0, \ldots, \sigma'_s)$ as defined in Lemma 7.2 is a sequence of swaps such that $\sigma'_0 = \sigma_0$ and $\sigma'_s(I) = x_w$. Let $a_1, a_2, \ldots, a_s$ be the agents that hold object x_w during ϕ'. It follows from Lemma 7.2(iv) and the definition of D_1 in Line 8 that ϕ' defines a directed path $(a_0, a_1, a_2, \ldots, a_s)$ with $a_0 = w$ and $a_s = I$ in D_1. By Lemma 7.2(v), no agent in $\{a_2, a_3, \ldots, a_{s-1}\}$ prefers x over its initially held object. By Lemma 7.2(ii) and (iv), it holds that $\tau'_1 = \{\{w, x_w\}, \{a_1, x_{a_1}\}\}$. We then distinguish between the two cases $a_1 = a$ and $a_1 \neq a$. If $a_1 = a$, then none of the agents in $\{a_2, a_3, \ldots, a_{s-1}\}$ can be involved in a swap where x is traded. Observe that in this case considering the initial assignment σ''_0 with $\sigma''_0(I) = x_w$, $\sigma''_a = x_I$, $\sigma''_{a_1} = x$, and $\sigma''_0(i) = \sigma_0(i)$ for all other agents i is equivalent to the original instance. Using Lemma 7.2 with $x_p = x_w$ and $x_q = x$ then states that there is a sequence $\phi^* = (\sigma^*_0, \sigma^*_0, \ldots, \sigma^*_{s'})$ of swaps with $\sigma^*_0 = \sigma''_0$ and $\sigma^*_{s'}(I) = x$. It follows from Lemma 7.2(iv) and the definition of D_2 in Line 9 that ϕ^* defines a directed path $(b_0, b_1, b_2 \ldots, b_{s'})$ with $b_0 = w$ and $b_{s'} = I$ in D_3. Thus, Algorithm 7.1 returns true in Line 11.

If $a_1 \neq a$, then none of the agents in $\{a_0, a_1, \ldots, a_{s-1}\}$ can be involved in a swap where x is traded and agent a cannot receive x_w during ϕ. Hence, $D_1 - \{a\}$ contains a directed path from w to I and D_2 contains a directed path from a to I. Thus, Algorithm 7.1 returns true in Line 13.

We next show that if the algorithm returns true, then there exists a sequence $\phi = (\sigma_0, \sigma_1, \ldots, \sigma_t)$ of swaps such that $\sigma_t(I) = x$. If the algorithm returns true, it does so either in Line 3, in Line 11, or in Line 13. If the algorithm returns true in Line 3, then let $(a_0 := a, a_1, \ldots, a_t := I)$ be a directed path in D. Let σ_i be an assignment such that $\sigma_i(a_i) := \sigma_{i-1}(a_{i-1})$, $\sigma_i(a_{i-1}) := \sigma_{i-1}(a_i)$ and $\sigma_i(j) = \sigma_{i-1}(j)$ for all other agents j. By definition of rational trades and D, it holds that $(\sigma_0, \sigma_1, \ldots, \sigma_t)$ is a sequence of swaps. Note that by construction $\sigma_i(a_i) = x$ for all $i \in [t]$ and thus $\sigma_t(a_t) = \sigma_t(I) = x$.

If the algorithm returns true in Line 11, then let $(a_0 := w, a_1 := a, \ldots, a_s := I)$ be a directed path in D_1 and let $(b_0 := w, b_1, \ldots, b_t := I)$ be a directed path in D_2. Let σ_i be an assignment such that

1. $\sigma_i(a_i) := \sigma_{i-1}(a_{i-1})$ and $\sigma_i(a_{i-1}) := \sigma_{i-1}(a_i)$ for all $i \in [s]$,

2. $\sigma_{s+i}(b_i) \coloneqq \sigma_{s+i-1}(b_{i-1})$ and $\sigma_{s+i}(b_{i-1}) \coloneqq \sigma_{s+i-1}(b_i)$ for all $i \in [t]$, and
3. $\sigma_i(j) = \sigma_{i-1}(j)$ for all $i \in [s+t]$ and all agents j that are not assigned objects by the above.

By definition of rational trades and D_1 and D_2, it holds that $(\sigma_0, \sigma_1, \dots, \sigma_{s+t})$ is a sequence of swaps. Note that by construction $\sigma_{s+i}(b_i) = x$ for all $i \in [t]$ and thus $\sigma_{s+t}(b_t) = \sigma_{s+t}(I) = x$.

Finally, the case where the algorithm returns true in Line 13 is completely analogous for the two directed paths

$$(a_0 \coloneqq w, a_1, \dots, a_s \coloneqq I) \text{ and } (b_0 \coloneqq a, b_1, \dots, b_t \coloneqq I).$$

It remains to analyze the running time. We start with constructing $D \coloneqq (V, F)$ in $O(n+m)$ time. The constructions of D_1 and D_2 are analogous. To construct $D = (V, F)$, we go through each edge $\{i, j\}$ in the input graph and check in constant time whether agent i prefers x_j over x and whether agent j prefers x over x_j. All remaining steps are searches for directed paths in graphs. Using dynamic programming on the topological orders of the constructed DAGs, each of these steps can be computed in $O(n+m)$ time. Thus, the overall running time is $O(n+m)$. $\square$

7.2.2 Maximum Length at Most Four

Complementing the result from the previous subsection, we next show that Reachable Object is already *NP*-hard when the maximum length of preference lists is four even if the input graph is restricted to be a clique. The hardness of cliques implies that the computational hardness of the problem does not stem from restricting the possible swaps between agents by an underlying social network. To show *NP*-hardness, we reduce from a restricted variant of 3-SAT. In this variant, called 2P1N-SAT (two-positive-and-one-negative SAT), each clause has either two or three literals, and each variable appears once as a negative literal and either once or twice as a positive literal. 2P1N-SAT is known to be *NP*-complete [Tov84].

We start with some intuition. Let

$$\Phi \coloneqq (\mathcal{V} \coloneqq \{v_1, \dots, v_n\}, \mathcal{C} \coloneqq \{C_1, \dots, C_m\})$$

be an instance of 2P1N-SAT. The general idea of the reduction is to have a set of agents for each variable and one agent for each literal in a clause in Φ.

The agents representing variables can then pass objects to agents representing an occurrence of this variable in a clause such that

1. only agents representing positive literals or only the agent that represents the negative literal of this variable can receive these objects,
2. an agent representing a certain literal in a clause can pass the target object x to an agent in "the next clause" if and only if it received a respective object from one of the agents representing the respective variable, and
3. the target object x can reach the target agent I only if it passes through all clauses.

Before we formally describe our construction, we first introduce some notation. For each variable $v_i \in \mathcal{V}$, let $\mathsf{occ}(i)$ be the number of occurrences of variable v_i (note that $\mathsf{occ}(i) \in \{2, 3\}$), let $\nu(i)$ denote the index of the clause that contains the negative literal $\neg\, v_i$, and let $\pi_1(i)$ and $\pi_2(i)$ with $\pi_1(i) < \pi_2(i)$ be the indices of the clauses that contain the positive literal v_i. If $\mathsf{occ}(v_i) = 2$, then we simply neglect $\pi_2(i)$. For a clause C_j, we denote by $|C_j|$ the number of literals that C_j contains. For each clause $C_j \in \mathcal{C}$, we use an arbitrary but fixed order of the literals in C_j to define a bijective function $f_j : C_j \to \{1, \ldots, |C_j|\}$, which assigns to each literal contained in C_j a distinct number from $\{1, 2, \ldots, |C_j|\}$.

We next give the formal description of the construction of $\mathcal{I}$. Afterwards we show how these definitions match the intuition we gave earlier and show an example of the construction. Afterwards, we formally prove the correctness of the construction.

Construction 7.4. Let $\Phi = (\mathcal{V} := \{v_1, \ldots, v_n\}, \mathcal{C} := \{C_1, \ldots, C_m\})$ be an instance of 2P1N-SAT. We construct an instance of Reachable Object as follows.

Agents and objects. For each variable $v_i \in \mathcal{V}$, we define $\mathsf{occ}(i) - 1$ *variable agents* U_i^1 (and U_i^2 if $\mathsf{occ}(i) = 3$) and $\mathsf{occ}(i) - 1$ objects x_i^1 (and x_i^2 if $\mathsf{occ}(i) = 3$). For each clause $C_j \in \mathcal{C}$, we define $2|C_j| + 1$ *clause agents* A_j, B_j^z, and D_j^z, where $z \in [|C_j|]$. Moreover, we define $2|C_j| + 1$ objects

$$a_j, b_j^1, b_j^2, \ldots, b_j^{|C_j|}, d_j^1, d_j^2, \ldots, d_j^{|C_j|}.$$

Finally, there is a special agent I and a special object x.

Initial assignment and graph. For each $i \in [n]$ and each $z \in [\mathsf{occ}(i)]$ agent U_i^z initially holds object $x_i^{\mathsf{occ}(i)-z}$. For each $j \in [m]$ and each $z \in [|C_j|]$ agent B_j^z initially holds object b_i^z and agent D_j^z initially holds object d_i^z. Finally, for each $j \in [m-1]$, agent A_{j+1} initially holds object a_j, agent a_1 initially holds x, and agent I initially holds a_m.

The graph $G := (V, E)$ is complete, that is, $E := \binom{V}{2}$.

Preference lists. We next describe the preference list of each agent. Therein, we only specify the relevant part, that is, the preference list up to the object that the agent initially holds. We again mark the initially held object with a box. For a given variable $v_i \in \mathcal{V}$, let $j = \nu(i)$, $j' = \pi_1(i)$, and if $\mathsf{occ}(i) = 3$, then $j'' = \pi_2(i)$. If $\mathsf{occ}(i) = 2$, then the preference list of U_i^1 is

$$d_{j'}^{f_{j'}(v_i)} \succ d_j^{f_j(\neg v_i)} \succ \boxed{x_i^1},$$

and if $\mathsf{occ}(i) = 3$, then the preference lists of U_i^1 and U_i^2 are

$$d_{j'}^{f_{j'}(v_i)} \succ x_i^1 \succ d_j^{f_j(\neg v_i)} \succ \boxed{x_i^2} \text{ and}$$
$$d_{j''}^{f_{j''}(v_i)} \succ x_i^2 \succ \boxed{x_i^1}, \text{ respectively.}$$

For $j \in [2, m]$, the preference list of A_j is

$$b_j^1 \succ \cdots \succ b_j^{|C_j|} \succ \boxed{a_{j-1}}.$$

Let for each $z \in [\mathsf{occ}(i)]$ be ℓ_z the index such that $f_j(\ell_z) = z$. The preference list of B_i^z is then

$$\tau(C_j, \ell_z) \succ x \succ a_{j-1} \succ \boxed{b_j^z},$$

where

$$\tau(C_j, \ell) := \begin{cases} x_i^1, & \text{if } \mathsf{occ}(i) = 2 \text{ and } \ell = \neg\, v_i \text{ for some variable } v_i, \\ x_i^2, & \text{if } \mathsf{occ}(i) = 3 \text{ and } \ell = \neg\, v_i \text{ for some variable } v_i, \\ x_i^1, & \text{if } \ell = v_i \text{ and } j = \pi_1(i) \text{ for some variable } v_i, \text{ and} \\ x_i^2, & \text{if } \ell = v_i \text{ and } j = \pi_2(i) \text{ for some variable } v_i. \end{cases}$$

The preference list of D_j^z is

$$a_j \succ x \succ \tau(C_j, \ell_z) \succ \boxed{d_j^z}.$$

The preference list of A_1 is

$$b_1^1 \succ b_1^2 \succ \cdots \succ b_1^{|C_1|} \succ \boxed{x}.$$

For each $z \in [\mathsf{occ}(i)]$ let ℓ_z be the index such that $f_1(\ell_z) = z$. The preference lists of B_1^z and D_1^z are

$$\tau(C_1, \ell_z) \succ x \succ \boxed{b_1^z} \text{ and}$$
$$a_1 \succ x \succ \tau(C_1, \ell_z) \succ \boxed{d_1^z}, \text{ respectively.}$$

Finally, the preference list of agent I is $x \succ \boxed{a_m}$.

We next explain how these formal definitions follow the general idea we started with. To this end, note that if for two agents i and j there are no two objects x_k and x_ℓ such that $x_k \succ_i x_\ell \succcurlyeq_i x_i$ and $x_\ell \succ_j x_k \succcurlyeq_j x_j$, then by definition of rational trades agents i and j will never swap objects. In this case, we say that the edge $\{i, j\}$ is *irrelevant* and ignore the edges henceforth. All other edges are *relevant* and by carefully examining the preference lists of all agents, there are only the following relevant edges.

(1) Relevant edges between clause agents representing one clause C_j are for each $z \in [|C_j|]$
$$\{A_j, B_j^z\}, \{B_j^z, D_j^z\},$$
that is, all clause agents for one clause form a subdivided star.

(2) Relevant edges between clause agents representing two consecutive clauses C_j and C_{j+1} are for each $z \in [|C_j|]$ and $z' \in [|C_{j+1}|]$
$$\{D_j^z, B_{j+1}^{z'}\},$$
that is, the two vertex sets $\{D_j^z \mid 1 \leq z \leq |C_j|\}$ and $\{B_{j+1}^{z'} \mid 1 \leq z' \leq |C_{j+1}|\}$ form a complete bipartite graph.

(3) Agent I is adjacent to all clause agents D_m^z for $z \in [|C_m|]\}$.

(4) There are no relevant edges between variable agents except between two agents U_i^1 and U_i^2 that represent the same variable v_i with $\mathsf{occ}(i) = 3$.

(5) Finally, for edges between the variable agent and clause agents, we distinguish for each variable $v_i \in \mathcal{V}$ between $\mathsf{occ}(i) = 2$ or $\mathsf{occ}(i) = 3$.

Table 7.1: The preference lists of all agents for the instance $\mathcal{V} := \{v_1, v_2, v_3\}$ and $\mathcal{C} := \{C_1 := (v_2 \vee v_3), C_2 := (v_1 \vee \neg\, v_2 \vee \neg\, v_3), C_3 := (\neg\, v_1 \vee v_2 \vee v_3)\}$.

$A_1 \colon b_1^1 \succ b_1^2 \succ \boxed{x}$	$A_2 \colon b_2^1 \succ b_2^2 \succ b_2^3 \succ \boxed{a_1}$	$A_3 \colon b_3^1 \succ b_3^2 \succ b_3^3 \succ \boxed{a_2}$
$B_1^1 \colon x_2^1 \succ x \succ \boxed{b_1^1}$	$B_2^1 \colon x_1^1 \succ x \succ a_1 \succ \boxed{b_2^1}$	$B_3^1 \colon x_1^1 \succ x \succ a_2 \succ \boxed{b_3^1}$
$B_1^2 \colon x_3^1 \succ x \succ \boxed{b_1^2}$	$B_2^2 \colon x_2^2 \succ x \succ a_1 \succ \boxed{b_2^2}$	$B_3^2 \colon x_2^2 \succ x \succ a_2 \succ \boxed{b_3^2}$
	$B_2^3 \colon x_3^2 \succ x \succ a_1 \succ \boxed{b_2^3}$	$B_3^3 \colon x_3^2 \succ x \succ a_2 \succ \boxed{b_3^3}$
$D_1^1 \colon a_1 \succ x \succ x_2^1 \succ \boxed{d_1^1}$	$D_2^1 \colon a_2 \succ x \succ x_1^1 \succ \boxed{d_2^1}$	$D_3^1 \colon a_3 \succ x \succ x_1^1 \succ \boxed{d_3^1}$
$D_1^2 \colon a_1 \succ x \succ x_3^1 \succ \boxed{d_1^2}$	$D_2^2 \colon a_2 \succ x \succ x_2^2 \succ \boxed{d_2^2}$	$D_3^2 \colon a_3 \succ x \succ x_2^2 \succ \boxed{d_3^2}$
	$D_2^3 \colon a_2 \succ x \succ x_3^2 \succ \boxed{d_2^3}$	$D_3^3 \colon a_3 \succ x \succ x_3^2 \succ \boxed{d_3^3}$
$I \colon x \succ \boxed{a_3}$		
$U_1^1 \colon d_2^1 \succ d_3^1 \succ \boxed{x_1^1}$	$U_2^1 \colon d_1^1 \succ x_2^1 \succ d_2^2 \succ \boxed{x_2^2}$	$U_3^1 \colon d_1^2 \succ x_3^1 \succ d_2^3 \succ \boxed{x_3^2}$
	$U_2^2 \colon d_3^2 \succ x_2^2 \succ \boxed{x_2^1}$	$U_3^2 \colon d_3^3 \succ x_3^2 \succ \boxed{x_3^1}$

(a) If $\mathsf{occ}(i) = 2$, then the relevant edges between the variable agent U_i^1 representing v_i and clause agents are

$$\{U_i^1, B_{\pi_1(i)}^{f_{\pi_1(i)}(v_i)}\} \text{ and } \{U_i^1, B_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}\}.$$

(b) If $\mathsf{occ}(i) = 3$, then the relevant edges between the variable agents U_i^1 and U_i^2 representing v_i and clause agents are

$$\{U_i^1, B_{\pi_1(i)}^{f_{\pi_1(i)}(v_i)}\},\ \{U_i^1, B_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}\},\ \text{and } \{U_i^2, B_{\pi_2(i)}^{f_{\pi_2(i)}(v_i)}\}.$$

We now briefly describe how a solution in the constructed instance corresponds to a satisfying truth assignment of the original formula. Afterwards, we present the formal proof. Consider Table 7.1 and Figure 7.3 for an example of Construction 7.4, relevant edges, and how a satisfying truth assignment to the original formula corresponds to a solution for the constructed Reachable Object instance. Note that only agents B_j^z and D_j^z for $j \in [m]$ and $z \in [|C_j|]$ as well as agents I and A_1 prefer object x at least as much as their initially held object. By the analysis of relevant edges above, one can easily verify that agent I can only receive object x if for each clause C_j at least one agent B_j^z

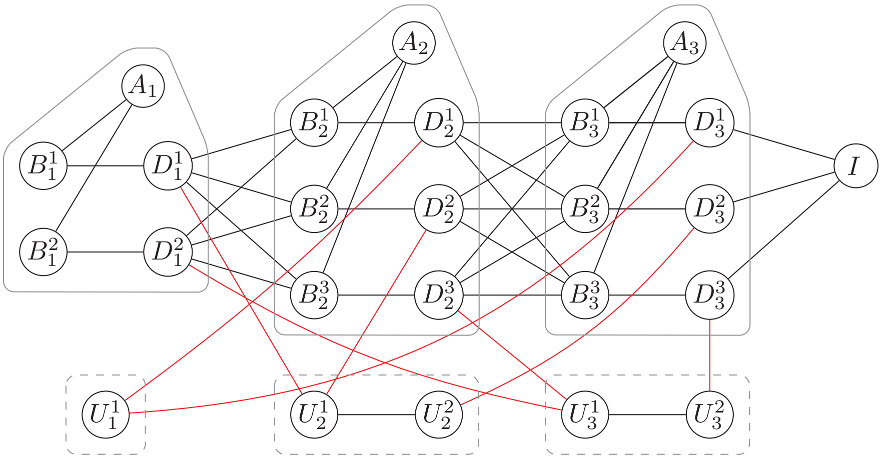


Figure 7.3: An example of the agents and relevant edges resulting from Construction 7.4 for the instance $\mathcal{V} := \{v_1, v_2, v_3\}$ and $\mathcal{C} := \{C_1 := (v_2 \vee v_3), C_2 := (v_1 \vee \neg\, v_2 \vee \neg\, v_3), C_3 := (\neg\, v_1 \vee v_2 \vee v_3)\}$. The boxes with solid lines indicate the three clause gadgets and the three boxes with dashed lines display the three variable gadgets. Relevant edges between variable agents and clause agents are only drawn red for easier distinction. The preference lists are listed in Table 7.1. Notice that setting v_1 and v_2 to false and v_3 to true is a satisfying truth assignment. This corresponds to the following sequence of swaps that lets agent I obtain object x. Setting a variable that occurs thrice to true (v_3 in our example) is represented by the swap of the initially held objects of the two respective variable agents (in our case U_3^1 and U_3^2 swap x_3^2 and x_3^1). Afterwards we decide for each clause for one literal to satisfy this clause. Since in our example C_1 and C_2 are only satisfied by one literal, we can only choose between $\neg\, v_1$ and v_3 in C_3. Let us choose v_3 in C_3. Next, for each clause C_j the agent A_j swaps with the chosen B-vertex and the chosen D-vertex swaps with the respective variable agent. In our case, A_1 swaps with B_1^2, A_2 swaps with B_2^2, A_3 swaps with B_3^3, D_1^2 swaps with U_3^1, D_2^2 swaps with U_2^1, and D_3^3 swaps with U_3^2. If all clauses are satisfied, then object x can be swapped "through all clauses". In the sequence of swaps we described, object x it is held by agents A_1, B_1^2, D_1^2, B_2^2, D_2^2, B_3^3, D_3^3, and I.

and $D_j^{z'}$ for $z, z' \in [|C_j|]$ held object x before. For agent $D_j^{z'}$ to pass object x to agent B_{j+1}^z, it has to receive a_j in return. Since this object is initially held by agent A_{j+1} and since this agent only shares relevant edges with agents B_{j+1}^z

for $z \in [|C_{j+1}|]$, this works as a selection gadget of which literal in clause C_{j+1} should be satisfied. For agent B^z_{j+1} to then trade object x to agent D^z_{j+1}, it has to receive an object representing the corresponding variable in return. For D^z_{j+1} to receive such an object, it has to receive this from an agent in the corresponding variable gadget. If this variable only occurs twice, then the respective object can only be given to one D-agent and this agent will never give it away as it is its most preferred object. If the variable occurs thrice, then the agents preference lists are constructed in a way that if either of the "positive occurrences" are given an object representing this variable, then the "negative occurrence" cannot be satisfied.

It remains to formally prove that Construction 7.4 is correct. This leads to the main result of this subsection which states that Reachable Object remains *NP*-hard when each preference list has length at most four and which complements Proposition 7.3.

Proposition 7.5. *Reachable Object is NP-hard even if the maximum length of preference lists is four and the input graph is restricted to complete graphs.*

Proof. Since each step in Construction 7.4 is polynomial-time computable, since all preference lists have by construction length at most four, and since the graph is complete, we will focus on showing that the constructed instance is equivalent to the original instance. To this end, let $\Phi := (\mathcal{V}, \mathcal{C})$ be an instance of 2P1N-SAT and consider the instance of Reachable Object resulting from Construction 7.4.

We will first show that if Φ is satisfiable, then there is a sequence of swaps such that object x reaches I. Let $\beta\colon \mathcal{V} \to \{\text{true}, \text{false}\}$ be a satisfying truth assignment for Φ. First, for each variable $v_i \in \mathcal{V}$, if $\mathsf{occ}(i) = 3$ and $\beta(v_i) = \text{true}$, then let agents U^1_i and U^2_i swap their initially held objects (so that U^1_i and U^2_i hold x^1_i and x^2_i, respectively). Second, identify for each clause C_j one literal ℓ_j that satisfies C_j under assignment β. Then, perform the following swaps.

1. Let agents A_j and $B^{f_j(\ell_j)}_j$ swap their initially held objects.
2. Let agents $D^{f_j(\ell_j)}_j$ and U^z_i swap their current objects such that
   (a) if $\ell_j = \neg\, v_i$, then $z = 1$ (note that in this case agent U^1_i is holding object $x^{\mathsf{occ}(i)-1}_i$),
   (b) if $\ell_j = v_i$ and $j = \pi_1(i)$, then $z = 1$ (note that in this case agent U^1_i is holding object x^1_i), and

(c) if $\ell_j = v_i$ and $j = \pi_2(i)$, then $z = 2$ (note that in this case agent U_i^2 is holding object x_i^2).

After these swaps, agent $B_1^{f_1(\ell_1)}$ holds object x and agent $B_j^{f_j(\ell_j)}$ holds object a_{j-1} for each $j \in [2, m]$. Moreover, for each $j \in [m]$, agent $D_j^{f_j(\ell_j)}$ holds object $\tau(C_j, \ell_j)$. Third, for each $j \in [m-1]$ iteratively perform the following swaps.

1. Agents $B_j^{f_j(\ell_j)}$ and $D_j^{f_j(\ell_j)}$ swap their current objects (so that $D_j^{f_j(\ell_j)}$ holds object x afterwards).

2. Agents $D_j^{f_j(\ell_j)}$ and $B_{j+1}^{f_{j+1}(\ell_{j+1})}$ swap their current objects (agent $B_{j+1}^{f_{j+1}(\ell_{j+1})}$ holds object x afterwards).

After these swaps, agent $B_m^{f_m(\ell_m)}$ holds object x. Finally, agent $B_m^{f_m(\ell_m)}$ can swap object x in exchange for object $\tau(C_m, \ell_m)$ with agent $D_m^{f_m(\ell_m)}$ who can then swap x in exchange for object a_m with agent I. Thus, object x is reachable for agent I and the constructed instance is a yes-instance.

For the other direction, assume that there is a sequence $(\sigma_0, \sigma_1, \ldots, \sigma_s)$ of swaps such that $\sigma_s(I) = x$. We show how to construct a satisfying truth assignment for Φ using the following claim that formalizes the idea that object x has to pass "through all clauses".

Claim 7.6. *For each clause $C_j \in \mathcal{C}$, there exist assignments σ_r and σ_{r+1} and a literal $\ell_j \in C_j$ such that*

1. $\sigma_r(B_j^{f_j(\ell_j)}) = x$,

2. $\sigma_r(D_j^{f_j(\ell_j)}) = \tau(C_j, \ell_j)$,

3. $\sigma_{r+1}(B_j^{f_j(\ell_j)}) = \tau(C_j, \ell_j)$, *and*

4. $\sigma_{r+1}(D_j^{f_j(\ell_j)}) = x$.

Proof of Claim 7.6. We prove the claim by induction over j, starting with $j = m$. In the initial assignment σ_0, agent I holds object a_m. Note that I prefers only x over its initially held object a_m and only the agents D_m^z with $z \in [|C_m|]$ prefer x over a_m. Hence, I has to swap with one of these agents to obtain x. Let ℓ_m be the literal with $f_m(\ell_m) = z$. In order for agent D_m^z to obtain object x to swap it to I, it must hold object $\tau(C_m, \ell_m)$ and swap it for x since no agent will trade x for d^z_m. Observe that agent B^z_m is the only agent that prefers $\tau(C_m, \ell_m)$ over x and B^z_m must therefore trade object x to D^z_m in exchange for object $\tau(C_m, \ell_m)$. Thus, there are assignments σ_r and σ_{r+1} such that

1. $\sigma_r(B^z_m) = x$,
2. $\sigma_r(D^z_m) = \tau(C_m, \ell_m)$,
3. $\sigma_{r+1}(B^z_m) = \tau(C_m, \ell_m)$, and
4. $\sigma_{r+1}(D^z_m) = x$.

We now show that if for some $j \in [m-1]$ there are assignments $\sigma_{r'}$ and $\sigma_{r'+1}$ and a literal ℓ_{j+1} that fulfill the claim for clause C_{j+1}, then there are also assignments σ_r and σ_{r+1} and a literal ℓ_j that fulfill the claim for clause C_j. By definition, agent $B^{f_{j+1}(\ell_{j+1})}_{j+1}$ must have obtained object x at some point. Since it prefers x only over objects a_j and $b^{f_{j+1}(\ell_{j+1})}_{j+1}$ and since no agent prefers $b^{f_{j+1}(\ell_{j+1})}_{j+1}$ over x, it follows that agent $B^{f_{j+1}(\ell_{j+1})}_{j+1}$ must have swapped object a_j with some other agent for x. Since only agents from $\{D^z_j \mid z \in [|C_j|]\}$ prefer a_j over x, it follows that $B^{f_{j+1}(\ell_{j+1})}_{j+1}$ must have swapped with some agent $D^{z'}_j$ with $z' \in [|C_j|]$ to obtain object x. Let ℓ_j be the literal such that $f_j(\ell_j) = z'$. Now consider how agent $D^{z'}_j$ can obtain object x. Similarly to the case with agent D^z_m, agent $D^{z'}_j$ must swap object $\tau(C_j, \ell_j)$ with agent $B^{z'}_j$ to obtain x as no agent prefers $d^{z'}_j$ over x. Thus, there are assignments σ_r and σ_{r+1} such that

1. $\sigma_r(B^{f_j(\ell_j)}_j) = x$,
2. $\sigma_r(D^{f_j(\ell_j)}_j) = \tau(C_j, \ell_j)$,
3. $\sigma_{r+1}(B^{f_j(\ell_j)}_j) = \tau(C_j, \ell_j)$, and
4. $\sigma_{r+1}(D^{f_j(\ell_j)}_j) = x$. $\diamond$

We conclude the proof by constructing a truth assignment β and show that it satisfies Φ using Claim 7.6. Let for each variable $v_i \in \mathcal{V}$ be

$$\beta(v_i) := \begin{cases} \text{false}, & \text{if } D^{f_{\nu(i)}(\neg\, v_i)}_{\nu(i)} \text{ swapped object } x \text{ with } B^{f_{\nu(i)}(\neg\, v_i)}_{\nu(i)} \\ \text{true}, & \text{otherwise.} \end{cases}$$

Assume towards a contradiction that β does not satisfy Φ, that is, there is some clause $C_j \in \mathcal{C}$ that is not satisfied by β. By Claim 7.6, let $\ell_j \in C_j$ be a literal such that $D_j^{f_j(\ell_j)}$ swapped object x with $B_j^{f_j(\ell_j)}$ for object $\tau(C_j, \ell_j)$. Observe that $\ell_j \in \{v_i, \neg\, v_i\}$ for some $v_i \in \mathcal{V}$. We now distinguish between the two cases $\ell_j = v_i$ and $\ell_j = \neg\, v_i$. If $\ell_j = \neg\, v_i$, then notice that $j = \nu(i)$ since each variable occurs exactly once as a negative literal. Thus, $D_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}$ swapped object x with $B_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}$. By construction, $\beta(v_i) = \text{false}$ and thus C_j is satisfied, a contradiction.

If $\ell_j = v_i$, then $v_i \in C_j$. Since C_j is not satisfied by β, it holds that $\neg\, v_i \notin C_j$ and $\beta(v_i) = \text{false}$. It then follows from the construction of β that $D_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}$ swapped object x with $B_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}$. Note that, by the construction of the preference lists, $B_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}$ can only have given object $\tau(C_{\nu(i)}, \neg\, v_i)$ for x in this trade. We will show that this contradicts the assumption that $D_j^{f_j(\ell_j)}$ swapped object x with $B_j^{f_j(\ell_j)}$ for object $\tau(C_j, \ell_j)$ using a case distinction over $\mathsf{occ}(i)$.

If $\mathsf{occ}(i) = 2$, then the definition of τ yields $\tau(C_j, v_i) = \tau(C_{j'}, \neg\, v_i) = x_i^1$. Since both agents $B_j^{f_j(v_i)}$ and $B_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}$ prefer object x_i^1 the most, once one of the two agents received it, the same object cannot be used to be swapped to the respective other agent. Thus, not both of the constructed swaps can happen during the sequence of swaps, a contradiction.

If $\mathsf{occ}(i) = 3$, then by the definition of τ it holds that $\tau(C_{\nu(i)}, \neg\, v_i) = x_i^2$. Since agent $D_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}$ received object x_i^2, it must have received it from agent U_i^1 (as U_i^2 and $D_{\nu(i)}^{f_{\nu(i)}(\neg\, v_i)}$ share no relevant edge). Thus agents U_i^1 and U_i^2 did not swap their initially held objects. We make a final case distinction on whether $j = \pi_1(\ell_j)$ or $j = \pi_2(\ell_j)$. If $j = \pi_1(\ell_j)$, then agent $D_j^{f_j(\ell_j)}$ must have received object x_i^1 from agent U_i^1. If $j = \pi_2(\ell_j)$, then agent $D_j^{f_j(\ell_j)}$ must have received object x_i^2 from agent U_i^2. In both cases agents U_i^1 and U_i^2 swapped their initially held objects, a contradiction. ☐

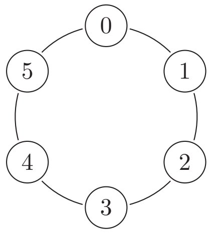
This concludes the dichotomy result for the maximum length ℓ of preference lists as Proposition 7.3 states that Reachable Object is linear-time solvable if $\ell \leq 3$ and Proposition 7.5 complements this result by showing that Reachable Object remains *NP*-hard for $\ell = 4$. Note that if we replace the complete graph in Construction 7.4 by the graph that only contains the relevant edges, then, for each $\ell > 4$, we can simply add a new agent with an arbitrary preference list of length ℓ that is only adjacent to agent I. This agent can never swap its initially held object o since I does not prefer o over its initially held object. This implies NP-hardness of Reachable Object with respect to the maximum length ℓ of preference lists for each $\ell > 4$.

7.3 Cycles

In this section, we prove that Reachable Object on n-vertex cycles is solvable in $O(n^4)$ time. This generalizes an $O(n^4)$-time algorithm for Reachable Object on paths by Huang and Xiao [HX20]. The main difference between our algorithm and the algorithm by Huang and Xiao is the fact that two objects can only be swapped once in a path but up to twice in a cycle. The main ingredient to overcome this obstacle is a structural observation which states that for each solution there is a constant c such that the following holds. For all pairs (x_i, x_j) of objects that are swapped twice in the solution, it holds that the two edges over which x_i and x_j are swapped have distance c in the input cycle. Since this constant c is the same for all pairs of objects, we will first determine the value of c and then use it to check for each pair of objects whether they can be swapped twice in a solution.

Note that we can ignore all connected components in the input instance of Reachable Object that do not contain I and hence we may assume that the input graph is connected. Note further that any connected graph with maximum degree two is either a path or a cycle. Thus, our algorithm for cycles and the algorithm for paths by Huang and Xiao [HX20] prove that Reachable Object is polynomial-time solvable for graphs of maximum degree two. Saffidine and Wilczynski [SW18] showed that Reachable Object remains NP-hard on graphs of maximum degree four. The general idea for our algorithm is as follows. Note that Observation 7.1 implies that there are only two possible paths of agents in the graph that can hold the target object x before the target agent I can obtain it. We will then guess[1] the path of agents that hold x during a solution (a sequence of swaps such that agent I obtains x). This will allow us to represent a solution by selecting one object to be swapped with x over each edge in the guessed path. An example of this is given in Figure 7.4. In Subsection 7.3.1, we show that for each edge in this path there are at most two

[1] As in Chapter 5, guessing refers to the procedure of iterating over all possibilities and considering “the correct” iteration for the proof.



$0 : x_5 \succ x_4 \succ x_3 \succ \boxed{x_0}$ $1 : x_3 \succ x_4 \succ x_5 \succ \boxed{x_1}$
$2 : x_5 \succ x_1 \succ x_4 \succ \boxed{x_2}$ $3 : x_2 \succ x_4 \succ \boxed{x_3}$
$4 : x_5 \succ x_3 \succ \boxed{x_4}$ $5 : x_0 \succ x_3 \succ \boxed{x_5}$

Figure 7.4: An example of Reachable Object on a cycle. Initially held objects are drawn in boxes and the question is whether x_4 is reachable for agent 0. Note that agent 5 does not accept object x_4 and hence object x_4 has to pass the edges $\{3,4\}$, $\{2,3\}$, $\{1,2\}$, and $\{0,1\}$ before agent 0 can obtain it. Considering the preference lists of the agents, it is easy to verify that only object x_3 can be swapped with x_4 over the edges $\{3,4\}$ and $\{0,1\}$. Analogously, only object x_2 can be swapped with x_4 over the edge $\{2,3\}$ and objects x_1 and x_5 are candidates for being swapped with x_4 over the edge $\{1,2\}$. Observe that for object x_5 to be swapped with x_4 over the edge $\{1,2\}$, it has to be swapped over the edge $\{0,1\}$ which is impossible as agent 0 does not prefer any object over x_5. Hence, the only solution selects objects x_1, x_2, and x_3 to move clockwise and objects x_0, x_4, and x_5 move counter-clockwise. Note that the sequence of swaps resulting from $3 \leftrightarrow 4$, $4 \leftrightarrow 5$, $5 \leftrightarrow 0$, $3 \leftrightarrow 2$, $2 \leftrightarrow 1$, $1 \leftrightarrow 0$ leads to agent I obtaining x_4. Therein, "$i \leftrightarrow j$" means that agents i and j swap the objects they currently hold.

candidate objects that can be swapped with x over the respective edge. Finally, in Subsection 7.3.2, we show how to partition the edges in the path such that

1. for each part of the partition there are at most two possible choices for selecting a candidate for all edges in the respective part and
2. candidates for two different parts are either incompatible, that is, there is no solution for the overall problem that uses the respective candidates, or they can be combined independently of the choices of candidates for other parts.

We conclude with the main theorem that states that Reachable Object on cycles can be solved in $O(n^4)$ time. The respective algorithm is a 2-SAT program with a variable for each part of the described partition. The truth value of this variable represents the choice of candidates for each edge in the respective part. The clauses will guarantee that no two incompatible candidates are chosen.

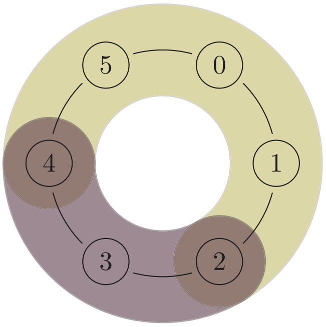


Figure 7.5: A cycle with six vertices. The part $[\![2,4]\!]$ is colored violet (darker) and $[\![4,2]\!]$ is colored yellow (brighter).

We start with some notation for this section. For the sake of readability, we assume that the graph is

$$G := (V := \{0\} \cup [n-1], E := \{\{i-1,i\} \mid i \in [n-1]\} \cup \{\{0,n-1\}\}).$$

Furthermore, if we refer to some agent j with $j \notin \{0\} \cup [n-1]$, then we mean agent j' with $j' \equiv j \pmod n$. For each object x_i, we denote by $\mathrm{A}(x_i)$ the agent that initially holds x_i, that is, $\sigma_0^{-1}(x_i)$. We assume without loss of generality that $I := 0$ and refer to the target object as x and define $k := \mathrm{A}(x)$.

We use $[\![i,j]\!]$ to denote the set $\{i, i+1 \bmod n, \dots, j \bmod n\}$, that is,

$$[\![i,j]\!] := \begin{cases} [i,j], & \text{if } j \geq i \text{, and} \\ [0,j] \cup [i,n-1] & \text{if } j < i. \end{cases}$$

See Figure 7.5 for an example. Finally, we say that an object x_i *moves clockwise* if it is swapped from some agent i to agent $i+1$. Analogously, we say that x_i *moves counter-clockwise* if it swapped from some agent i to agent $i-1$. By Observation 7.1, an object moving clockwise (or counter-clockwise) once, will only move clockwise (or counter-clockwise) in the future.

Observation 7.7. *Let ϕ be a sequence of swaps and let x_i be an object. If x_i is swapped during ϕ, then it either only moves clockwise or only moves counter-clockwise during ϕ.*

Note that the object x has to move clockwise or counter-clockwise in a solution. Our main algorithm just tries out both possibilities one after another and since these two cases work analogously, we will only present the case where x moves counter-clockwise here. Since x moves counter-clockwise and is initially held by agent $k := \mathrm{A}(x)$, if there is a solution (a sequence of swaps such that agent I obtains object x), then x is swapped over each edge in $\{\{i-1, i\} \mid i \in [k]\}$. Moreover, we can assume that x is swapped over the edge $\{0, 1\}$ in the last swap of the solution as all swaps afterwards are irrelevant. Our algorithm guesses the object z with which object x is swapped in this last swap. Note that there are two possibilities for x moving clockwise or counter-clockwise and at most n possibilities for choosing z. Hence, there are $O(n)$ iterations in our main algorithm and we can assume that x moves counter-clockwise and that object z is known. We will use this assumption throughout this section.

Assumption 7.8. *Let $\mathcal{I} = (V, X, \mathcal{P}, \sigma_0, G, 0, x)$ with*

$$G := (V := \{0\} \cup [n-1], E := \{\{i-1, i\} \mid i \in [n-1]\} \cup \{1, n-1\}))$$

be an instance of Reachable Object *on cycles. If x is reachable for agent 0, then there is a solution in which x moves counter-clockwise and in the last swap of the solution it is swapped with object z over the edge $\{0, 1\}$.*

We continue with an analysis of how often objects can be swapped in a cycle. To this end, we first show a helpful lemma which states for two objects x_i and x_j that are swapped in a sequence of swaps which other objects are swapped with either of them before x_i and x_j can be swapped.

Lemma 7.9. *Let x_h, x_i, and x_j be three distinct objects. Let $\phi = (\sigma_0, \sigma_1, \ldots, \sigma_t)$ be a sequence of swaps such that x_i and x_j are swapped between σ_{t-1} and σ_t and $x_i \neq x$ moves clockwise in ϕ. Let $r < t-1$ such that x_i and x_j are not swapped between σ_{s-1} and σ_s for any $s \in [r+1, t-1]$. Then, object x_h is swapped with either x_i or x_j between σ_{s-1} and σ_s for some $s \in [r+1, t-1]$ if and only if $\sigma_r^{-1}(x_h) \in [\![\sigma_r^{-1}(x_i), \sigma_r^{-1}(x_j)]\!]$.*

Proof. Note that since x_i and x_j are swapped in ϕ and since x_i moves clockwise, it holds that x_j moves counter-clockwise in ϕ. We prove the claim by induction over $|[\![\sigma_r^{-1}(x_i), \sigma_r^{-1}(x_j)]\!]|$. If $|[\![\sigma_r^{-1}(x_i), \sigma_r^{-1}(x_j)]\!]| = 2$, then x_i and x_j can only be swapped over the edge $\{\sigma_r^{-1}(x_i), \sigma_r^{-1}(x_j)\}$ and hence no other object can be swapped with either object before x_i and x_j are swapped. Since no object x_h

other than x_i and x_j fulfills $\sigma_r^{-1}(x_h) \in [\![\sigma_r^{-1}(x_i), \sigma_r^{-1}(x_j)]\!]$, this concludes the base case.

Now assume the statement holds for all objects $x_{i'}$ and $x_{j'}$ such that $x_{i'}$ moves clockwise, $x_{j'}$ moves counter-clockwise, and $|[\![\mathrm{A}(x_{i'}), \mathrm{A}(x_{j'})]\!]| < |[\![\mathrm{A}(x_i), \mathrm{A}(x_j)]\!]|$. Take any object x_ℓ such that $\mathrm{A}(x_\ell) \in [\![\mathrm{A}(x_i), \mathrm{A}(x_j)]\!] \setminus \{x_i, x_j\}$. We assume without loss of generality that x_ℓ moves counter-clockwise in ϕ as the other case is analogous. Note that x_i and x_ℓ are swapped in ϕ as otherwise x_ℓ would always stay "between" x_i and x_j and hence x_i and x_j could not be swapped in ϕ. By induction hypothesis, if x_i and x_ℓ are swapped between σ_{s-1} and σ_s for some $s \geq t$, then x_i and x_j are not swapped between σ_{t-1} and σ_t, a contradiction. Hence x_i and x_ℓ are swapped before x_i and x_j are swapped, that is, there is some $s \in [r+1, t-1]$ such that x_i and x_h are swapped between σ_{s-1} and σ_s.

It remains to show that no object x_h with $\mathrm{A}(x_h) \notin [\![\mathrm{A}(x_i), \mathrm{A}(x_j)]\!]$, is swapped with x_i or x_j before x_i and x_j are swapped. This follows from a simple counting argument. There are $|[\![\sigma_r^{-1}(x_i), \sigma_r^{-1}(x_j)]\!]| - 1$ edges between $\sigma_r^{-1}(x_i)$ and $\sigma_r^{-1}(x_j)$. The two objects x_i and x_j are swapped over one of these edges. Over each of the other edges exactly one of the objects is swapped before x_i and x_j are swapped. Thus, $|[\![\sigma_r^{-1}(x_i), \sigma_r^{-1}(x_j)]\!]| - 2$ objects are swapped with either x_i or x_j before x_i and x_j are swapped. As shown above, each agent x_h with $\mathrm{A}(x_h) \in [\![\sigma_r^{-1}(x_i), \sigma_r^{-1}(x_j)]\!]$ and $x_h \notin \{x_i, x_j\}$ is swapped with x_i or x_j before x_i and x_j are swapped. The counting argument is then completed by observing that there are $|[\![\sigma_r^{-1}(x_i), \sigma_r^{-1}(x_j)]\!]| - 2$ such objects. □

For an example of Lemma 7.9, recall Figure 7.4. Therein, object x_3 moves clockwise, object x_0 moves counter-clockwise, and objects x_4 and x_5 are initially held by agents in $[\![3, 0]\!]$. Lemma 7.9 states that objects x_4 and x_5 are swapped with x_0 or x_3 before objects x_0 and x_3 are swapped and objects x_1 and x_2 are not swapped with x_0 or x_3 before x_0 and x_3 are swapped. Lemma 7.9 has three interesting implications. First, it implies that each pair of objects is swapped at most twice. Note that the example in Figure 7.4 shows that two objects (x_3 and x_4 in the example) can be swapped twice in a cycle. To verify that each pair of objects is swapped at most twice, consider two objects x_i and x_j and the assignment σ_r after x_i and x_j are swapped for the first time over an edge $\{\ell, \ell+1 \bmod n\}$. Lemma 7.9 then states that each object x_h (except for x_i and x_j) have to be swapped with either x_i or x_j before x_i and x_j can be swapped for a second time. Thus, each agent has to hold x_i or x_j between the two swaps of x_i and x_j as for each of the $n-2$ objects that are swapped with x_i or x_j a new agent holds x_i or x_j. Since after the second swap of x_i and x_j agent $\mathrm{A}(x_i)$

has held x_i and x_j, it will by Assumption 7.8 and Observation 7.1 not accept either of the objects again, so x_i and x_j cannot be swapped thrice in a cycle.

Corollary 7.10. *Each pair of objects can be swapped at most twice in a cycle.*

The second interesting implication of Lemma 7.9 is that if $\mathrm{A}(z) \in [\![k, I]\!]$, then each object x_i with $\mathrm{A}(x_i) \in [\![k, \mathrm{A}(z)]\!]$ except for x and z is not swapped with x or z before x and z are swapped. Moreover, no such object can be swapped with another object that is then swapped with x or z. Notice that in this case agent I holds object z before x and z are swapped and thus x and z are by Observation 7.1 only swapped once. Hence, an object x_i with $\mathrm{A}(x_i) \in [\![k, \mathrm{A}(z)]\!]$ does not have to be swapped at all and thus no two objects have to be swapped over the edge $\{k, k+1\}$. We can therefore remove the edge from the cycle to obtain a path and use the algorithm by Huang and Xiao [HX20]. For the remainder of this section, we will therefore assume the following.

Assumption 7.11. $\mathrm{A}(z) \in [I+1, k]$

The third implication of Lemma 7.9 concerns how often an object is swapped with x or z. Note that each object moving clockwise is swapped with x (as all objects are swapped with x or z between the first and second swap of x and z and an object moving clockwise can never be swapped with z). Analogously, each object moving counter-clockwise is swapped with z. Thus, Lemma 7.9 implies the following.

Observation 7.12. *Each object x_i with $\mathrm{A}(x_i) \in [\![\mathrm{A}(z), k]\!]$ is swapped exactly twice with x or z and each object x_j with $\mathrm{A}(x_j) \notin [\![\mathrm{A}(z), k]\!]$ is swapped exactly once with x or z.*

Note that for each edge $e \in \{\{i-1, i\} \mid i \in [k]\}$, there is exactly one object that is swapped with x over e and this object moves clockwise. Moreover, by Observation 7.12 each object moving clockwise is swapped with x over one of these edges. Thus, we can characterize a solution by choosing for each edge $e \in \{\{i-1, i\} \mid i \in [k]\}$ one object to move clockwise and be swapped with x over e.

7.3.1 Limited Number of Candidates

In this subsection, we will show that once object z is fixed, there are for each edge $e \in \{\{i-1, i\} \mid i \in [k]\}$ at most two candidate objects c_1, c_2 such that x is swapped with either c_1 or c_2 over e. We start with a series of helpful lemmata that will be used often throughout this subsection. The first lemma states that for each pair (x_i, x_j) of objects and each agent ℓ, the edge where x_i and x_j are swapped for the first time is the same in all sequences of swaps where x_i moves clockwise, x_j moves counter-clockwise, and agent ℓ holds both x_i and x_j during the sequence of swaps.

Lemma 7.13. *Let x_i and x_j be two objects and let $\ell \in [\![\mathrm{A}(x_i), \mathrm{A}(x_j)]\!]$ be an agent. There is an edge e such that for each sequence of swaps ϕ such that x_i moves clockwise in ϕ, object x_j moves counter-clockwise in ϕ, and agent ℓ holds both x_i and x_j during ϕ, it holds that x_i and x_j are swapped over e during ϕ. Deciding whether such an edge exists and computing it if it exists takes $O(n)$ time after an $O(n^2)$-time preprocessing step.*

Proof. We distinguish between the two cases $x_j \succ_\ell x_i$ and $x_i \succ_\ell x_j$. Since both cases are completely analogous, we only show the proof for the former case. Note that agent ℓ must then first hold object x_i before it holds object x_j as it would otherwise not accept object x_i after already holding x_j or an object it prefers over x_j. Thus, objects x_i and x_j must be swapped between two agents in $[\![\ell, \mathrm{A}(x_j)]\!]$. Now iteratively consider the preference list of an agent $\ell' \in [\![\ell, \mathrm{A}(x_j)]\!]$ (starting with agent $\ell+1 \bmod n$). If agent ℓ' also prefers x_j over x_i, then x_i and x_j cannot be swapped over the edge $\{\ell'-1, \ell'\}$. Hence, agent ℓ' must also hold object x_i before it holds x_j and we can continue the argumentation until we either find an agent who prefers x_i over x_j or we reach agent $\mathrm{A}(x_j)$ and $\mathrm{A}(x_j)$ also prefers x_j over x_i. If we reach agent $\mathrm{A}(x_j)$ and $\mathrm{A}(x_j)$ also prefers x_j over x_i, then all agents in $[\![\ell, \mathrm{A}(x_j)]\!]$ prefer x_j over x_i and hence these two objects cannot be swapped between two such agents. If agent ℓ' prefers x_i over x_j, then these two objects can only be swapped over the edge $\{\ell'-1, \ell'\}$ as shown next. Assume towards a contradiction that x_i and x_j were swapped over another edge $\{h-1, h\}$ where $h \in [\![\ell'+1, \mathrm{A}(x_j)]\!]$. Then, agent ℓ' has to pass object x_i towards agent h before agent ℓ' holds object x_j. This means, however, that agent ℓ' will not accept object x_j as it prefers x_i over x_j. Thus, object x_i cannot be passed to agent ℓ, a contradiction.

It remains to analyze the running time of the algorithm. We first describe a simple preprocessing step that eases the computation of deciding which of two objects an agent prefers. We define $\mathrm{pos}(i, x_j)$ as the position of object x_j in the preference list of agent i. Note that pos can be precomputed once in $O(n^2)$ time by iterating over the preference list of each agent. Once the preprocessing is done, we have to (in the worst case) check for each agent $\ell' \in [\![\ell, \mathrm{A}(x_j)]\!]$ whether they prefer x_i over x_j or not. Since this is only a check whether $\mathrm{pos}(\ell', x_i) < \mathrm{pos}(\ell', x_j)$ or not, the whole procedure takes $O(n)$ time in total after preprocessing. Note that the preprocessing can be done once and then be reused for each application of Lemma 7.13. $\square$

Based on Lemma 7.13, we define the set of edges where two objects x_i and x_j can be swapped.

Definition 7.1. Let x_i and x_j be two objects and let a, b be two agents such that $a \in [\![\mathrm{A}(x_i), \mathrm{A}(x_j)]\!]$ and $b \notin [\![\mathrm{A}(x_i), \mathrm{A}(x_j)]\!]$. The *first edge* $\mathsf{fe}_a(x_i, x_j)$ is the edge computed by Lemma 7.13 for x_i, x_j, and a. If this edge does not exist, then $\mathsf{fe}_a(x_i, x_j) := \perp$. Let $\{s, s+1 \bmod n\} := \mathsf{fe}_{\mathrm{A}(x_i)}(x_i, x_j)$. The *second edge* $\mathsf{se}_b(x_i, x_j)$ is the edge computed by Lemma 7.13 for x_i, x_j, and b after x_i and x_j have been swapped over $\{s, s+1 \bmod n\}$, that is, when agent s initially holds x_j and agent $s+1 \bmod n$ initially holds object x_i. If this edge does not exist, then $\mathsf{se}_b(x_i, x_j) := \perp$.

The second lemma states that an object can never "overtake" another object that moves in the same direction in the cycle.

Lemma 7.14. *Let x_h, x_i, and x_j be three objects such that $\mathrm{A}(x_i) \in [\![x_h, x_j]\!]$. Let $\phi = (\sigma_0, \sigma_1, \ldots, \sigma_s)$ be a sequence of swaps in which the three objects move in the same direction. For each $p \in [s]$, it holds that $\sigma_p^{-1}(x_i) \in [\![\sigma_p^{-1}(x_h), \sigma_p^{-1}(x_j)]\!]$.*

Proof. We assume that x_h, x_i, and x_j are distinct as otherwise the statement trivially holds. Assume towards a contradiction that there is some minimal $p \in [s]$ such that $\sigma_p^{-1}(x_i) \notin [\![\sigma_p^{-1}(x_h), \sigma_p^{-1}(x_j)]\!]$. Since p is minimal, it holds that $\sigma_{p-1}^{-1}(x_i) \in [\![\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)]\!]$. We now consider the swap $\{(a, x_c), (b, x_d)\}$ between σ_{p-1} and σ_p. Since x_c and x_d are swapped in ϕ they move in different directions. Let without loss of generality x_p be the object that moves in the same direction as x_i, x_j, and x_h. If $\{x_i, x_j, x_h\} \cap \{x_c\} = \emptyset$, then

$$\sigma_p^{-1}(x_j) = \sigma_{p-1}^{-1}(x_j) \in [\![\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)]\!] = [\![\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)]\!],$$

a contradiction. Hence, $x_c \in \{x_i, x_j, x_h\}$. We distinguish between the two cases $x_c = x_j$ and $x_c \in \{x_i, x_h\}$. If $x_c = x_j$, then

$$[\![\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)]\!] = [\![\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)]\!].$$

Since x_i, x_j and x_h are distinct objects, it holds that

$$\sigma_{p-1}^{-1}(x_j) \in [\![\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)]\!] \setminus \{\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)\}.$$

Thus, since each object can only move one position in each swap, it holds that

$$\sigma_p^{-1}(x_j) \in [\![\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)]\!] = [\![\sigma_p^{-1}(x_h), \sigma_p^{-1}(x_j)]\!],$$

which is again a contradiction. Finally, if $x_c \in \{x_i, x_h\}$, then note that $x_d \neq x_i$ as they move in different directions. Hence,

$$\sigma_p^{-1}(x_j) = \sigma_{p-1}^{-1}(x_j) \in [\![\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)]\!].$$

Again, since the three objects are distinct, x_j held by an agent

$$\sigma_{p-1}^{-1}(x_j) \in [\![\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)]\!] \setminus \{\sigma_{p-1}^{-1}(x_h), \sigma_{p-1}^{-1}(x_j)\}$$

and since in each step this interval can shrink by at most one, it follows that

$$\sigma_p^{-1}(x_j) = \sigma_{p-1}^{-1}(x_j) \in [\![\sigma_p^{-1}(x_h), \sigma_p^{-1}(x_j)]\!],$$

a contradiction. □

Finally, the third lemma states that there is a constant distance c such that for each object x_i which is swapped twice with x, the distance between the two edges where x_i is swapped with x have distance c in the input graph.

Lemma 7.15. *Let ϕ be a sequence of swaps such that agent I swaps z for x in the last swap of ϕ. Let x_i be an object that moves clockwise in ϕ such that $\mathrm{A}(x_i) \in [\![\mathrm{A}(x), \mathrm{A}(z)]\!]$. Then, $\mathsf{se}_I(x_1, x) \neq \bot \neq \mathsf{se}_I(z, x)$. Let*

$$\{s_1, s_1+1\} := \mathsf{fe}_k(x_1, x), \qquad \{t_1, t_1+1\} := \mathsf{se}_I(x_1, x),$$
$$\{s_2, s_2+1\} := \mathsf{fe}_k(z, x), \text{ and} \qquad \{t_2, t_2+1\} := \mathsf{se}_I(z, x).$$

Then, it holds that $s_1 - t_1 = s_2 - t_2$.

Proof. This lemma almost directly follows from Lemmata 7.9 and 7.14. Assume towards a contradiction that there is some x_i that moves clockwise and swapped twice with x in ϕ such that $s_1 - t_1 \neq s_2 - t_2$. Consider the set Y of objects x_j that move clockwise in ϕ and such that $\mathrm{A}(x_j) \in [\![\mathrm{A}(z), \mathrm{A}(x_i)]\!]$ (excluding x_i and z). By Lemma 7.9, $s_1 - t_1 = |Y|$. Now consider the assignment σ_r in ϕ after the first swap of x_i and x and the set Y' of objects x_j that move clockwise in ϕ with $\sigma_r^{-1}(x_j) \in [\![\sigma_r^{-1}(z), \sigma_r^{-1}(x_i)]\!]$. By Lemma 7.9 it holds that $s_2 - t_2 = |Y'|$ and by Lemma 7.14 that $Y = Y'$. Thus, $s_1 - t_1 = |Y| = |Y'| = s_2 - t_2$, a contradiction. □

We continue with the central definition of this subsection: the *type* of an object. The type of an object is represented by the index of the edge where the object can possibly be swapped with x for the first time. The idea behind types is the following. We will develop a 2-SAT program to determine which objects move clockwise in a solution. We will show that there are at most two objects of each type an, roughly speaking, we will introduce a variable for each type that represents which of the two objects of a type moves clockwise. We will use Lemma 7.13 to define the type of an object x_j. It only remains to find an agent which holds each of x_j and x at some point in time. If $\mathrm{A}(x_j) \in [I, k]$, then object x has to pass agent $\mathrm{A}(x_j)$ and hence we can use this agent. If $\mathrm{A}(x_j) \notin [I, k]$, then $I \in [\![\mathrm{A}(x_j), k]\!]$ and hence agent I has to first hold object x_j before it can receive object z and hence we can use agent I in Lemma 7.13.

Definition 7.2. The *index* of an edge $\{t-1, t\}$ with $t \in [k]$ is t. For each object x_j with $\mathrm{A}(x_j) \in [I, k]$, the *type* of y is the index of $\mathsf{fe}_{\mathrm{A}(x_j)}(x_j, x)$. For each object x_j with $\mathrm{A}(x_j) \notin [I, k]$, the *type* of y is the index of $\mathsf{fe}_I(x_j, x)$. If the respective value is $\perp$, then the type of x_j is 0. The *candidate set* $\mathcal{C}_\alpha$ *for* α contains all objects of type α.

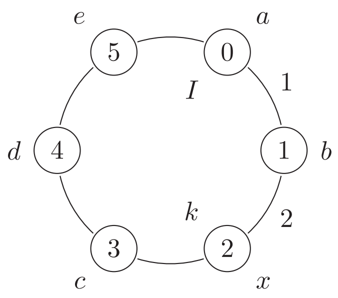
Figure 7.6 shows an example of types. We continue by showing that exactly one object of each type moves clockwise in any solution. We use t_z to denote the type of z. Note that x and z have to be swapped for the first time over the edge $\{t_z - 1, t_z\}$. By Lemma 7.9, for each edge

$$\{t_z, t_z + 1\},\ \{t_z + 1, t_z + 2\}, \dots, \{k-1, k\}$$

one object x_i with $\mathrm{A}(x_i) \in [\![\mathrm{A}(z), k]\!]$ has to move clockwise and be swapped with x over the respective edge. By Definition 7.2, these objects have types

$$t_z, t_z + 1, \dots, k$$

and, by Observation 7.12 and Lemma 7.15, these $k - t_z + 1$ objects are swapped a second time with x over the edges $\{0, 1\}, \{1, 2\}, \dots, \{k - t_z, k - t_z + 1\}$. Hence for each edge $\{k - t_z + 1, k - t_z + 2\}, \{k - t_z + 2, k - t_z + 3\}, \dots, \{t_z - 2, t_z - 1\}$ there is an object that is swapped once with x. Since the number of such objects is $(t_z - 1) - (k - t_z + 1) = 2t_z - k - 2$, there are $(2t_z - k - 2) + (k - t_z + 1) = t_z - 1$ objects that move clockwise in total in each solution where x moves counterclockwise and z moves clockwise. By definition of types, these have to have types $\alpha \in [k - t_z + 1, k]$.



$$\begin{aligned}
&I : x \succ d \succ \boxed{a}\\
&1 : d \succ a \succ x \succ \boxed{b}\\
&2 : b \succ \boxed{x}\\
&3 : \boxed{c}\\
&4 : e \succ \boxed{d}\\
&5 : a \succ d \succ \boxed{e}
\end{aligned}$$

Figure 7.6: An example for types. The objects a, b, c, d, e, and x are placed next to the agents that initially hold them. The numbers next to edges between I and k depict the index of the respective edge. Objects c and e can never be swapped with x before x reaches agent I. Thus they both have type 0. The type of object b is 2 as objects b and x can only be swapped over the edge with index 2 for the first time if x moves counter-clockwise. The type of a and d is 1 since if x moves counter-clockwise and is swapped with either a or d over a different edge than $\{0, 1\}$, then agent 1 holds the respective object before it holds x. Since agent 1 prefers either object over object x, it would not accept x and hence there is no solution in which a or d is swapped with x over a different edge than $\{0, 1\}$.

Observation 7.16. *Let $\phi := (\sigma_0, \sigma_1, \dots, \sigma_t)$ be a sequence of swaps which satisfies Assumptions 7.8 and 7.11 and such that $\sigma_t(I) = x$.*

For each type $\alpha \in [k - t_z + 1, k]$ there is exactly one object x_α of type α that moves clockwise in ϕ. All objects whose type is not in $[k-t_z+1, k]$ move counter-clockwise in ϕ. For $\alpha \in [k - t_z + 1, t_z - 1]$, it holds that $\mathrm{A}(x_\alpha) \notin [\![\mathrm{A}(z), k]\!]$. For $\alpha \in [t_z, k]$, it holds that $\mathrm{A}(x_\alpha) \in [\![\mathrm{A}(z), k]\!]$.

Using Observation 7.16, we can now formalize *selections.* Selections are an equivalent way of think about Reachable Object on cycles. They characterize which objects move clockwise and which objects move counter-clockwise.

Definition 7.3. Let $\lambda \subseteq [k]$. A set ι of objects is a *selection for* λ if it contains exactly one object of each type in λ and no other objects. A set ι is a *selection* if it is a selection for $[k - t_z + 1, k]$.

We will show in Subsection 7.3.2 how to test whether a given selection leads to a solution, that is, a sequence of swaps such that agent I obtains object x. In the remainder of this subsection, we focus on eliminating possible selections.

Observe that if the type of an object is 0, then it cannot be swapped with x and hence it has to be moved counter-clockwise. We will slightly misuse the definition of types and relabel the type of any object x_j to 0 if we know for some reason that x_j has to move counter-clockwise. Lemma 7.15 states a first rule that can be used to relabel the type of an object to 0. Hence, we assume that each object of type $\alpha \neq 0$ fulfills the conditions of Lemma 7.15. We conclude this subsection with a proposition that identifies at most two "relevant" objects of each type and allows us to relabel the type of all other objects of this type to 0 (because they can not be moved clockwise in a solution). To this end, we define the *subtypes* of an object. Roughly speaking, the subtype of an object x_i encodes whether x_i is "closer" (counted in clockwise steps) to z than the other object that can be considered or whether x_i is "further away". We distinguish between objects that are possibly swapped once with x and objects that are possibly swapped twice with x. The main idea is that if x_i is not selected (it moves counter-clockwise), then it has to be swapped with z. Thus we can use Lemma 7.13 to compute the edge where x_i and z can be swapped. We can then check whether another object of the same type α as x_i has to move clockwise in order for x_i to reach the respective edge where x_i and z can be swapped. We say that x_i has subtype f if another object of type α between z and x_i has to move clockwise in order for x_i to reach the specified edge. Otherwise, we say that x_i has subtype c. This characteristic is captured by the following definition.

Definition 7.4. Let x_i be an object of type $\alpha \in [k - t_z + 1, t_z - 1]$, let x_j be an object of type $\beta \in [t_z + 1, k]$, and let t_z be the type of z. If $\mathrm{A}(x_i) \in [I, \mathrm{A}(z) - 1]$, then let $e := \mathsf{fe}_I(z, x_i)$ and if $\mathrm{A}(x_i) \in [k, n-1]$, then let $e := \mathsf{fe}_{\mathrm{A}(x_i)}(z, x_i)$. If $\{a-1, a\} := e \neq \bot$, then $h := |[\![a, \mathrm{A}(y)]\!]| - 1$ is the *distance between x_i and z*. If $\alpha > h$, then the *subtype* of x_i is c (for closer) and if $\alpha \leq h$, then the *subtype* of x_i is f (for further).

Let $e_1 := \mathsf{fe}_{\mathrm{A}(x_j)}(z, x_j)$ and $e_2 := \mathsf{se}_I(z, x_j)$. If $\{b-1, b\} := e_1 \neq \bot$ and $\{c-1, c\} := e_2 \neq \bot$, then $h := |[\![b, \mathrm{A}(x_j)]\!]| - 1$ is the *distance between x_j and z* and let $h' := |[\![c, b-1]\!]| - 1$. If $\alpha > t_z + h$ and $h' = t_z - 2$, then the *subtype* of x_j is c and if $\alpha \leq t_z + h_1$ and $h' = t_z - 2$, then the *subtype* of x_j is f.

See Figure 7.7 for an illustration of subtypes. If $e = \bot$, then x_i cannot move counter-clockwise and hence we can relabel the type of all other objects of type α to 0. Analogously, if $\{e_1, e_2\} \cap \{\bot\} \neq \emptyset$, then x_j cannot move counter-clockwise and hence we relabel the type of all other objects of type β to 0.

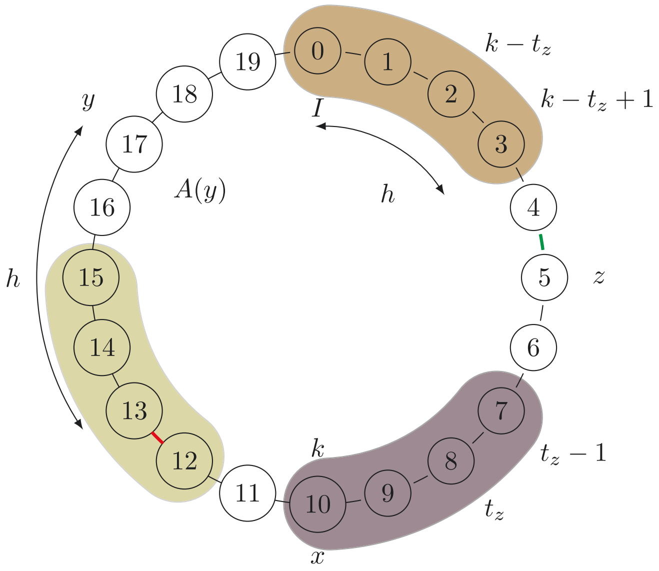


Figure 7.7: An example that illustrates the main idea behind Definition 7.4 and Proposition 7.18. Some objects are depicted next to the agents that initially hold them. Let the type of y be $\beta = 6$ and assume that object y moves counter-clockwise. It therefore has to be swapped with z at some point and since z has to pass agent $\mathrm{A}(y)$ to reach agent I, we can use Lemma 7.13 to compute the edge $e := \mathsf{fe}_{\mathrm{A}(y)}(z, y)$ where y and z swap for the first time. Let $e = \{12, 13\}$ (the red edge). The distance h computed in Definition 7.4 is then $h := |[\![13, \mathrm{A}(y)]\!]| - 1 = |[\![13, 17]\!]| - 1 = 4$ and describes the number of edges that object y has to pass before it can be swapped with z. Note that for each type $\alpha \geq t_z$, there is an object of type α that is swapped with x twice (first in the violet (bottom right) region between agents $t_z - 1$ and k and, by Lemma 7.15, a second time in the orange region (top right) between agents I and $k - t_z + 1$). All of these objects have to be swapped with y and, by Lemma 7.14, this has to be in the yellow (left) region as z is swapped with y over the red edge and all other objects have to be swapped with y on consecutive edges. Similarly, the object that is swapped with y over the edge $\{15, 16\}$ is swapped with x over the edge $\{3, 4\}$, the object that is swapped with y over the edge $\{16, 17\}$ is swapped with x over the edge $\{4, 5\}$, and so on. Hence, the object that is swapped with y over the edge $\{16, 17\}$ (the first edge of y in counter-clockwise direction) is swapped with x over the edge $\{h, h+1\} = \{4, 5\}$ (green) and is therefore of type $h + 1 = 5$. Since $\beta > h$, the subtype of y is f and y is not swapped with an object of type β that is initially held by an agent in $[\![\mathrm{A}(z), \mathrm{A}(y)]\!]$.

Before we show the main proposition of this section, we prove a lemma that characterizes the types of all objects an object moving counter-clockwise is swapped with.

Lemma 7.17. *Let x_i be an object of type α and let h be the distance between x_i and z. If x_i moves counter-clockwise, then*

- *if $h \leq k - t_z$, then x_i is swapped with an object of each type $\beta \in [t_z, t_z + h]$ before x_i is swapped with z for the first time, and*
- *if $h > k - t_z$, then for each type $\beta \in [t_z, k] \cup [k - t_z + 1, h]$ it holds that x_i is swapped with an object of type β before x_i is swapped with z for the first time.*

Proof. Let x_i be an object of type α that moves counter-clockwise. We consider the two cases $\mathrm{A}(x_i) \in [I, \mathrm{A}(z) - 1]$ and $\mathrm{A}(x_i) \in [\mathrm{A}(z), n - 1]$.

If $\mathrm{A}(x_i) \in [I, \mathrm{A}(z) - 1]$, then note that agent I has to hold object x_i before it can obtain z. Hence, by Lemma 7.13, objects x_i and z are swapped over the edge $\mathsf{fe}_I(z, x_i)$. If $\mathrm{A}(x_i) \in [\mathrm{A}(z), n - 1]$, then note that agent $\mathrm{A}(x_i)$ has to hold object z before agent I can hold object z. Hence, by Lemma 7.13, objects x_i and z are swapped over the edge $\mathsf{fe}_{\mathrm{A}(x_i)}(z, x_i)$.

If the respective edge where x_i and z can swap for the first time exists, then we denote it by $\{a - 1, a\}$. Note that the distance h between x_i and z exactly describes the number of edges in the path between $\mathrm{A}(x_i)$ and a that x_i has to pass before it can be swapped with z. Note that each object with which x_i is swapped moves clockwise. By Lemmata 7.9 and 7.15, the object that is swapped with x_i over the edge $\{a, a+1\}$ is swapped with x over the edge $\{t_z, t_z + 1\}$ and it therefore has type $t_z + 1$. Repeating this argument, the object that is swapped with x_i over the edge $\{a + 1, a + 2\}$ is of type $t_z + 2$ and so on until type k is reached (after $k - t_z$ iterations). Thus, if $h \leq k - t_z$, then x_i is swapped with an object of each type $\beta \in [t_z, t_z + h]$. If $h > k - t_z$, then x_i is swapped with an object of each type $\beta \in [t_z, k]$ in the first $k - t_z$ iterations. The next type after k has then to be $k - t_z + 1$ as the object of type k is swapped with x also over the edge $\{k - t_z - 2, k - t_z - 1\}$. Thus, if $h > k - t_z$, then x_i is also swapped with an object of each type $\beta \in [k - t_z + 1, h]$. □

We conclude with the main result of this subsection. This allows us to relabel the type of all except for two objects of some type $\alpha \neq 0$ to 0. If two objects of type $\alpha \neq 0$ remain afterwards, then they have different subtypes.

Proposition 7.18. *Given objects x and z, there is an $O(n^2)$-time preprocessing that excludes all but at most two objects of each type $\alpha \geq k - t_z + 2$ as potential candidates for being swapped with x. Afterwards $|\mathcal{C}_\alpha| \leq 2$ for all $\alpha \in [k-t_z+1, k]$.*

Proof. Consider a type $\alpha \geq k - t_z + 2$ and all objects of type α. Compute the subtype of each of these objects. Exactly one of them is moved clockwise and all others have to be swapped with z at some point. Let x_i be an object of type α. We now consider the two cases $\alpha \in [k-t_z+1, t_z-1]$ and $\alpha \in [t_z+1, k]$. We start with the case where $\alpha \in [k-t_z+1, t_z]$. Note that in this case if $\mathrm{A}(x_i) \in [\![\mathrm{A}(z), k]\!]$ and x_i moves clockwise, then, by Lemma 7.9, x_i is swapped with x before x and z are swapped. Hence, x_i cannot be swapped with x for the first time over the edge $\{\alpha - 1, \alpha\}$. Thus, x_i cannot move clockwise and its type can be relabeled to 0. We therefore assume that $\mathrm{A}(x_i) \notin [\![\mathrm{A}(z), k]\!]$. We consider the two cases $\mathrm{A}(x_i) \in [I, \mathrm{A}(z)-1]$ and $\mathrm{A}(x_i) \in [k, n-1]$. If $\mathrm{A}(x_i) \in [I, \mathrm{A}(z)-1]$ and x_i moves counter-clockwise, then note that agent I has to hold object x_i before it can obtain x. Hence, by Lemma 7.13, objects x_i and z are swapped over the edge $\mathsf{fe}_I(z, x_i)$. If $\mathrm{A}(x_i) \in [k, n-1]$, then note that agent $\mathrm{A}(x_i)$ has to hold object z before agent I can hold object z. Hence, by Lemma 7.13, objects x_i and z are swapped over the edge $\mathsf{fe}_{\mathrm{A}(x_i)}(z, x_i)$. Note that if the respective edge does not exist (the respective value is $\perp$), then x_i cannot move counter-clockwise and thus all other objects of type α have to move counter-clockwise and we can therefore relabel their type to 0. Note further that there is no solution if such an edge does not exist for multiple objects of the same type.

If $\mathrm{A}(x_i) \in [t_z+1, k]$ and x_i moves clockwise, then note that $\mathrm{A}(x_i) \in [\![\mathrm{A}(z), k]\!]$ or object x_i cannot be swapped twice with x before x and z are swapped twice. Hence, $\mathrm{A}(x_i)$ holds z during a solution and we can use Lemma 7.13 to compute the edge $\mathsf{fe}_{\mathrm{A}(x_i)}(z, x_i)$ where x_i and z are swapped over for the first time. Again, if the respective edge does not exist (the respective value is $\perp$), then x_i cannot move counter-clockwise and thus all other objects of type α have to move counter-clockwise and we can therefore relabel their type to 0. Moreover, if x_i and z have to be swapped a second time over the edge $e' := \mathsf{se}_I(z, x_j)$. If $\{b-1, b\} := e \neq \perp$ and $\{c-1, c\} := e' \neq \perp$, then let $h' := |[\![c, b-1]\!]| - 1$. Note that if $h \neq t_z - 1$, then x_i and z cannot be swapped over $\{b-1, b\}$ and $\{c-1, c\}$ as there are, by Observation 7.16, exactly $t_z - 1$ objects that move clockwise in any solution. Hence, in this case x_i has to move clockwise and we can relabel the type of all other objects to 0.

By Lemma 7.17, object x_i is swapped with an object of type α if and only if $h \geq \alpha$, that is, if the subtype of x_i is f.

We now show that there are at most two candidates for each type α. To this end, we iterate over all agents, starting with $\mathrm{A}(z)$ and iterating clockwise. If the object that is initially held by the agent is of type α, then we add its subtype to the end of an initially empty sequence. We then distinguish whether the sequence is *sorted*, that is, it is $(c, c, \ldots, c, f, f, \ldots, f)$, or not. If the sequence is not sorted, then for each object x_i of type α, there exists an object of type α and subtype f that starts in $[\![\mathrm{A}(z), \mathrm{A}(x_i)]\!]$ or an object of type α and subtype c that starts in $[\![\mathrm{A}(x_i), \mathrm{A}(z)]\!]$. Thus, if x_i moves clockwise, then the number of counter-clockwise steps of some other object of type α does not match the number of swaps needed to reach the edge where the objects can be swapped with z. Hence, no object of type α can be moved clockwise and therefore there is no solution.

Now consider the case where the objects are sorted by their subtype. By the same argument as above there are only two possible objects of type α that can possibly be moved clockwise: The "last" object of subtype c and the "first" object of subtype f. We can therefore set the type of all other objects of type α to 0.

It remains to analyze the running time. Let n_α be the number of objects of type α. Since the subtype for each object of type α can be computed in $O(n)$ time, we obtain that the described preprocessing takes $O(n_\alpha \cdot n)$ time for type α. After having computed the subtype of each object of type α, we iterate over all these objects and find in $O(n)$ time the two specified objects or determine that the objects are not ordered by their subtype. Hence, the overall running time is in $O(\sum_{\alpha > t_z} (n_\alpha \cdot n))$. Note that each object (except for x) has exactly one type and hence $\sum_{\alpha > t_z} n_\alpha < n$. Thus, the overall running time is bounded by $O(\sum_{\alpha > t_z} (n_\alpha \cdot n)) \subseteq O(n^2)$. $\square$

Proposition 7.18 shows that there are at most two objects of each type. In the following, we will partition types into blocks where we will observe that for each block there are at most two possible choices of which objects of the respective types to move clockwise. These choices will then be used to develop a 2-SAT program. Note that the proof of Proposition 7.18 also states that if there are two objects of some type $\alpha \neq 0$, then one of them has subtype c and one has subtype f. Hence, we can uniquely identify any object x_i which does not have type 0 by its type-subtype combination. For the sake of readability, we will denote the unique object of type α and subtype c by α_c. Analogously, β_f is the unique object of type β and subtype f. If an object x_i is the only object of some type $\alpha \neq 0$, then we say that $\alpha_c = \alpha_f = x_i$.

7.3.2 Compatibility of Solutions

So far, we have shown how to compute a set of at most two candidates to be swapped with x over each edge which x has to pass. Recall that the main idea of our algorithm for Reachable Object on cycles is as follows. We first partition the edges which x has to pass such that

1. for each part of the partition there are at most two possible choices for selecting a candidate for all edges in the respective part and
2. candidates for two different parts are either incompatible, that is, there is no solution for the overall problem that uses the respective candidates, or they can be combined independently of the choices of candidates for other parts.

We then develop a 2-SAT program with a variable for each part of the described partition and use it to compute a set of pairwise compatible candidates for each edge. We next show how to partition types (these represent all edges that x has to pass) such that there are only two possible selections for all types in one part of the partition (we will call those parts *blocks*). Afterwards, we prove that selections for different blocks can be picked almost independently such that a set of pairwise compatible selections for each block can be computed by a 2-SAT program.

Before we provide the formal definition of blocks, we first focus on objects of type 0 and show that no object of type 0 can initially be held by an agent "between" the two agents that initially hold the two objects of some type $\alpha \neq 0$. Note that Proposition 7.18 states that there are at most two objects of type α (one of subtype c and one of subtype f).

Lemma 7.19. *For each object x_h of type 0 and each type $\alpha \neq 0$, if there are two objects x_i and x_j of type $\alpha \neq 0$, then both of them are initially held by agents in $[\![\mathrm{A}(x_h), \mathrm{A}(z)]\!]$, both of them initially start in $[\![\mathrm{A}(z), \mathrm{A}(x_h)]\!]$, or the type of one of these two objects can be relabeled to 0.*

Proof. Assume that there is an object x_h of type 0 such that there are two objects x_i and x_j such that $\mathrm{A}(x_i) \in [\![\mathrm{A}(x_h), \mathrm{A}(z)]\!]$, $\mathrm{A}(x_j) \in [\![\mathrm{A}(z), \mathrm{A}(x_h)]\!]$, and both x_i and x_h have type $\alpha \neq 0$. Let d be the distance between $\mathrm{A}(x_h)$ and $\mathrm{A}(z)$. By Lemma 7.17, we can compute whether x_i swaps with an object of type α before it is swapped with z. If so, then x_i cannot move clockwise and hence its type can be relabeled to 0. If not, then x_j cannot move clockwise and hence its type can be relabeled to 0. □

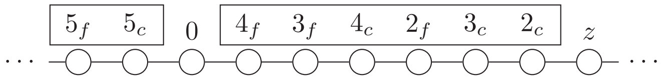


Figure 7.8: An example of *blocks*. Only a subpath of the input cycle with the objects initially held by the agents is depicted. The object 0 represents an object of type 0. The blocks in this example are $\{1\}$, $\{2, 3, 4\}$, and $\{5\}$. The boxes indicate all objects of types corresponding to each block. Note that 5_c and 5_f are adjacent and hence $\{5\}$ is a block as blocks are minimal. Note that $\{2, 3\}$ is not a block as 4_c is initially held by an agent in $[\![\mathrm{A}(2_c), \mathrm{A}(3_f)]\!]$. Since z is the only object of type 1, it always holds that $\{1\}$ is a block.

We assume that Lemma 7.19 has been exhaustively applied to relabel the type of objects to 0. This will help us to define *blocks*. Intuitively, blocks are sets of consecutive types $\alpha, \alpha + 1, \dots, \beta$ such that all objects of those types start on a (connected) subpath of the input graph.

Definition 7.5. A *block* is a minimal subset $B \subseteq [k - t_z + 1, k]$ of types such that there are two agents a and b and all objects whose type is in B are initially held by agents in $[\![a, b]\!]$ and all objects that are initially held by agents in $[\![a, b]\!]$ have a type in B.

Figure 7.8 depicts an example of blocks. Based on blocks we can state a new rule to relabel the type of an object to 0.

Lemma 7.20. *Let $A = [\alpha, \beta]$ be a block and let $\gamma \in [\alpha, \beta]$ be a type. If for some $\delta \in [\gamma + 1, \beta]$ it holds that $\mathrm{A}(\delta_c) \in [\![\mathrm{A}(z), \mathrm{A}(\gamma_c)]\!]$, then there is no solution in which δ_c moves clockwise. If $\mathrm{A}(\epsilon_f) \in [\![\mathrm{A}(z), \mathrm{A}(\gamma_f)]\!]$ for some $\epsilon \in [\alpha, \gamma - 1]$, then there is no solution in which γ_f moves clockwise.*

Proof. First, assume towards a contradiction that $\mathrm{A}(\delta_c) \in [\![\mathrm{A}(z), \mathrm{A}(\gamma_c)]\!]$ and that there is a solution in which δ_c moves clockwise. Note that by Proposition 7.18 and the definition of subtypes, it holds that $\mathrm{A}(\eta_c) \in [\![\mathrm{A}(z), \mathrm{A}(\eta_f)]\!]$ for each $\eta \neq 0$. Hence, there is no object of type γ that is initially held by an agent in $[\![\mathrm{A}(z), \mathrm{A}(\delta_c)]\!]$. Consider the solution where δ_c moves clockwise up to the assignment σ_r after the swap of δ_c and x over the edge $\{\delta - 1, \delta\}$. By Lemma 7.9, no object of type γ is held by an agent in $[\![\sigma_r^{-1}(z), \sigma_r^{-1}(x)]\!]$. Thus, x cannot be swapped over the edge $\{\gamma - 1, \gamma\}$, a contradiction to the assumption that we considered a solution.

Second, assume towards a contradiction that $\mathrm{A}(\epsilon_f) \in [\![\mathrm{A}(z), \mathrm{A}(\gamma_f)]\!]$ and there is a solution in which γ_f moves clockwise. Since $\mathrm{A}(\epsilon_c) \in [\![\mathrm{A}(z), \mathrm{A}(\epsilon_f)]\!]$, all objects of type ϵ are initially held by agents in $[\![\mathrm{A}(z), \mathrm{A}(\gamma_f)]\!]$. Consider the solution where γ_f moves clockwise and the assignment after γ_f and x are swapped over the edge $\{\gamma - 1, \gamma\}$. By Lemma 7.9 object x was not swapped with any object of type ϵ before it was swapped with γ_f. Hence, x cannot have passed the edge $\{\epsilon - 1, \epsilon\}$, a contradiction to Assumption 7.8. □

We henceforth assume that the type of objects satisfying Lemma 7.20 is relabeled to 0. We will show that blocks are the partitions we are looking for, that is, for each block A there are only two possible selections for A and selections for different blocks can be chosen almost independently. We start with a lemma that states that blocks define a partition of types.

Lemma 7.21. *Each type $\eta \geq k - t_z + 2$ is contained in exactly one block and all blocks can be computed in linear time.*

Proof. We first show that $\mathrm{A}(z)$ and $\mathrm{A}(x)$ divide the types into two intervals. Observe that all objects of type $\alpha \in [t_z + 1, k]$ have to start in $[\mathrm{A}(z), \mathrm{A}(x)]$ and all objects of type $\alpha \in [k - t_z + 2, t_z - 1]$ have to be initially hold by an agent in $[\![\mathrm{A}(x), \mathrm{A}(z)]\!]$. Since z is the only object of type t_z, the interval $[t_z, t_z]$ is a block and no other block can contain the type t_z. We now show that blocks are a partition of types that can be computed in linear time. We first focus on all other types starting with types in $[t_z + 1, k]$. Consider the object $(t_z + 1)_c$. This has to be the initially "closest" non-type-0 object to agent $\mathrm{A}(z)$. If $(t_z + 1)_c = (t_z + 1)_f$, then $[t_z + 1, t_z + 1]$ is a block. Otherwise, we know by Lemma 7.20 that the next object (in clockwise steps) has to be either $(t_z + 2)_c$ or $(t_z + 1)_f$. If the object ℓ_f is found, where ℓ is the largest type that is so far considered in the block, then the block $[t_z + 1, \ell]$ is found. Notice that $\ell \leq k$ and if a block is found, then we can redo the whole process starting with the object $(\ell + 1)_c$ until the block $[\ell', k]$ is found for some ℓ'. Starting then from agent k, we can search for object $(k - t_z + 2)_c$ and repeat the whole argumentation until a block $[\ell', t_z - 1]$ is found. At this point, each type is contained in exactly one block. Since we need only a constant amount of computation time for each object, all blocks can be computed in linear time. □

In order to prove that there are only two possible selections for each block that can lead to a solution, we first show an intermediate lemma.

Lemma 7.22. *Let $A = [\alpha, \beta]$ be a block and let ι_A be a selection for A. Let $\gamma \in A$ be a type. If $\gamma_f \in \iota_A$, then $\delta_f \in \iota_A$ for each $\delta \in [\gamma, \beta]$ or ι_A cannot be part of a selection that corresponds to a solution in which x reaches I.*

Proof. We prove the statement by induction on γ. Note that if $\gamma = \beta$, then the statement trivially holds. Now assume that $\gamma < \beta$, $\gamma_f \in \iota_A$, and if $(\gamma+1)_f \in \iota_A$, then $\delta_f \in \iota_A$ for each $\delta \in [\gamma+1, \beta]$. By Lemma 7.20, it holds that $(\gamma+1)_c$ is initially hold by an agent in $[\![\mathrm{A}(\alpha_c), \mathrm{A}(\gamma_f)]\!]$ and $(\gamma+1)_c$ cannot move clockwise since γ_f moves clockwise and is swapped with x over the edge $\{\gamma-1, \gamma\}$. This holds true as if $(\gamma+1)_c$ moved clockwise, then it holds by Lemma 7.9 that x is swapped with γ_f before it is swapped with $(\gamma+1)_c$. Hence, x cannot be swapped with $(\gamma+1)_c$ over the edge $\{\gamma, \gamma+1\}$ which is the type of $(\gamma+1)_f$. Thus, object $(\gamma+1)_c$ moves counter-clockwise and $(\gamma+1)_f$ moves clockwise. By induction hypothesis, $\delta_f \in \iota_A$ for each $\delta \in [\gamma, \beta]$. □

Based on Lemmata 7.20 and 7.22, we can now prove that there are only two possible selections for each block that can lead to a solution.

Lemma 7.23. *Let $A = [\alpha, \beta]$ be a block. There are at most two selections for A that can be part of a selection that corresponds to a solution in which x reaches I. These selections can be computed in $O(n \cdot |A|)$ time.*

Proof. We will construct two selections ι_1 and ι_2 for A that can be part of a selection that corresponds to a solution in which x reaches I. We start with $\alpha_c \in \iota_1$ and $\alpha_f \in \iota_2$. Note that by Lemma 7.22 $\iota_2 = \{\alpha_f, (\alpha+1)_f, \ldots, \beta_f\}$. Thus it remains to show that ι_1 is unique.

If α_c moves clockwise, then α_f has to move counter-clockwise. Using Lemma 7.17, we can compute the number h of objects x_i that are initially held by $\mathrm{A}(x_i) \in [\![\mathrm{A}(\alpha_c), \mathrm{A}(\alpha_f)]\!]$, that have types $\alpha+1, \alpha+2, \ldots, \alpha+h$, and that have to move clockwise. We now switch to an arbitrary type γ as we will use the statement iteratively (starting with $\gamma = \alpha$). If $\gamma = \beta$, then $\iota_1 = \{\alpha_c, (\alpha+1)_c, \ldots, \beta_c\}$. We therefore assume that $\gamma < \beta$, that γ_c moves clockwise, and that h objects x_i initially held by $\mathrm{A}(x_i) \in [\![\mathrm{A}(\gamma_c), \mathrm{A}(\gamma_f)]\!]$ and of types $\gamma+1, \gamma+2, \ldots, \gamma+h$ move clockwise. We consider the two cases $h = 0$ and $h > 0$. If $h = 0$, then note that, by Lemma 7.20, $\mathrm{A}((\gamma+1)_c) \in [\![\mathrm{A}(\gamma_c), \mathrm{A}(\gamma_f)]\!]$. Hence, $(\gamma+1)_c$ moves counter-clockwise and $(\gamma+1)_f$ moves clockwise. By Lemma 7.22, it holds for each $\delta \in [\gamma+1, \beta]$ that $\delta_f \in \iota_A$ and thus

$$\iota_1 = \{\alpha_c, (\alpha+1)_c, \ldots, \gamma_c, (\gamma+1)_f, (\gamma+2)_f, \ldots, \beta_f\}.$$

If $h > 0$, then we will show that $(\gamma+1)_c$ moves clockwise and hence we can repeat the argument. Assume towards a contradiction that $(\gamma+1)_c$ moves counter-clockwise. Then $(\gamma+1)_f$ moves clockwise and, by Lemma 7.22, so does δ_f

for each $\delta \in [\gamma+1, \beta]$. Hence, no object δ_c moves clockwise for $\delta \in [\gamma+1, \beta]$. Note that by definition of blocks it holds for each object x_i which is initially held by $\mathrm{A}(x_i) \in [\![\mathrm{A}(\gamma_c), \mathrm{A}(\gamma_f)]\!]$ that $x_i = \delta_c$ for some $\delta \in [\gamma, \beta]$ or that $x_i = \eta_f$ for some $\eta \in [\alpha, \gamma]$. If η_f moved clockwise for some $\eta \in [\alpha, \gamma]$, then, by Lemma 7.22, object γ_f also moved clockwise, a contradiction. Thus, if $(\gamma+1)_c$ moves counter-clockwise, then no object x_i initially held by $\mathrm{A}(x_i) \in [\![\mathrm{A}(\gamma_c), \mathrm{A}(\gamma_f)]\!]$ moves clockwise. Thus, γ_c and γ_f cannot be swapped and x cannot reach I.

It remains to analyze the running time. Computing ι_2 takes $O(n)$ time. Computing ι_1 takes $O(n)$ time for each γ_c with $\gamma \in [\alpha, \beta]$, that is, $O(n \cdot |A|)$ time in total. □

It is finally time to explain how to check whether a selection leads to a solution, that is, a sequence of swaps such that agent I obtains object x. Note that once a selection ι is fixed, Observation 7.12 states which objects are swapped how often with x or z. We assume that no object moves after it is swapped with x or z for the final time as these swaps are not necessary for x reaching I. Thus, we can compute the final position of each object and also the path $P^{\iota}_{x_i}$ of agents that hold each object x_i during a solution corresponding to ι. Gourvès et al. [GLW17] observed that once the path P_{x_i} of each object x_i is fixed, then the order in which objects are swapped is irrelevant as long as all objects "follow" their respective paths. Thus, there is a unique set of edges where two objects x_i and x_j swap in each solution in which the objects in ι move clockwise and all other objects move counter-clockwise. We denote this set by $e^{\iota}_{x_i,x_j}$. An example of $e^{\iota}_{x_i,x_j}$ and P_{x_i} is given in Figure 7.9. It remains to show how to compute $e^{\iota}_{x_i,x_j}$ and how to find a selection where each pair of objects can be swapped at the respective edge. To this end, we show how to compute $e^{\iota}_{x_i,x_j}$ from partial selections, that is, from selections for some subset λ of types.

Lemma 7.24. *Let x_i be an object of type $\alpha \in [k - t_z + 1, k]$, let x_j be an object of type $\alpha \in [k - t_z + 1, k]$, and let x_h be an object of type 0. Let A and B be two blocks with $\alpha \in A$ and $\beta \in B$. Given a selection ι_A for A, the set $e^{\iota}_{x_i,x_h}$ is the same for each selection $\iota \supseteq \iota_A$ and can be computed in $O(n)$ time. Given selections ι_A for A and ι_B for B, the set $e^{\iota}_{x_i,x_j}$ is the same for each selection $\iota \supseteq \iota_A \cup \iota_B$ and can be computed in $O(n)$ time.*

Proof. We start with determining which pairs of objects are not swapped, which pairs are swapped once and which are swapped twice. Let x_p be an object moving clockwise and let x_q be an object moving counter-clockwise. We consider

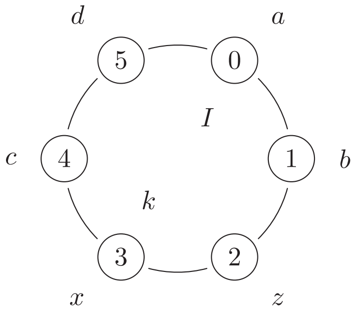


Figure 7.9: An example of Reachable Object on cycles. The objects initially held by agent are depicted outside each vertex. If objects z and c move clockwise ($\iota = \{c, z\}$ is the selection) and all other objects move counter-clockwise, then objects a and z swap over the edge $\{4, 5\}$ as z swaps with objects x and d over the edges $\{2, 3\}$ and $\{3, 4\}$, respectively, and object a swaps with c over the edge $\{0, 5\}$. The path P_c for this solution is $(4, 5, 0, 1, 2)$ as c is initially held by agent 4, moves clockwise, and is swapped with x over the edge $\{1, 2\}$.

the two cases $\mathrm{A}(x_p) \in [\![\mathrm{A}(z), k]\!]$ and $\mathrm{A}(x_p) \notin [\![\mathrm{A}(z), k]\!]$. If $\mathrm{A}(x_p) \in [\![\mathrm{A}(z), k]\!]$ and $\mathrm{A}(x_q) \in [\![\mathrm{A}(x_p), k]\!]$, then x_p and x_q are, by Lemma 7.9, swapped once before x and z are swapped for the first time and once afterwards. Thus, by Corollary 7.10, they are swapped exactly twice. If $\mathrm{A}(x_p) \in [\![\mathrm{A}(z), k]\!]$ and $\mathrm{A}(x_q) \notin [\![\mathrm{A}(x_p), k]\!]$, then x_p and x_q are swapped once as z and x_q are, by Observation 7.12, swapped exactly once and we assume that no object moves after it swapped with x or z for the final time.

If $\mathrm{A}(x_p) \notin [\![\mathrm{A}(z), k]\!]$, then we distinguish between the three cases

$$\mathrm{A}(x_q) \in [\![\mathrm{A}(z), k]\!],\ \mathrm{A}(x_q) \in [\![k+1, \mathrm{A}(x_p)]\!],\ \text{and}\ \mathrm{A}(x_q) \in [\![\mathrm{A}(x_p), \mathrm{A}(z)]\!].$$

In the first case, x_q is swapped twice with z and since, by Lemma 7.9, it is not swapped with x_p before it is swapped with z for the first time, it is swapped with x_p once. In the second case, x_q is swapped once with z and since, by Lemma 7.9, it is not swapped with x_p before it is swapped with z for the first time, it is not swapped with x_p. In the third case, x_q is swapped once with z and, by Lemma 7.9, it is swapped with x_p before it is swapped with z. Thus, in this case x_p and x_q are swapped once.

We now show how to compute the set of edges where two objects can be swapped. Note that it is enough to compute the first edge where two objects can be swapped as if two objects x_p and x_q are swapped twice, then the object moving counter-clockwise is, by Lemma 7.9, swapped with all objects moving clockwise before x_p and x_q are swapped again and there are always t_z objects moving clockwise. By definition of blocks and by Lemma 7.20, each object x_a with a type in $A \coloneqq [\gamma, \delta]$ is initially held by agent $\mathrm{A}(x_a) \in [\![\mathrm{A}(\gamma_c), \mathrm{A}(\delta_f)]\!]$. By Lemma 7.19, it holds for each type $\mu \in [k - t_z + 1, k]$ that $\mathrm{A}(x_h) \notin [\![\mathrm{A}(\mu_c), \mathrm{A}(\mu_f)]\!]$ and thus there is some type μ' such that $\mathrm{A}(x_h) \in [\![\mathrm{A}(\mu'_f), \mathrm{A}((\mu'+1)_c)]\!]$. We now compute the first edge where x_i and x_h can be swapped if x_i moves clockwise. Note that if x_i moves counter-clockwise, then it is not swapped with x_h. If $\alpha \leq \mu'$, then x_h is swapped once with an object of each type in $[\alpha, \mu'] \setminus \{\alpha\}$ before it is swapped with x_i. Hence $\{\mathrm{A}(x_h) - |\alpha - \mu'|, \mathrm{A}(x_h) - |\alpha - \mu'| + 1\} \in e^{\iota}_{x_i,x_h}$ is the first edge where x_i and x_h can be swapped. If $\alpha > \mu'$, then x_h is first swapped once with an object of each type $\mu', \mu' - 1, \ldots, k - t_z + 1, k, k-1, \ldots, \alpha + 1$ before it is swapped with x_i. Note that these are $t_z - |\{i \mid \mu' < i < \alpha\}|$ objects and therefore in this case $\{\mathrm{A}(x_h) - t_z + \alpha - \mu' - 1, \mathrm{A}(x_h) - t_z + \alpha - \mu'\} \in e^{\iota}_{x_i,x_h}$ is the first edge where x_i and x_h can be swapped.

It remains to analyze the possible edges for x_i and x_j. We assume without loss of generality that x_i moves clockwise and x_j moves counter-clockwise, that is, $x_i \in \iota_A$ and $x_j \notin \iota_B$. We distinguish between the two cases $A = B$ and $A \neq B$. If $A = B$, then $\iota_A = \iota_B$ and the direction of each object initially held by agents in $[\![\mathrm{A}(\gamma_c), \mathrm{A}(\delta_f)]\!]$ is known. Since the number of objects moving clockwise is constant (and equal to t_z), the number c of objects moving clockwise in $[\![\mathrm{A}(x_i), \mathrm{A}(x_j)]\!]$ is known and the first edge where x_i and x_j can be swapped is $\{\mathrm{A}(x_j) - c - 1, \mathrm{A}(x_j) - c\}$. If $A \neq B$, then let $B \coloneqq [\psi, \chi]$. Since ι_B is given, the number of objects in $[\![\mathrm{A}(\psi_c), \mathrm{A}(x_j)]\!]$ moving clockwise is known. We can then use the same argument as for x_h, where $\mu' = \psi - 1$ (or $\mu' = k$ if $\psi = k - t_z + 1$). Note that computing the unique set $e^{\iota}_{x_i,x_j}$ (or $e^{\iota}_{x_i,x_j}$) takes $O(n)$ time as we only compute the type of certain objects and the number of objects of certain types moving clockwise. □

An example of the set of edges computed in Lemma 7.24 is given in Figure 7.10. A selection ι leads to a solution if and only if for each pair (x_i, x_j) of objects such that $x_i \in \iota$ and $x_j \notin \iota$ and each edge $e \in e^{\iota}_{x_i,x_j}$, the agents incident to e can agree on swapping x_i and x_j. Hence, to check for a given selection ι whether it leads to a solution, we iterate over all pairs (x_i, x_j) of objects such that $x_i \in \iota$ and $x_j \notin \iota$ and distinguish between the following three cases.

- Either x_j does not have a type in $[k - t_z + 1, k]$, that is, $x_j = x$ or the type of x_j is 0,

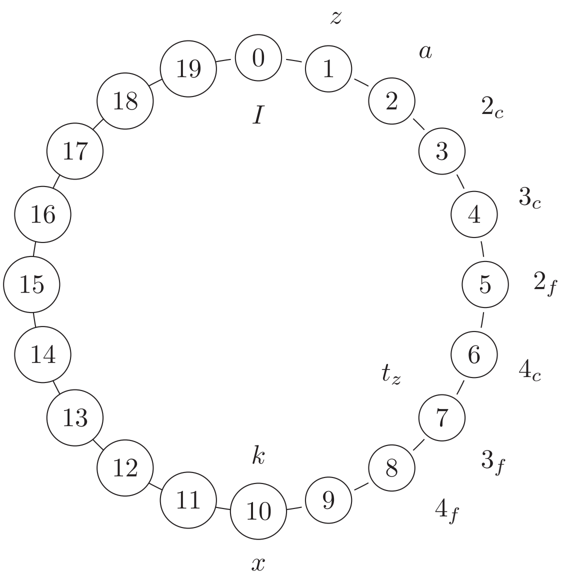


Figure 7.10: An example to illustrate Lemma 7.24. Depicted is a set of agents that are arranged in a cycle. Some objects are depicted next to the vertices of the agents which initially hold them. Notice that $A = [2, 3, 4]$ is a block and let $\iota_A := \{2_c, 3_f, 4_f\}$ be a selection for A. We show how to compute the edges $e^{\iota}_{2_c,4_c}$ for each selection $\iota \supset \iota_A$. Since objects 3_c and 2_f move counter-clockwise (they are not contained in ι_A), the two objects 2_c and 4_c next to them have to be swapped over edge $\{5, 6\}$. Afterwards, by Lemma 7.9, object 4_c is swapped with each other object moving clockwise before it is swapped with 2_c for a second time. Since $t_z - 1 = 6$ objects move clockwise, object 4_c moves over five edges after the first swap before it is swapped with 2_c for a second time. Thus, 2_c and 4_c are swapped for a second time over the edge $\{0, 19\}$.

- the types of x_i and x_j are in the same block, or
- the types of x_i and x_j are in different blocks.

The first two cases give rise to the notion of *consistent selections.* A selection ι_A for a block A is consistent if each object in ι_A can be swapped with each object x_j that has type 0 or a type in A over the respective edges in $e \in e^{\iota}_{x_i,x_j}$. Therein ι is any selection that generalizes ι_A, that is, $\iota \supseteq \iota_A$. For the sake of readability, we use $\mathcal{C}_A := \bigcup_{\alpha \in A} \mathcal{C}_\alpha$ to denote the set of all objects of a type in A.

Definition 7.6. Let A be a block and let ι_A be a selection for A. Let $x_i \in \iota_A$ be an object. Then, ι_A is *consistent* if

- for any object x_j of type 0 or $x_j \in \mathcal{C}_A \setminus \iota_A$ and
- for any edge $e \in e^{\iota}_{x_i,x_j}$ for any selection $\iota \supseteq \iota_A$

the agents incident to e can agree on swapping x_i and x_j.

We call a selection *inconsistent* if it is not consistent. Note that a given selection ι_A for a block A can be checked for consistency in $O(n^2 \cdot |A|)$ time by iterating over all $x_i \in \iota_A$ and all $x_j \in X$, computing the set $e^{\iota}_{x_i,x_j}$ in $O(n)$ time, and checking whether they can agree on the swap in constant time using the preprocessed pos-values.

The third requirement (checking whether two objects of types in different blocks can be swapped) gives rise to the notion of *compatible* selections. We say that two selections ι_A and ι_B for blocks A and B are compatible, if all pairs of objects of types in A and B, respectively, can be swapped at their respective edges.

Definition 7.7. Let $A = [\alpha, \beta]$ and $B = [\gamma, \delta]$ be two blocks. Let ι_A and ι_B be two selections for A and B, respectively. The selections are *compatible* if

- for all $x_i \in \iota_A$ and all $x_j \in \mathcal{C}_B \setminus \iota_B$,
- for all $x_i \in \mathcal{C}_A \setminus \iota_A$ and all $x_j \in \iota_B$,

and each edge $e \in e^{\iota}_{x_i,x_j}$ for some $\iota \supseteq \iota_A \cup \iota_B$, the agents incident to e can agree on swapping x_i and x_j.

We say that two selections ι_A and ι_B are *incompatible* if they are not compatible. Observe that given two selections ι_A and ι_B for blocks A and B, we can check them for compatibility in $O(|A| \cdot |B| \cdot n)$ time by iterating over all $x_i \in \iota_A$ and all $x_j \in \mathcal{C}_B \setminus \iota_B$ and compute the respective set of edges in $O(n)$ time using Lemma 7.24. Afterwards, we iterate over all $x_i \in \mathcal{C}_A \setminus \iota_A$ and all $x_j \in \iota_B$ and compute the respective set of edges. Checking whether the two agents incident to each of the at most two edges can agree on swapping x_i and x_j takes constant time as the pos-values are precomputed. It remains to find consistent selections for each block that are pairwise compatible. We solve this problem using 2-SAT programming. Therein, we have a variable for each block. The truth value of each variable represents which selection for the respective block is chosen and the clauses guarantee that no two incompatible selections are chosen.

Theorem 7.25. *Reachable Object on cycles can be solved in $O(n^4)$ time.*

Proof. We start the proof with an overview over the algorithm and an analysis of its running time. Afterwards, we show that the algorithm is correct. For each possible choice of object z (there are at most n many possibilities) and each of the two possible direction we assume x to move (two possibilities) we do the following. First, compute the type and subtype of each object. Second, compute all blocks and use Lemma 7.23 to compute two possible selections ι_A^1, ι_A^2 for each block A in overall $O(n^2)$ time. Afterwards, check each of these selections for consistency in overall $O(n^3)$ time and discard all inconsistent selections. Next, compute for each pair of consistent selections for different blocks whether they are compatible. This takes overall $O(n^3)$ time. Finally, check whether there are pairwise compatible selections for each block using the 2-SAT program below and return true if so.

Before we present the 2-SAT program, we first show a small preprocessing step. If for some block A there is only one consistent selection ι_A for A, then we discard all selections that are not compatible with ι_A as there is no set of pairwise compatible and consistent selections for all blocks that do not contain ι_A. Since all remaining selections are compatible with ι_A, we can ignore ι_A from now on. If this rule discards any selection, then the respective other selection for this block is the only consistent selection for this block and hence, we repeat the process. After at most n rounds, each of which only takes $O(n)$ time, we arrive at a situation where there are exactly two consistent selections for each block and the task is to find a set of pairwise compatible selections that include a selection for each block. We finally reduce this problem to a 2-SAT formula.

We start with a variable v_B for each block B which is set to true if we select ι_A^1 and set to false if we select ι_A^2. For each pair (ι_A, ι_B) of incompatible selections for different blocks A and B do the following. For the sake of simplicity, we use u and w to denote the literals representing the selections for blocks A and B that are incompatible. Formally, if $\iota_A = \iota_A^1$, then $u := v_A$ and otherwise $u := \neg\, v_A$. Analogously, if $\iota_B = \iota_B^1$, then $w := v_B$ and otherwise $w := \neg\, v_B$. Since we cannot select ι_A and ι_B at the same time (the formula cannot satisfy u and w at the same time), we add the clause $(\neg\, u \vee \neg\, w)$ to our 2-SAT formula.

Observe that if there is a set of pairwise compatible selections for each block, then the 2-SAT formula is satisfied by the corresponding truth assignment of the variables. Conversely, if the formula is satisfiable, then the selections for each block corresponding to a satisfying truth assignment specify a direction for each object. Any sequence of swaps that follows these directions will eventually lead to agent I obtaining object x. Since 2-SAT can be solved in linear time [APT79] and the constructed formula has $O(n^2)$ clauses of constant size, the overall running time for each possible choice of z and the direction of x is in $O(n^3)$. Since there are $O(n)$ possible choices for combinations of z and the direction of x, the overall running time for all iterations is in $O(n^4)$. □

7.4 Concluding Remarks

We investigated the computational complexity of Reachable Object with respect to restrictions on the maximum degree of the input graph and the maximum length of preference lists. Our work narrows the gap between known tractable and intractable cases leading to a more comprehensive understanding of the computational complexity of Reachable Object. In particular, we showed a dichotomy result regarding the length of the preference lists of the agents and showed polynomial-time solvability for Reachable Object on graphs with maximum degree at most two (note that a graph of maximum degree two is the disjoint union of paths and cycles and Huang and Xiao [HX20] resolved the case of paths). Saffidine and Wilczynski [SW18, Theorem 4] showed *NP*-hardness of Reachable Object on graphs of maximum degree at most four. Hence, the computational complexity of Reachable Object on graphs of maximum degree three remains the only open case towards a dichotomy result with respect to the parameter maximum degree. We conjecture that this case is NP-hard. Other interesting question regarding the maximum degree of a graph are whether Reachable Object is polynomial-time solvable on trees if the maximum degree is some constant and whether our running-time bound of $O(n^4)$ for graphs of maximum degree two is tight, that is, can it be improved to e. g. $O(n^3 \log n)$ or is there some (e. g. ETH-based) lower bound?

Note that in a cycle each object can take one of two paths towards its target object and these two paths translate to assigning each variable in our 2-SAT program one of the two possible truth values true or false. Jansen [Jan17] used a variant of 2-SAT where each variable can have one of N values (where N is some constant) to show containment in P for a variant of Hitting Set. It would be interesting to see whether there are graph classes in which each object can take one of constantly many paths to its target object where this generalization of 2-SAT can be used to show polynomial-time solvability.

Regarding modifications and generalizations of Reachable Object, note that in the Housing Market problem the agents cannot only swap in pairs but also in trading cycles. Trading cycles quite naturally translate into hyperedges in the input graph of Reachable Object. A set of agents can swap their currently held objects along a trading cycle only if they share a hyperedge. This generalization of Reachable Object seems to be a quite natural link between Housing Market and Reachable Object and has not been studied so far.

Chapter 8

Outlook

In this thesis, we shed some light on the computational complexity of special cases of different graph problems using mostly dynamic and 2-SAT programming. We conclude the thesis with a summary of our main results and a broader reflection on what we observed and how our work can be continued. Since we already provided directions for further research related to the problems studied in the respective chapters, we will focus on dynamic and 2-SAT programming here.

We start with a summary of our main results. For DIAMETER, we presented results within the field of *FPT in P*. In particular, using dynamic programming, we showed that DIAMETER is solvable in $O((n+m) \cdot h \cdot (d^h + h^d))$ time when parameterized by the h-index and diameter d. We further presented an $O(n^2 \cdot m)$-time algorithm for LENGTH-BOUNDED CUT on proper interval graphs and proved that k-DISJOINT SHORTEST PATHS is solvable in $n^{O((k+1)!)}$ time.

Using 2-SAT programming, we showed that SOFT TREE CONTAINMENT is solvable in $O(n^5)$ time when the input network is a 2-labeled phylogenetic tree. Complementing this result, we showed that SOFT TREE CONTAINMENT remains *NP*-hard when restricted to binary 3-labeled phylogenetic trees. Finally, we showed how to solve REACHABLE OBJECT in $O(n^4)$ time on cycles and proved a dichotomy result on arbitrary graphs parameterized by the length of the longest preference list of an agent. If all preference lists are of length at most three, then the problem can be solved in linear time and for lists of length at most four it remains *NP*-hard.

We continue with some concluding thoughts on dynamic and 2-SAT programming. Concerning dynamic programming, note that the three problems we studied in the first part of this thesis were all related to shortest paths in graphs. All three respective dynamic programs used the length of solution paths to some extent in the representation of a table entry. For DIAMETER, one of the dimensions of the respective table measures the length of a longest shortest path in the input graph. For Length-Bounded Cut, each table entry represents the minimum number of edges to delete in order to increase the length of a shortest path between the terminal s and each vertex in a given subset of vertices including t to some given threshold. Finally, for k-Disjoint Shortest Paths, each table entry represents whether there are disjoint paths between terminal pairs in a directed acyclic graph. We iterate over a topological order of this graph and hence allow for longer and longer disjoint paths. Concluding, we observed the following heuristic for how to use dynamic programming: *If the problem is about paths in graphs, then try to design a dynamic program that iteratively allows for longer and longer paths.*

Finally, we reflect on 2-SAT programming and answer the question we started this thesis with: When is 2-SAT programming a promising tool for solving algorithmic problems? Let us begin with revisiting how we and other authors used 2-SAT programming. All 2-SAT programs (including the ones from the literature) had in common that they, to some extent, compute a solution consisting of a set of pairwise compatible elements. In this thesis, these elements were either canonical vertices (in Chapter 6) or selections for blocks (in Chapter 7). Notice that in both cases exactly one out of at most two alternatives was chosen in the solution. The same holds true for all 2-SAT programs that we could find in the literature with one exception. Jansen [Jan17] used a version of 2-SAT where each variable can have one of N possible truth values in $[N]$ (where N is some constant) and a literal expresses that the truth value of a certain variable is at least or at most some given threshold. This version of 2-SAT is known to be polynomial-time solvable [BHM00]. Combining these insights, we present two heuristics of when to try applying 2-SAT programming to a new problem.

1. A (polynomial-time) reduction from the considered problem to Satisfiability or 3-SAT is known or easy to achieve. Observe in what special cases the reduction yields 2-SAT formulas.

2. The considered problem is (thought to be) polynomial-time solvable and has some *independence structure*, that is, a solution consists of some elements that can
   - be partitioned into constant-size parts and at most one element from each part is picked into the solution and
   - a set of elements forms a solution if each pair of elements in this set can be contained in the same solution.

With these heuristics at hand, we remark that we could not find any application of 2-SAT programming in the context of scheduling. This is surprising as scheduling is fundamentally about finding pairwise non-conflicting assignments of jobs to machines and time slots. We therefore conjecture that 2-SAT programming should be applicable in the context of scheduling quite often.

The importance of 2-SAT programming is not limited to exact polynomial-time algorithms either. 2-SAT programming has already been used in an approximation algorithm [WW95] and we believe that it might have further applications, for instance in heuristics or data reductions. It might even be useful to reduce some *NP*-hard problem to an exponential number of 2-SAT formulas (or one formula of exponential size) to achieve faster exponential-time algorithms.

Finally, there are other special cases of Satisfiability that are polynomial-time solvable. Most notably, XOR-SAT (clauses consist of exclusive-or operations and clauses are connected by and operations) is also linear-time solvable [Sch78]. We could only find a single reference ([Rad+07]) where a problem was reduced to XOR-SAT but there it was not used for a polynomial-time algorithm. We believe that *XOR-SAT programming* is also worth investigating as a potential tool for exact polynomial-time algorithms and we believe it to be most useful for problems in which the parity of numbers is important.

Bibliography

[ABH17] Elliot Anshelevich, Onkar Bhardwaj, and Martin Hoefer. “Stable matching with network externalities”. In: *Algorithmica* 78.3 (2017), pp. 1067–1106 (cited on p. 134).

[Abr+05] David J. Abraham, Katarína Cechlárová, David Manlove, and Kurt Mehlhorn. “Pareto optimality in house allocation problems”. In: *Proceedings of the 16th International Symposium on Algorithms and Computation (ISAAC ’05)*. Springer, 2005, pp. 1163–1175 (cited on p. 134).

[Ain+99] Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. “Fast estimation of diameter and shortest paths (without matrix multiplication)”. In: *SIAM Journal on Computing* 28.4 (1999), pp. 1167–1181 (cited on pp. 25, 27).

[AJB99] Réka Albert, Hawoong Jeong, and Albert-László Barabási. “Diameter of the world-wide web”. In: *Nature* 401.6749 (1999), pp. 130–131 (cited on p. 9).

[AK71] Jiří Adámek and Václav Koubek. “Remarks on flows in network with short paths”. In: *Commentationes Mathematicae Universitatis Carolinae* 12.4 (1971), pp. 661–667 (cited on p. 46).

[APT79] Bengt Aspvall, Michael F. Plass, and Robert E. Tarjan. “A linear-time algorithm for testing the truth of certain quantified boolean formulas”. In: *Information Processing Letters* 8.3 (1979), pp. 121–123 (cited on pp. 7, 180).

[Ari00] Rutherford Aris. *The Optimal Design of Chemical Reactors – A Study in Dynamic Programming*. Elsevier, 2000 (cited on p. 2).

[AV09] Esteban Arcaute and Sergei Vassilvitskii. “Social networks and stable matchings in the job market”. In: *Proceedings of the 5th Conference on Web and Internet Economics (WINE ’09)*. Springer, 2009, pp. 220–231 (cited on p. 134).

[AVW16] Amir Abboud, Virginia Vassilevska Williams, and Joshua R. Wang. "Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs". In: *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '16)*. SIAM, 2016, pp. 377–391 (cited on pp. 25, 27, 29, 32, 34).

[AWJ90] Amir A. Amini, Terry E. Weymouth, and Ramesh C. Jain. "Using dynamic programming for solving variational problems in vision". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12.9 (1990), pp. 855–867 (cited on p. 2).

[Bac+18] Arturs Backurs, Liam Roditty, Gilad Segal, Virginia Vassilevska Williams, and Nicole Wein. "Towards tight approximation bounds for graph diameter and eccentricities". In: *Proceedings of the 50th ACM Symposium on Theory of Computing (STOC '18)*. ACM, 2018, pp. 267–280 (cited on pp. 25, 29).

[Bai+10] Georg Baier, Thomas Erlebach, Alexander Hall, Ekkehard Köhler, Petr Kolman, Ondřej Pangrác, Heiko Schilling, and Martin Skutella. "Length-bounded cuts and flows". In: *ACM Transactions on Algorithms* 7.1 (2010), 4:1–4:27 (cited on pp. 46, 47).

[Baz+19] Cristina Bazgan, Till Fluschnik, André Nichterlein, Rolf Niedermeier, and Maximilian Stahlberg. "A more fine-grained complexity analysis of finding the most vital edges for undirected shortest paths". In: *Networks* 73.1 (2019), pp. 23–37 (cited on pp. 9, 46, 47, 60).

[BBN19] Matthias Bentert, René van Bevern, and Rolf Niedermeier. "Inductive k-independent graphs and c-colorable subgraphs in scheduling: a review". In: *Journal of Scheduling* 22.1 (2019), pp. 3–20 (cited on p. 125).

[BCM16] Sylvain Bouveret, Yann Chevaleyre, and Nicolas Maudet. "Fair allocation of indivisible goods". In: *Handbook of Computational Social Choice*. Cambridge University Press, 2016, pp. 284–310 (cited on p. 134).

[BCT17] Michele Borassi, Pierluigi Crescenzi, and Luca Trevisan. "An axiomatic and an average-case analysis of algorithms and heuristics for metric properties of graphs". In: *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '17)*. SIAM, 2017, pp. 920–939 (cited on p. 27).

[Bel52] Richard Bellman. "On the theory of dynamic programming". In: *Proceedings of the National Academy of Sciences of the United States of America* 38.8 (1952), p. 716 (cited on p. 2).

[Ben+19a] Matthias Bentert, Jiehua Chen, Vincent Froese, and Gerhard J. Woeginger. "Good things come to those who swap objects on paths". In: *Computing Research Repository* abs/1905.04219 (2019) (cited on pp. xi, 137).

[Ben+19b] Matthias Bentert, Till Fluschnik, André Nichterlein, and Rolf Niedermeier. “Parameterized aspects of triangle enumeration”. In: *Journal of Computer and System Sciences* 103.1 (2019), pp. 61–77 (cited on pp. 21, 22, 25).

[Ben+20] Matthias Bentert, Alexander Dittmann, Leon Kellerhals, André Nichterlein, and Rolf Niedermeier. “An adaptive version of Brandes’ algorithm for betweenness centrality”. In: *Journal of Graph Algorithms and Applications* 24.3 (2020), pp. 483–522 (cited on p. 25).

[Ben+21] Matthias Bentert, André Nichterlein, Malte Renken, and Philipp Zschoche. “Using a geometric lens to find k disjoint shortest paths”. In: *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP ’21).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 25:1–25:13. Full version available at https://arxiv.org/abs/2007.12502. (Cited on pp. x, 62).

[Bev+15] René van Bevern, Matthias Mnich, Rolf Niedermeier, and Mathias Weller. “Interval scheduling and colorful independent sets”. In: *Journal of Scheduling* 18.5 (2015), pp. 449–469 (cited on p. 125).

[Bey+19] Aurélie Beynier, Yann Chevaleyre, Laurent Gourvès, Ararat Harutyunyan, Julien Lesca, Nicolas Maudet, and Anaëlle Wilczynski. “Local envy-freeness in house allocation problems”. In: *Autonomous Agents and Multi-Agent Systems* 33.5 (2019), pp. 591–627 (cited on p. 134).

[BH19] Andreas Björklund and Thore Husfeldt. “Shortest two disjoint paths in polynomial time”. In: *SIAM Journal on Computing* 48.6 (2019), pp. 1698–1710 (cited on pp. 63, 100).

[BHK20] Matthias Bentert, Klaus Heeger, and Dušan Knop. “Length-bounded cuts: proper interval graphs and structural parameters”. In: *Proceedings of the 31st International Symposium on Algorithms and Computation (ISAAC ’20).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 36:1–36:14. Full version available at https://arxiv.org/abs/1910.03409. (Cited on pp. x, 48, 60).

[BHM00] Bernhard Beckert, Reiner Hähnle, and Felip Manyà. “The 2-SAT problem of regular signed CNF formulas”. In: *Proceedings of the 30th IEEE International Symposium on Multiple-Valued Logic ISMVL ’00.* IEEE Computer Society, 2000, pp. 331–336 (cited on p. 184).

[BHM20] Karl Bringmann, Thore Husfeldt, and Måns Magnusson. “Multivariate analysis of orthogonal range searching and graph distances”. In: *Algorithmica* 82.8 (2020), pp. 2292–2315 (cited on pp. 25, 27, 28).

[BK17] Kristóf Bérczi and Yusuke Kobayashi. “The directed disjoint shortest paths problem”. In: *Proceedings of the 25th Annual European Symposium on Algorithms (ESA ’17)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017, 13:1–13:13 (cited on p. 62).

[BKN18] Robert Bredereck, Andrzej Kaczmarczyk, and Rolf Niedermeier. “Envy-free allocations respecting social networks”. In: *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS ’18)*. International Foundation for Autonomous Agents and Multiagent Systems, 2018, pp. 283–291 (cited on p. 134).

[BLS99] Andreas Brandstädt, Van B. Le, and Jeremy P. Spinrad. *Graph Classes: A Survey*. SIAM, 1999 (cited on p. 16).

[BMW18] Matthias Bentert, Josef Malík, and Mathias Weller. “Tree containment with soft polytomies”. In: *Proceedings of the 16th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT ’18)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 9:1–9:14. Full version available at https://hal.archives-ouvertes.fr/hal-01734619. (Cited on pp. xi, 122, 124).

[BN19] Matthias Bentert and André Nichterlein. “Parameterized complexity of diameter”. In: *Proceedings of the 11th International on Algorithms and Complexity (CIAC ’19)*. Springer, 2019, pp. 50–61. Full version available at https://arxiv.org/abs/1802.10048. (Cited on pp. ix, 26–28).

[BNK20] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. “Machine learning for fluid mechanics”. In: *Annual Review of Fluid Mechanics* 52.1 (2020), pp. 477–508 (cited on p. 2).

[Bod88] Hans L. Bodlaender. “Dynamic programming on graphs with bounded treewidth”. In: *Proceedings of the 15th International Colloquium on Automata, Languages and Programming (ICALP ’88)*. Springer, 1988, pp. 105–118 (cited on p. 2).

[Bor+15] Michele Borassi, Pierluigi Crescenzi, Michel Habib, Walter A. Kosters, Andrea Marino, and Frank W. Takes. “Fast diameter and radius BFS-based computation in (weakly connected) real-world graphs: with an application to the six degrees of separation games”. In: *Theoretical Computer Science* 586.1 (2015), pp. 59–80 (cited on p. 27).

[Bre+08] Anna Bretscher, Derek G. Corneil, Michel Habib, and Christophe Paul. “A simple linear time LexBFS cograph recognition algorithm”. In: *SIAM Journal on Discrete Mathematics* 22.4 (2008), pp. 1277–1296 (cited on p. 34).

[BS16] Magnus Bordewich and Charles Semple. “Reticulation-visible networks”. In: *Advances in Applied Mathematics* 78.1 (2016), pp. 114–141 (cited on p. 106).

[BSW89] Andrew Gehret Barto, Richard S Sutton, and Christopher J. C. H. Watkins. *Learning and Sequential Decision Making*. University of Massachusetts Press, 1989 (cited on p. 2).

[Bui+87] Thang Nguyen Bui, Soma Chaudhuri, Frank Thomson Leighton, and Michael Sipser. “Graph bisection algorithms with good average case behavior”. In: *Combinatorica* 7.2 (1987), pp. 171–191 (cited on p. 35).

[Bul+16] Laurent Bulteau, Vincent Froese, Sepp Hartung, and Rolf Niedermeier. “Co-clustering under the maximum norm”. In: *Algorithms* 9.1 (2016), 17:1–17:17 (cited on pp. 6, 7).

[CCR13] Joseph M. Chan, Gunnar Carlsson, and Raul Rabadan. “Topology of viral evolution”. In: *Proceedings of the National Academy of Sciences* 110.46 (2013), pp. 18566–18571 (cited on p. 103).

[CGR16] Massimo Cairo, Roberto Grossi, and Romeo Rizzi. “New bounds for approximating extremal distances in undirected graphs”. In: *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’16)*. SIAM, 2016, pp. 363–376 (cited on p. 29).

[Che+01] Ting Chen, Ming-Yang Kao, Matthew Tepel, John Rush, and George M. Church. “A dynamic programming approach to de novo peptide sequencing via tandem mass spectrometry”. In: *Journal of Computational Biology* 8.3 (2001), pp. 325–337 (cited on p. 2).

[Che+05] Jianer Chen, Benny Chor, Mike Fellows, Xiuzhen Huang, David W. Juedes, Iyad A. Kanj, and Ge Xia. “Tight lower bounds for certain parameterized NP-hard problems”. In: *Information and Computation* 201.2 (2005), pp. 216–231 (cited on p. 20).

[Col+00] Richard Cole, Martin Farach-Colton, Ramesh Hariharan, Teresa Przytycka, and Mikkel Thorup. “An $O(n \log n)$ algorithm for the maximum agreement subtree problem for binary trees”. In: *SIAM Journal on Computing* 30.5 (2000), pp. 1385–1404 (cited on p. 109).

[Cor+01] Derek G. Corneil, Feodor F. Dragan, Michel Habib, and Christophe Paul. “Diameter determination on restricted graph families”. In: *Discrete Applied Mathematics* 113.2-3 (2001), pp. 143–166 (cited on pp. 25, 27).

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009 (cited on p. 2).

[CPS85] Derek G. Corneil, Yehoshua Perl, and Lorna K. Stewart. “A linear recognition algorithm for cographs”. In: *SIAM Journal on Computing* 14.4 (1985), pp. 926–934 (cited on p. 34).

[CW21] Timothy M. Chan and R. Ryan Williams. “Deterministic APSP, orthogonal vectors, and more: quickly derandomizing Razborov-Smolensky”. In: *ACM Transactions on Algorithms* 17.1 (2021), 2:1–2:14 (cited on p. 26).

[Cyg+15] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015 (cited on p. 19).

[DBC15] Anastasia Damamme, Aurélie Beynier, and Yann Chevaleyre. “The power of swap deals in distributed resource allocation”. In: *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS ’15)*. ACM, 2015, pp. 625–633 (cited on p. 134).

[DF13] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013 (cited on p. 19).

[Din06] Yefim Dinitz. “Dinitz’ algorithm: the original version and Even’s version”. In: *Theoretical Computer Science, Essays in Memory of Shimon Even*. Springer, 2006, pp. 218–240 (cited on p. 45).

[DK18] Pavel Dvořák and Dušan Knop. “Parameterized complexity of length-bounded cuts and multicuts”. In: *Algorithmica* 80.12 (2018), pp. 3597–3617 (cited on p. 47).

[Dre+12] Andreas W. M. Dress, Katharina T. Huber, Jacobus H. Koolen, Vincent Moulton, and Andreas Spillner. *Basic Phylogenetic Combinatorics*. Cambridge University Press, 2012 (cited on p. 110).

[ED16] Jacob Evald and Søren Dahlgaard. “Tight hardness results for distance and centrality problems in constant degree graphs”. In: *Computing Research Repository* abs/1609.08403 (2016) (cited on pp. 27, 28, 35).

[EHK03] Eleazar Eskin, Eran Halperin, and Richard M Karp. “Efficient reconstruction of haplotype structure via perfect phylogeny”. In: *Journal of Bioinformatics and Computational Biology* 1.1 (2003), pp. 1–20 (cited on pp. 6, 7).

[Eil98] Tali Eilam-Tzoreff. “The disjoint shortest paths problem”. In: *Discrete Applied Mathematics* 85.2 (1998), pp. 113–138 (cited on pp. 9, 62).

[FF56] Lester R. Ford and Delbert R. Fulkerson. “Maximal flow through a network”. In: *Canadian Journal of Mathematics* 8.3 (1956), pp. 399–404 (cited on pp. 45, 54).

[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006 (cited on p. 19).

[FHW80] Steven Fortune, John E. Hopcroft, and James Wyllie. “The directed subgraph homeomorphism problem”. In: *Theoretical Computer Science* 10.1 (1980), pp. 111–121 (cited on pp. 61, 62, 65, 77, 94).

[FKP15] Jittat Fakcharoenphol, Tanee Kumpijit, and Attakorn Putwattana. “A faster algorithm for the tree containment problem for binary nearly stable phylogenetic networks”. In: *Proceedings of the 12th International Joint Conference on Computer Science and Software Engineering (JCSSE ’15)*. IEEE, 2015, pp. 337–342 (cited on p. 106).

[Flu+18] Till Fluschnik, Danny Hermelin, André Nichterlein, and Rolf Niedermeier. “Fractals for kernelization lower bounds”. In: *SIAM Journal on Discrete Mathematics* 32.1 (2018), pp. 656–681 (cited on p. 47).

[Fom+19] Fedor V. Fomin, Dániel Marx, Saket Saurabh, and Meirav Zehavi. “New horizons in parameterized complexity (Dagstuhl Seminar 19041)”. In: *Dagstuhl Reports* 9.1 (2019), pp. 67–87 (cited on pp. 9, 62).

[FP80] Arthur M. Farley and Andrzej Proskurowski. “Computation of the center and diameter of outerplanar graphs”. In: *Discrete Applied Mathematics* 2.3 (1980), pp. 185–191 (cited on pp. 25, 27).

[FT97] Martin Farach and Mikkel Thorup. “Sparse dynamic programming for evolutionary-tree comparison”. In: *SIAM Journal on Computing* 26.1 (1997), pp. 210–230 (cited on p. 2).

[Gam+15] Philippe Gambette, Andreas D. M. Gunawan, Anthony Labarre, Stéphane Vialette, and Louxin Zhang. “Locating a tree in a phylogenetic network in quadratic time”. In: *Proceedings of the 19th Annual International Conference on Research in Computational Molecular Biology (RECOMB ’15)*. Springer, 2015, pp. 96–107 (cited on pp. 104, 106, 107).

[Gaw+21] Pawel Gawrychowski, Haim Kaplan, Shay Mozes, Micha Sharir, and Oren Weimann. “Voronoi diagrams on planar graphs, and computing the diameter in deterministic $\tilde{O}(n^{5/3})$ time”. In: *SIAM Journal on Computing* 50.2 (2021), pp. 509–554 (cited on pp. 25, 27).

[GDZ17] Andreas D. M. Gunawan, Bhaskar DasGupta, and Louxin Zhang. “A decomposition theorem and two algorithms for reticulation-visible networks”. In: *Information and Computation* 252.1 (2017), pp. 161–175 (cited on p. 106).

[GHN04] Jiong Guo, Falk Hüffner, and Rolf Niedermeier. “A structural view on parameterizing problems: distance from triviality”. In: *Proceedings of the 1st International Workshop on Parameterized and Exact Computation (IWPEC ’04)*. Springer, 2004, pp. 162–173 (cited on pp. 9, 25).

[GKW19] Marinus Gottschau, Marcus Kaiser, and Clara Waldmann. “The undirected two disjoint shortest paths problem”. In: *Operations Research Letters* 47.1 (2019), pp. 70–75 (cited on p. 62).

[GLW17] Laurent Gourvès, Julien Lesca, and Anaëlle Wilczynski. “Object allocation via swaps along a social network”. In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI ’17)*. ijcai.org, 2017, pp. 213–219 (cited on pp. 10, 11, 134–136, 174).

[GLZ16] Andreas D. M. Gunawan, Bingxin Lu, and Louxin Zhang. “A program for verification of phylogenetic network models”. In: *Bioinformatics* 32.17 (2016), pp. 503–510 (cited on pp. 106, 131).

[GMN17] Archontia C. Giannopoulou, George B. Mertzios, and Rolf Niedermeier. “Polynomial fixed-parameter algorithms: a case study for longest path on interval graphs”. In: *Theoretical Computer Science* 689.1 (2017), pp. 67–95 (cited on pp. 9, 19, 25, 27).

[GN02] M. Girvan and M. E. J. Newman. “Community structure in social and biological networks”. In: *Proceedings of the National Academy of Sciences* 99.12 (2002), pp. 7821–7826 (cited on p. 44).

[GPS08] Luís Gouveia, Pedro Patrício, and Amaro de Sousa. “Hop-constrained node survivable network design: an application to MPLS over WDM”. In: *Networks and Spatial Economics* 8.1 (2008), pp. 3–21 (cited on p. 46).

[Gro+19] Stephanie M. Groman, Bart Massi, Samuel R. Mathias, Daniel W. Curry, Daeyeol Lee, and Jane R. Taylor. “Neurochemical and behavioral dissections of decision-making in a rodent multistage task”. In: *Journal of Neuroscience* 39.2 (2019), pp. 295–306 (cited on p. 2).

[GT11] Petr A. Golovach and Dimitrios M. Thilikos. “Paths of bounded length and their cuts: parameterized complexity and algorithms”. In: *Discrete Optimization* 8.1 (2011), pp. 72–86 (cited on p. 47).

[Gun18] Andreas D. M. Gunawan. “Solving the tree containment problem for reticulation-visible networks in linear time”. In: *Proceedings of the 5th International Conference on Algorithms for Computational Biology (AlCoB ’18)*. Springer, 2018, pp. 24–36 (cited on p. 106).

[Gus14] Dan Gusfield. *ReCombinatorics: The Algorithmics of Ancestral Recombination Graphs and Explicit Phylogenetic Networks*. MIT Press, 2014 (cited on p. 103).

[GW09] Dan Gusfield and Yufeng Wu. “The three-state perfect phylogeny problem reduces to 2-SAT”. In: *Communications in Information & Systems* 9.4 (2009), pp. 295–302 (cited on pp. 6, 7).

[HHW03] Jurriaan Hage, Tero Harju, and Emo Welzl. “Euler graphs, triangle-free graphs and bipartite graphs in switching classes”. In: *Fundamenta Informaticae* 58.1 (2003), pp. 23–37 (cited on pp. 6, 7).

[HL00] Chính T. Hoàng and Van B. Le. “On P_4-transversals of perfect graphs”. In: *Discrete Mathematics* 216.1–3 (2000), pp. 195–210 (cited on pp. 6, 7).

[HR07] David Huygens and Ali Ridha Mahjoub. “Integer programming formulations for the two 4-hop-constrained paths problem”. In: *Networks* 49.2 (2007), pp. 135–144 (cited on p. 46).

[HRS10] Daniel H. Huson, Regula Rupp, and Celine Scornavacca. *Phylogenetic Networks: Concepts, Algorithms and Applications*. Cambridge University Press, 2010 (cited on p. 103).

[Huy+07] David Huygens, Martine Labbé, Ali Ridha Mahjoub, and Pierre Pesneau. “The two-edge connected hop-constrained network design problem: valid inequalities and branch-and-cut”. In: *Networks* 49.1 (2007), pp. 116–133 (cited on p. 46).

[HX20] Sen Huang and Mingyu Xiao. “Object reachability via swaps under strict and weak preferences”. In: *Autonomous Agents and Multi-Agent Systems* 34.2 (2020), p. 51 (cited on pp. xi, 6, 7, 11, 137, 154, 159, 180).

[IP01] Russell Impagliazzo and Ramamohan Paturi. “On the complexity of k-SAT”. In: *Journal of Computer and System Sciences* 62.2 (2001), pp. 367–375 (cited on pp. 19, 20).

[IP19] Ayumi Igarashi and Dominik Peters. “Pareto-optimal allocation of indivisible goods with connectivity constraints”. In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI ’19)*. AAAI Press, 2019, pp. 2045–2052 (cited on p. 134).

[ISS10] Leo van Iersel, Charles Semple, and Mike Steel. “Locating a tree in a phylogenetic network”. In: *Information Processing Letters* 110.23 (2010), pp. 1037–1043 (cited on pp. 10, 103, 106).

[Jan17] Bart M. P. Jansen. “On structural parameterizations of hitting set: hitting paths in graphs using 2-SAT”. In: *Journal of Graph Algorithms and Applications* 21.2 (2017), pp. 219–243 (cited on pp. 6, 7, 180, 184).

[Kan+08] Iyad A. Kanj, Luay Nakhleh, Cuong Than, and Ge Xia. “Seeing the trees and their branches in the network is hard”. In: *Theoretical Computer Science* 401.1-3 (2008), pp. 153–164 (cited on pp. 103, 106).

[Kar75] Richard M. Karp. “On the computational complexity of combinatorial problems”. In: *Networks* 5.1 (1975), pp. 45–68 (cited on pp. 20, 61).

[KKR12] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce A. Reed. “The disjoint paths problem in quadratic time”. In: *Journal of Combinatorial Theory, Series B* 102.2 (2012), pp. 424–435 (cited on p. 61).

[Kol18] Petr Kolman. “On algorithms employing treewidth for L-bounded cut problems”. In: *Journal of Graph Algorithms and Applications* 22.2 (2018), pp. 177–191 (cited on p. 47).

[KS06] Petr Kolman and Christian Scheideler. “Improved bounds for the unsplittable flow problem”. In: *Journal of Algorithms* 61.1 (2006), pp. 20–44 (cited on p. 47).

[KS19] Yusuke Kobayashi and Ryo Sako. “Two disjoint shortest paths problem with non-negative edge length”. In: *Operations Research Letters* 47.1 (2019), pp. 66–69 (cited on p. 62).

[LH08] Jure Leskovec and Eric Horvitz. “Planetary-scale views on a large instant-messaging network”. In: *Proceedings of the 17th International World Wide Web Conference (WWW ’08).* ACM, 2008, pp. 915–924 (cited on pp. 9, 26, 35).

[Loc21] William Lochet. “A polynomial time algorithm for the k-disjoint shortest paths problem”. In: *Proceedings of the 32nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’21).* SIAM, 2021 (cited on pp. x, 62, 65, 93).

[LY80] John M. Lewis and Mihalis Yannakakis. “The node-deletion problem for hereditary properties is NP-complete”. In: *Journal of Computer and System Sciences* 20.2 (1980), pp. 219–230 (cited on p. 35).

[LZ20] Bi Li and Xin Zhang. “Tree-coloring problems of bounded treewidth graphs”. In: *Journal of Combinatorial Optimization* 39.1 (2020), pp. 156–169 (cited on p. 2).

[Mar20] Dániel Marx. “Four shorts stories on surprising algorithmic uses of treewidth”. In: *Treewidth, Kernels, and Algorithms – Essays Dedicated to Hans L. Bodlaender on the Occasion of His 60th Birthday.* Springer, 2020, pp. 129–144 (cited on p. 2).

[MB20] Luis Müller and Matthias Bentert. “On reachable assignments in cycles and cliques”. In: *Computing Research Repository* abs/2005.02218 (2020) (cited on pp. 6, 7).

[Mil67] Stanley Milgram. “The small world problem”. In: *Psychology Today* 2.1 (1967), pp. 61–67 (cited on pp. 9, 26, 35).

[MM10] Ali Ridha Mahjoub and S. Thomas McCormick. “Max flow and min cut with bounded-length paths: complexity, algorithms, and approximation”. In: *Mathematical Programming* 124.1-2 (2010), pp. 271–284 (cited on pp. 46, 47).

[New03] Mark E. J. Newman. “The structure and function of complex networks”. In: *SIAM Review* 45.2 (2003), pp. 167–256 (cited on pp. 9, 25, 26, 35).

[New10] Mark E. J. Newman. *Networks: An Introduction*. Oxford University Press, 2010 (cited on pp. 26, 35).

[Nie06] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006 (cited on p. 19).

[NP03] Mark E. J. Newman and Juyong Park. "Why social networks are different from other types of networks". In: *Physical Review E* 68.3 (2003), p. 036122 (cited on pp. 26, 35).

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994 (cited on p. 18).

[Rad+07] Filippo Radicchi, Daniele Vilone, Sooeyon Yoon, and Hildegard Meyer-Ortmanns. "Social balance as a satisfiability problem of computer science". In: *Physical Review E* 75.2 (2007), p. 026106 (cited on p. 185).

[Rot82] Alvin E. Roth. "Incentive compatibility in a market with indivisible goods". In: *Economics Letters* 9.2 (1982), pp. 127–132 (cited on p. 134).

[RS95] Neil Robertson and Paul D. Seymour. "Graph minors. XIII. The disjoint paths problem". In: *Journal of Combinatorial Theory, Series B* 63.1 (1995), pp. 65–110 (cited on p. 61).

[RW13] Liam Roditty and Virginia Vassilevska Williams. "Fast approximation algorithms for the diameter and radius of sparse graphs". In: *Proceedings of the 45th ACM Symposium on Theory of Computing Conference (STOC '13)*. ACM, 2013, pp. 515–524 (cited on pp. 25, 26, 28, 29, 37, 40, 43).

[San00] David Sankoff. "The early introduction of dynamic programming into computational biology". In: *Bioinformatics* 16.1 (2000), pp. 41–47 (cited on p. 2).

[Sch03] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer Science & Business Media, 2003 (cited on p. 46).

[Sch19] Johannes C. B. Schröder. "Comparing Graph Parameters". 2019. Available under https://fpt.akt.tu-berlin.de/publications/theses/BA-Schröder.pdf (cited on p. 28).

[Sch78] Thomas J. Schaefer. "The complexity of satisfiability problems". In: *Proceedings of the 10th ACM Symposium on Theory of Computing (STOC '78)*. 1978, pp. 216–226 (cited on p. 185).

[Sei95] Raimund Seidel. "On the all-pairs-shortest-path problem in unweighted undirected graphs". In: *Journal of Computer and System Sciences* 51.3 (1995), pp. 400–403 (cited on p. 26).

[Ski20] Steven Skiena. *The Algorithm Design Manual, Third Edition*. Springer, 2020 (cited on p. 2).

[SS74] Lloyd Shapley and Herbert Scarf. “On cores and indivisibility”. In: *Journal of Mathematical Economics* 1.1 (1974), pp. 23–37 (cited on p. 133).

[SU10] Tayfun Sönmez and M. Utku Ünver. “House allocation with existing tenants: a characterization”. In: *Games and Economic Behavior* 69.2 (2010), pp. 425–445 (cited on p. 134).

[SW18] Abdallah Saffidine and Anaëlle Wilczynski. “Constrained swap dynamics over a social network in distributed resource reallocation”. In: *Proceedings of the 11th International Symposium on Algorithmic Game Theory (SAGT ’18)*. Springer, 2018, pp. 213–225 (cited on pp. 133, 137, 154, 180).

[TM90] Paolo Toth and Silvano Martello. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, 1990 (cited on pp. 3, 5).

[Tov84] Craig A. Tovey. “A simplified NP-complete satisfiability problem”. In: *Discrete Applied Mathematics* 8.1 (1984), pp. 85–89 (cited on p. 144).

[TR11] Todd J. Treangen and Eduardo P. C. Rocha. “Horizontal transfer, not duplication, drives the expansion of protein families in prokaryotes”. In: *PLoS Genetics* 7.1 (2011), e1001284 (cited on p. 103).

[TV96] Spyros Tragoudas and Yaakov L. Varol. “Computing disjoint paths with lenght constraints”. In: *Proceedings of the 22nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG ’96)*. Springer, 1996, pp. 375–389 (cited on p. 63).

[Wal15] Toby Walsh. “Challenges in resource and cost allocation”. In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI ’15)*. AAAI Press, 2015, pp. 4073–4077 (cited on p. 133).

[Wel18] Mathias Weller. “Linear-time tree containment in phylogenetic networks”. In: *Proceedings of the 16th RECOMB Comparative Genomics Satellite Conference (RECOMB-CG ’18)*. Springer, 2018, pp. 309–323 (cited on pp. 106, 131).

[WF94] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994 (cited on pp. 9, 25).

[WW95] Frank Wagner and Alexander Wolff. “Map labeling heuristics: provably good and practically useful”. In: *Proceedings of the 11th Annual Symposium on Computational Geometry (SoCG ’95)*. ACM, 1995, pp. 109–118 (cited on pp. 6, 7, 185).

[WY16] Oren Weimann and Raphael Yuster. “Approximating the diameter of planar graphs in near linear time”. In: *ACM Transactions on Algorithms* 12.1 (2016), 12:1–12:13 (cited on pp. 25, 29).

[Zwi01] Uri Zwick. “Exact and approximate distances in graphs – a survey”. In: *Proceedings of the 9th Annual European Symposium on Algorithms (ESA ’01)*. Springer, 2001, pp. 33–48 (cited on p. 9).

Universitätsverlag der TU Berlin

Elements of Dynamic and 2-SAT Programming: Paths, Trees, and Cuts

This thesis presents faster (in terms of worst-case running times and compared to the fastest previously known) exact algorithms for special cases of graph problems through dynamic programming and 2-SAT programming. Dynamic programming describes the procedure of breaking down a problem recursively into overlapping subproblems and then combining optimal solutions for these subproblems to an optimal solution for the larger problem. 2-SAT programming refers to the procedure of reducing a problem to a set of 2-SAT formulas, that is, Boolean formulas in conjunctive normal form in which each clause contains at most two literals. Computing a satisfying truth assignment (if one exists) of a 2-SAT formula takes linear time in the formula length. Hence, when satisfying truth assignments to some 2-SAT formulas correspond to a solution of the original problem and all formulas can be computed in polynomial time, then the original problem can be solved in polynomial time. Our main results are polynomial-time algorithms for special graph classes and parameterized algorithms.

ISBN 978-3-7983-3209-6 (print)
ISBN 978-3-7983-3210-2 (online)

ISBN 978-3-7983-3209-6

https://verlag.tu-berlin.de