Foundations of Computing Volume 1

René van Bevern

Fixed-Parameter Linear-Time Algorithms for NP-hard Graph and Hypergraph Problems Arising in Industrial Applications

Universitätsverlag der TU Berlin

Technische Universität Berlin

René van Bevern

Fixed-Parameter Linear-Time Algorithms for NP-hard Graph and Hypergraph Problems Arising in Industrial Applications

Die Schriftenreihe *Foundations of Computing* der Technischen Universität Berlin wird herausgegeben von:
Prof. Dr. Stephan Kreutzer,
Prof. Dr. Uwe Nestmann,
Prof. Dr. Rolf Niedermeier

Foundations of Computing | 01

René van Bevern

Fixed-Parameter Linear-Time Algorithms for NP-hard Graph and Hypergraph Problems Arising in Industrial Applications

Universitätsverlag der TU Berlin

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Universitätsverlag der TU Berlin 2014
http://www.univerlag.tu-berlin.de

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

Zugl.: Berlin, Technische Universität, Diss., 2014
1. Gutachter: Prof. Dr. Rolf Niedermeier
2. Gutachter: Prof. Dr. Fedor V. Fomin
3. Gutachter: Prof. Dr. Rolf Möhring
Die Arbeit wurde am 17. Juni 2014 an der Fakultät IV unter Vorsitz von Prof. Anja Feldmann, Ph. D. erfolgreich verteidigt.

Druck: docupoint GmbH
Satz/Layout: René van Bevern
Umschlagfoto: mr172: Stahlwerk Hennigsdorf
https://www.flickr.com/photos/mr172/5461921016/

ISBN 978-3-7983-2705-4 (print)
ISBN 978-3-7983-2706-1 (online)

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

Zugleich online veröffentlicht auf dem Digitalen Repositorium der Technischen Universität Berlin:
URL http://opus4.kobv.de/opus4-tuberlin/frontdoor/index/index/docId/5529
URN urn:nbn:de:kobv:83-opus4-55292
[http://nbn-resolving.de/urn:nbn:de:kobv:83-opus4-55292]

Zusammenfassung

Gegenstand dieser Dissertation ist die Frage, inwieweit große Instanzen NP-schwerer Graphen- und Hypergraphenprobleme, die in industriellen Anwendungen auftreten, optimal lösbar sind. Eine in der Komplexitätstheorie weithin akzeptierte Annahme ist, dass NP-schwere Probleme nicht optimal in einer Zeit lösbar sind, die polynomiell mit der Eingabegröße wächst. Der in dieser Arbeit untersuchte Ansatz, NP-schwere Probleme dennoch effizient zu lösen, sind Festparameter-Linearzeitalgorithmen – Algorithmen, welche NP-schwere Probleme in linear mit der Eingabegröße wachsender Zeit lösen, wenn bestimmte Parameter der Eingabeinstanz konstant sind. Genauer betrachten wir die folgenden Probleme, präsentieren Beispielanwendungen und entwickeln Festparameter-Linearzeitalgorithmen zu ihrer optimalen Lösung. In allen außer einem Fall erproben wir die entwickelten Algorithmen experimentell.

2-Union Independent Set ist das Problem, eine Menge mindestens k paarweise nichtbenachbarter Knoten in einem Graphen zu finden, der die kantenweise Vereinigung zweier Intervallgraphen ist. Ein Spezialfall von 2-Union Independent Set ist Job Interval Selection; hier ist der Eingabegraph die kantenweise Vereinigung eines Intervallgraphen und eines Clustergraphen. Beide Probleme modellieren zahlreiche Ablaufplanungsaufgaben.

Wir geben eine vereinheitlichte Sicht auf beide Probleme, die auf einem allgemeineren Problem über knotengefärbten Intervallgraphen basiert. Ferner rücken wir das Augenmerk parametrisierter Komplexitätsbetrachtungen auf den Intervallgraphenparameter „Kompaktheit“: ein Intervallgraph ist c-kompakt, wenn er eine Intervalldarstellung über den natürlichen Zahlen $\{1, \ldots, c\}$ hat.

Wir zeigen, dass Job Interval Selection in $O(2^k k \cdot n)$ Zeit lösbar ist; hierbei ist n die Anzahl der Intervalle in den beiden Eingabeintervallgraphen. Ferner zeigen wir, dass 2-Union Independent Set in $O(2^c c \cdot n)$ Zeit lösbar ist, wenn einer der beiden Eingabeintervallgraphen c-kompakt ist. Experimente zeigen, dass Instanzen mit 10^5 Knoten und $c \leq 15$ in weniger als fünf Minuten gelöst werden können.

Wir verstärken bekannte NP-Schwereergebnisse für beide Probleme und zeigen die Möglichkeiten und Grenzen von Polynomialzeitdatenreduktion.

DAG PARTITIONING ist das Problem, Kanten mit einem Gesamtgewicht von höchstens k aus einem kantengewichteten gerichteten azyklischen Graphen zu löschen, sodass im resultierenden Graphen jede schwache Zusammenhangskomponente nur eine Senke (einen Knoten ohne ausgehende Kanten) enthält. DAG PARTITIONING modelliert die Aufgabe, Varianten kurzer Wortgruppen entsprechend ihrer Ursprünge zu gruppieren.

Wir entwickeln einen Algorithmus, der DAG PARTITIONING in $O(2^k \cdot (n+m))$ Zeit auf Eingabegraphen mit n Knoten und m Kanten löst. Wir ergänzen den Algorithmus um Linearzeitdatenreduktionsalgorithmen. Experimente zeigen, dass der Algorithmus in Kombination mit der Datenreduktion Instanzen mit 10^7 Kanten und $k \leq 190$ in weniger als fünf Minuten löst. Die gewonnenen optimalen Lösungen benutzen wir zur Bewertung der Qualität bereits bekannter heuristischer Lösungsalgorithmen.

Wir verstärken bekannte NP-Schwereergebnisse und ergänzen unsere algorithmischen Resultate um untere Schranken für die Laufzeit exakter Lösungsalgorithmen und für die Wirksamkeit von Polynomialzeitdatenreduktion.

d-HITTING SET ist das Problem, höchstens k Knoten eines Hypergraphen so auszuwählen, dass jede Hyperkante mindestens einen ausgewählten Knoten enthält. Hierbei haben alle Hyperkanten eine Kardinalität von höchstens d. Die Anwendungen von d-HITTING SET liegen in der automatischen Programmverifikation und in der rauschminimierenden Zuordnung von Sendefrequenzen zu Sendern.

Wir zeigen einen Linearzeitalgorithmus, der eine d-HITTING SET-Instanz zu einer äquivalenten Instanz mit $O(k^d)$ Hyperkanten – zu einem sogenannten *Problemkern* – reduziert. Ein beliebiger der zahlreichen bekannten Lösungsalgorithmen für d-HITTING SET kann anschließend auf die geschrumpfte Instanz angewendet werden. Experimente zeigen, dass Hypergraphen mit 10^7 Hyperkanten in weniger als fünf Minuten reduziert werden. Wir zeigen außerdem, wie die Anzahl der Knoten im resultierenden Hypergraphen mit $O(k^{1.5d})$ zusätzlicher Zeit auf $O(k^{d-1})$ reduziert werden kann.

HYPERGRAPH CUTWIDTH ist das Problem, die Knoten eines Hypergraphen so auf der reellen Zahlenachse anzuordnen, dass für jedes $i \in \mathbb{R}$ höchstens k Hyperkanten je einen Knoten mit Position größer als i und kleiner als i enthalten.

Das Problem hat Anwendungen im automatischen Testen elektronischer Schaltkreise. Wir zeigen einen Algorithmus, der HYPERGRAPH CUTWIDTH in linearer Zeit für konstantes k löst. Obwohl der Algorithmus vermutlich zu technisch für praktische Anwendungen ist, ist die zur Gewinnung des Algorithmus entwickelte Methode geeignet, um generell Hypergraphenprobleme als linearzeitlösbar für feste Parameter zu klassifizieren und damit die Richtung für weitergehende Forschung an einem Problem zu weisen.

Abstract

This thesis investigates the question to which extent large instances of NP-hard graph and hypergraph problems arising in industrial applications can be solved optimally. A widely accepted conjecture in computational complexity theory is that NP-hard problems cannot be solved optimally in a time growing polynomially in the input instance size. The approach to overcome this difficulty investigated in this thesis are fixed-parameter linear-time algorithms, which run in time linear in the input size when certain parameters of the input instances are constant. More precisely, we consider the following problems, give examples for their applications, and develop fixed-parameter linear-time algorithms to solve them optimally. In all but one case, we evaluate our algorithms experimentally.

2-UNION INDEPENDENT SET is the problem of finding a set of at least k pairwise nonadjacent vertices in a graph that is the edge-wise union of two interval graphs. In its special case JOB INTERVAL SELECTION, the input graph is the edge-wise union of an interval graph and a cluster graph. Both problems model several scheduling tasks.

We give a unified view on the two problems in form of a more general problem on vertex-colored interval graphs. Moreover, we bring the "compactness" interval graph parameter into parameterized complexity considerations: an interval graph is c-compact if it has an interval representation using only the numbers $\{1, \ldots, c\}$.

We show that JOB INTERVAL SELECTION is solvable in $O(2^k k \cdot n)$ time, where n is the number of intervals in the two input interval graphs. Moreover, we show that 2-UNION INDEPENDENT SET is solvable in $O(2^c c \cdot n)$ time if one of the two input interval graphs is c-compact. Experimental results show that instances with 10^5 vertices and $c \leq 15$ are solvable in less than five minutes.

We strengthen known NP-hardness results for both problems and chart the possibilities of polynomial-time data reduction.

DAG PARTITIONING is the problem of deleting a set of arcs of total weight at most k from an arc-weighted directed acyclic graph so that, in the resulting

directed acyclic graph, each weakly connected component has at most one sink—a vertex with outdegree zero. DAG PARTITIONING models the task of clustering variations of short phrases according to their origins.

We provide an algorithm to solve DAG PARTITIONING in $O(2^k \cdot (n+m))$ time on input graphs with n vertices and m arcs. We complement it with linear-time executable data reduction rules. Experiments show that, in combination, they can solve instances with 10^7 input arcs and $k \leq 190$ in less than five minutes. We use the obtained optimal solutions to evaluate the quality of previously known heuristics for the problem.

We strengthen known NP-hardness results and complement our algorithmic results with lower bounds on the running time of exact algorithms and on the effectiveness of polynomial-time data reduction.

The d-HITTING SET problem is the task of selecting at most k vertices of a hypergraph so that each hyperedge, all of which have cardinality at most d, contains at least one selected vertex. The applications of d-HITTING SET are in automatic program verification and in the noise-minimizing assignment of frequencies to radio transmitters.

We present an algorithm that, in linear time, transforms a d-HITTING SET instance into an equivalent instance comprising $O(k^d)$ hyperedges and vertices—a so-called *problem kernel*. Any of several known algorithms for d-HITTING SET can then be applied to the equivalent shrunken instance. Experiments show that hypergraphs with 10^7 input hyperedges can be kernelized in less than five minutes. We show how to shrink the number of vertices to $O(k^{d-1})$ using $O(k^{1.5d})$ additional time.

HYPERGRAPH CUTWIDTH is the problem of positioning the vertices of a hypergraph on the real line so that, for any $i \in \mathbb{R}$, there are at most k hyperedges containing vertices with positions less than as well as greater than i. The problem has applications in the automatic testing of electronic circuits. We present an algorithm that solves HYPERGRAPH CUTWIDTH in linear time for constant k. While the algorithm might be too technical to implement it in practical applications, the technique that we developed to obtain the algorithm can generally be used to classify hypergraph problems as solvable in fixed-parameter linear-time and, thus, to lead the way to future research.

Preface

This thesis summarizes parts of my studies in the field of fixed-parameter algorithms between May 2010 and January 2011 at Friedrich-Schiller-Universität Jena, Germany, and between January 2011 and February 2014 at Technische Universität Berlin. During the entire time, I have been working in the research group of Rolf Niedermeier, who moved from Jena to Berlin in October 2010. I was funded by the Deutsche Forschungsgemeinschaft (DFG) from May 2010 till February 2012 under the project "Algorithms for generating quasi-regular structures in graphs" (AREG, NI 369/9) and from February 2012 till February 2014 under the project "Data-driven parameterized algorithmics for graph modification problems" (DAPA, NI 369/12).

During this time, besides working on this thesis, I have been involved in algorithmics for NP-hard problems related to the fields of data clustering [BFSS14; BKMN10; BMN12], bioinformatics [Bet+11], machine learning [FBNS13], graph theory [Bev+14a; Bev+14b], linear-time data reduction [Bev+12], and arc routing [BHNS14; SBNW11; SBNW12]. Our efforts in the field of arc routing led to a chapter in a new arc routing book [BNSW14].

This thesis revolves around a topic that I have been following with increased interest only since 2011: linear-time data reduction and fixed-parameter linear-time algorithms for graph and hypergraph problems.

Besides two introductory and one concluding chapter, this thesis comprises four main chapters—Chapter 3 to Chapter 6. These present results, preliminary versions of which have been published at conferences [Bev+13; BFGR13; BMNW12] and in journals [Bev14]. The conference publications are the result of close and fruitful collaboration with my coauthors. In the following, I will go into details regarding the differences between the chapters as presented in this thesis and the preliminary versions published at conferences and in journals. Moreover, I will elaborate on my own contributions to the presented results.

Chapter 3 studies the Independent Set problem on edge-wise unions of two interval graphs. Herein, the task is to find a set of at least k pairwise nonadjacent vertices. Studying this problem was one of the goals of the research project "Algo-

rithms for generating quasi-regular structures in graphs," which I continued after Hannes Moser left our group in February 2010. I studied the problem together with my office mate Mathias Weller, our short-term guest Matthias Mnich, and Rolf Niedermeier. The results of our work were published at the *23rd International Symposium on Algorithms and Computation (ISAAC'12)* [BMNW12]. My main contributions to this publication were a complexity dichotomy with respect to the considered input graph classes, the ideas leading to the problem kernel with respect to the parameter "number of maximal cliques" and the very idea of considering this graph parameter, which was the crucial point of attack in most of our positive results.

Chapter 3 strongly differs from the conference publication, mainly due to the insights gained from an experimental evaluation of our algorithms. These led to simplified proofs and fundamental conceptual changes: while implementing the dynamic programming algorithm from our conference publication, I noticed how to improve the algorithm to run in linear time when the number of maximal cliques of one of the two input interval graphs is constant. Moreover, in order to execute experiments, it became necessary to reduce its space requirements. While reducing its space requirements, I noticed that the algorithm runs in linear time even when a smaller structural parameter is constant. After multiple iterations of testing and simplifying the algorithm, I arrived at an algorithm resembling that of Halldórsson and Karlsson [HK06] for JOB INTERVAL SELECTION, which is INDEPENDENT SET on the edge-wise unions of one interval graph and one cluster graph. That is, JOB INTERVAL SELECTION is a special case of our INDEPENDENT SET variant.

The simplified dynamic program now suggested that a viewpoint closer to the interval structure, as opposed to the graph structure, helps better understanding the graph parameters exploited in our work as well as simplifying many of our proofs. This led to fundamental conceptual changes of the work. For example, while the conference version relied on the parameter "number of maximal cliques" in an interval graph, the chapter in this thesis relies on the parameter "number of numbers required to represent all intervals," which I termed *compactness*. The improvements over the conference article have been accepted for publication in the *Journal of Scheduling* [BMNW14].

Chapter 4 analyzes the parameterized complexity of the DAG PARTITIONING problem. Herein, the task is to remove at most k arcs from a directed acyclic graph so that each remaining weakly connected component contains at most one sink. The problem came to my attention due to my coauthor Falk Hüffner, who sug-

gested investigating its parameterized complexity at our annual group-internal workshop, which in March 2012 took place on the Baltic sea island Usedom. After group discussions with the workshop participants Robert Bredereck, Sepp Hartung, Falk Hüffner, and André Nichterlein, we found NP-hardness results for many restricted versions of DAG PARTITIONING and got an idea on how to solve the problem on trees. Later, back in Berlin, Morgan Chopin and Ondřej Suchý joined our discussion group and Ondřej brought the major breakthrough on the algorithmic front: he found DAG PARTITIONING to be fixed-parameter tractable with respect to k using a simple search tree algorithm and with respect to treewidth using a very technical dynamic programming algorithm. The results of our joint work were published at the *8th International Conference on Algorithms and Complexity (CIAC'13)* [Bev+13].

For the thesis, I implemented the simple search tree algorithm and evaluated it experimentally not only with the goal of evaluating its practical applicability, but also to evaluate the quality of the heuristic that Leskovec, Backstrom, and Kleinberg [LBK09] earlier developed for the problem. In order to be able to solve larger problem instances, I developed linear-time executable data reduction rules for DAG PARTITIONING and finally replaced the search tree algorithm of the conference version by a new one, which is more technical but runs in linear instead of quadratic time for fixed k.

In contrast to the conference version, Chapter 4 does not contain the dynamic programming algorithm for graphs on bounded treewidth, since I consider the algorithm to be too technical in view of its very limited applicability: including all correctness proofs, it is 15 pages long and its running time is $2^{O(t^2)} \cdot n$ when a width-t tree decomposition of the n-vertex input graph is given. Instead, Chapter 4 only contains a linear-time algorithm on trees.

Chapter 5 considers d-HITTING SET—a variant of the well-known VERTEX COVER problem. Herein, the task is to select at most k vertices of a hypergraph so that each hyperedge, all of which have cardinality at most d, contains a selected vertex. Niedermeier and Rossmanith [NR03] claimed that a problem kernel of size $O(k^d)$ for d-HITTING SET is computable in linear time. However, the paper gives no proof of the running time and the proof of the running time for the case $d = 3$ is incomplete.

Since I had already gathered some experience with the linear-time computation of problem kernels [Bev+12], Rolf Niedermeier approached me with the following words in summer 2011 after one of his parameterized complexity lectures: "René, I just told the students that a problem kernel for d-HITTING SET can be computed

in linear time, but I do not see how." Thus, he got me interested in computing a problem kernel for d-HITTING SET in linear time. I decided that doing so would be interesting using the sunflower technique, which yields problem kernels that have desired properties in theoretical applications [FSV13; Kra12]. The results were first published at the *18th Annual International Conference on Computing and Combinatorics (COCOON'12)* [Bev12].

The conference paper was invited to the *COCOON'12* special issue of *Algorithmica*, for which I implemented and experimentally evaluated the kernelization algorithm [Bev14]. Unfortunately, the implementation contained a bug that did not occur on small test instances, but in the large-scale instances used in the experiments. As a result, while the running times observed in the *Algorithmica* article are faithful, the observed problem kernel sizes in the *Algorithmica* article are incorrect. Chapter 5 is based on the *Algorithmica* article, but provides experiments with a corrected implementation.

Chapter 6 generalizes the Myhill-Nerode theorem from formal language theory to hypergraphs and, as a result, provides a general technique to obtain fixed-parameter linear-time algorithms for hypergraph problems on hypergraphs with incidence graphs of bounded treewidth.

The work presented in Chapter 6 was initiated when Serge Gaspers (University of New South Wales, Sydney) and I visited Michael R. Fellows and Frances A. Rosamond at Charles Darwin University in Darwin, Australia, in August 2012. The goal of our mini-workshop was far from generalizing the Myhill-Nerode theorem to hypergraphs: in fact, Michael suggested looking for some interesting problem from coding theory to be solved using fixed-parameter algorithms. A challenging open problem was quickly found: finding the permutation of the coordinates of a linear code that minimizes its decoding complexity (the problem is known as TRELLIS WIDTH [Kas08]). We thought we had found a reduction of the problem to HYPERGRAPH CUTWIDTH, where the task is to position the vertices of a hypergraph on the real line so that, for any $i \in \mathbb{R}$, there are at most k hyperedges containing vertices with positions less than as well as greater than i. Thus, we focused on solving HYPERGRAPH CUTWIDTH.

Michael assumed that HYPERGRAPH CUTWIDTH can be solved using a Myhill-Nerode approach, which was successfully applied to CUTWIDTH on ordinary graphs [DF13, Section 12.7]. Having returned from Australia, I worked out the generalization of the Myhill-Nerode theorem to hypergraphs and the details of the proof that HYPERGRAPH CUTWIDTH is solvable in linear time for

fixed k. Serge applied the Myhill-Nerode theorem to prove negative results for GENERALIZED HYPERTREE WIDTH.

It later turned out that our reduction from TRELLIS WIDTH to HYPERGRAPH CUTWIDTH was wrong. Thus, what remained of our meeting were the generalization of the Myhill-Nerode theorem to hypergraphs and the results on HYPERGRAPH CUTWIDTH and GENERALIZED HYPERTREE WIDTH, which were presented at the *24th International Symposium on Algorithms and Computation (ISAAC'13)* [BFGR13]. Our work was invited to the *ISAAC'13* special issue of *Algorithmica*. For the journal version, which is still under review, I reworked large parts of our article in order to make the Myhill-Nerode technique easier accessible to a more algorithm-oriented audience [BFGR14]. The thesis chapter is based on the article prepared for *Algorithmica*, but it skips the proof of the results on GENERALIZED HYPERTREE WIDTH.

Acknowledgments. I am thankful to Rolf Niedermeier, who shaped my academic life: his lecture on discrete mathematics and logic was the most interesting lecture I got to follow as a first semester student at Friedrich-Schiller-Universität Jena in 2005. At the end of 2006, he hired me as a student research assistant, which led me to my first conference visit—the *33rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG'07)* in Jena. In 2009, he supervised my student research project, in 2010 my Diploma thesis, and till 2014 my work as a PhD student.

I thank all of my current and former colleagues Nadja Betzler, Robert Bredereck, Laurent Bulteau, Jiehua Chen, Michael Dom, Vincent Froese, Jiong Guo, Sepp Hartung, Falk Hüffner, Christian Komusiewicz, Stefan Kratsch, Hannes Moser, André Nichterlein, Manuel Sorge, Ondřej Suchý, Nimrod Talmon, Johannes Uhlmann, Mathias Weller, and Gerhard J. Woeginger for a good working climate and fruitful discussions. Moreover, it was pleasant to work with my group-external coauthors Morgan Chopin, Andreas Emil Feldmann, Michael R. Fellows, Serge Gaspers, Frank Kammer, Matthias Mnich, and Frances A. Rosamond.

I thank Fedor V. Fomin and Rolf Möhring for reviewing this thesis; "Благодарю" and "Vielen Dank."

Contents

1 Introduction

The aim of this thesis is exploring to which extent fixed-parameter algorithms can *optimally* solve large instances of NP-hard graph and hypergraph problems arising in industrial applications. Currently, many NP-hard graph and hypergraph problems are solved heuristically instead of optimally; the reason is that NP-hard problems resisted all tries of solving them in polynomial time since the time of their discovery by Cook [Coo71] and Levin [Lev73].

For example, consider the following four problems, which will be subject to the studies in this thesis and will be formally defined later.

2-Union Independent Set. Using a heuristic for 2-UNION INDEPENDENT SET, Höhn et al. [HKML11] optimized the production plans of a major German steel producer, lowering their production times by about 13%. Using lower bounds on the time required to produce the requested items, Höhn et al. [HKML11] showed that their heuristic solved the instances at hand to within 10% of the optimum. A gentle introduction to the application of 2-UNION INDEPENDENT SET in steel manufacturing and a description of the modeling problems that had to be mastered is given by Möhring [Möh11].

DAG Partitioning. Suen et al. [Sue+13] presented NIFTY, a system that allows for near real-time observation of the rise and fall of trends, ideas, and topics on the internet, which is of interest to the advertising industry and news media. A core component of NIFTY is a heuristic for the DAG PARTITIONING problem. It is used to cluster short phrases, which may undergo modifications while propagating through the web, with respect to their origins. Suen et al. [Sue+13] stated that, at the time of writing, the DAG PARTITIONING heuristic implemented in NIFTY clustered about $8.5 \cdot 10^6$ phrases per day. DAG PARTITIONING was introduced for this very purpose by Leskovec, Backstrom, and Kleinberg [LBK09], their work being featured in the New York Times [Loh09].

***d*-Hitting Set.** As one of the initial 21 NP-complete problems discovered by Karp [Kar72], d-HITTING SET is a problem so basic that it implicitly occurs in many tasks, among others, in bioinformatics [MK13].

O'Callahan and Choi [OC03] used a heuristic for d-HITTING SET in order to automatically detect bugs in parallel Java programs while aiming for a small slowdown of the program monitored at execution time. As a result, they discovered bugs in widespread web application servers like Apache Tomcat—even bugs that might have led to data corruption.

Sorge et al. [SMNW14] showed how d-HITTING SET can be used to compute a maximum set of numbers in $\{1,\dots,n\}$ such that no pair of numbers has the same distance as another pair. Such sets are known as Golomb rulers [Dra09] or Sidon sets [ET41]. As observed by Babcock [Bab53], noise that one radio transmitter causes on frequencies of other radio transmitters is avoided when choosing the frequencies assigned to the radio transmitters in form of a Golomb ruler.

Hypergraph Cutwidth. Prasad, Chong, and Keutzer [PCK99] investigated the question why CNF-SAT—the NP-hard problem of deciding whether a Boolean formula in conjunctive normal form is satisfiable—is often easily solvable for formulas occurring in automatic fault-testing of digital hardware, that is, of Boolean circuits. They empirically discovered that many practically occurring Boolean circuits, when transformed into hypergraphs, have small hypergraph cutwidth. Moreover, the authors proved that the CNF-SAT formulas obtained from Boolean circuits with logarithmic hypergraph cutwidth can be solved in polynomial time.

Similarly, Wang et al. [WCZK01] presented an algorithm for CNF-SAT whose running time grows linearly with the number of variables and exponentially only with the cutwidth of a hypergraph obtained from the input formula. Specifically, they speeded up a classical algorithm (known as DPLL) that still lies at the heart of modern industry-strength CNF-SAT solvers like, for example, the winner of the 2013 SAT competition [Bie13]. They experimentally evaluated their algorithm using a heuristic for HYPERGRAPH CUTWIDTH.

While heuristic algorithms solve the described NP-hard problems satisfactory in their applications, the goal of this thesis is solving them optimally. Optimum solutions are desirable for at least two reasons:

Resource economy. Being able to quickly compute optimal solutions for 2-UNION INDEPENDENT SET, Höhn et al. [HKML11] could possibly further increase the productivity of the steel production line. By solving d-HITTING SET optimally, one could make more effective use of the radio spectrum [Bab53; SMNW14] or more efficiently detect bugs in parallel programs at execution time [OC03]. Optimum solutions for HYPERGRAPH CUTWIDTH could enable the CNF-SAT algorithms by Prasad, Chong, and Keutzer [PCK99] and Wang et al. [WCZK01] to solve previously unsolvable instances, especially those arising in fault testing of digital hardware.

Model validation. As contradictory as it might sound, but not in all scenarios optimal solutions are really *better*: it is a priori unclear, for example, whether optimal solutions for DAG PARTITIONING would enable Leskovec, Backstrom, and Kleinberg [LBK09] and Suen et al. [Sue+13] to obtain phrase clusterings of higher quality. However, if better solutions for DAG PARTITIONING yielded worse clusterings, this would indicate that DAG PARTITIONING insufficiently models the actual clustering task.

The amount of data that has to be handled in applications, like the $8.5 \cdot 10^6$ phrases that Suen et al. [Sue+13] clustered per day using DAG PARTITIONING, creates problems in optimally solving NP-hard problems, which presumably cannot be solved in time polynomial in the input size. Suen et al. [Sue+13] already consider a quadratic running time "prohibitively large" for their application of DAG PARTITIONING. In the following, we describe the algorithmic framework that shall help us to overcome these difficulties.

1.1 Fixed-parameter linear-time algorithms

NP-hard problems presumably cannot be solved in time that grows polynomially in the size of the input. That is, all known algorithms for NP-hard problems have running times that grow faster in the input size than any polynomial. A comprehensive introduction into the concept of NP-hardness is given in the book of Garey and Johnson [GJ79].

In applications, where the size of the input data can be so large that even a quadratic running time is prohibitively large, it is usually not worthwhile even thinking about solving the problem using an algorithm that has a running time exponential in the input size. The trick is to measure the running time

of algorithms not only in the size of the input, but in additional parameters. This leads to the notion of *fixed-parameter algorithms* pioneered by Downey and Fellows [DF99]—algorithms whose running time grows polynomially in the input size and exponentially only in a smaller *parameter* k. Problems allowing for such algorithms are called *fixed-parameter tractable* with respect to a parameter k.

A main challenge in the search for fixed-parameter algorithms is identifying parameters that are small in practical applications and with respect to which a problem is fixed-parameter tractable [FJR13; KN12; Nie10]. In order to stick to our running examples: Wang et al. [WCZK01] provided an algorithm for CNF-SAT whose running time scales linearly in the number of variables in the input formula and exponentially in its cutwidth.

While the field of fixed-parameter algorithms got enthusiastically involved into races for algorithms with the lowest running time dependence on the parameter and for the smallest parameters with respect to which a problem is fixed-parameter tractable, the field partly started neglecting the growth of the running time in dependence of the input size. As pointed out by Komusiewicz and Niedermeier [KN12], it was satisfied with this growth being merely polynomial.

The aim of this thesis is to steer in the opposite direction by also focusing on the growth of the running time in dependence of the input size: while we do not expect fixed-parameter algorithms to be competitive in running time with heuristics currently used to solve NP-hard problems, we investigate the question to which extent large input instances can be solved *optimally* using fixed-parameter algorithms. To this end, we search for *fixed-parameter linear-time* algorithms—algorithms whose running time grows linearly in the input size and exponentially only in a parameter k. We conduct experiments in order to evaluate to which extent our fixed-parameter linear-time algorithms are applicable to large input instances.

An important building step in fixed-parameter linear-time algorithms is linear-time data reduction, a notion motivated in Section 1.2.

1.2 Linear-time kernelization

A subfield of fixed-parameter algorithmics and a powerful approach to solve NP-hard problems more efficiently is polynomial-time data reduction [Bod09; GN07; Kra14]. Intuitively, we want to solve and remove polynomial-time solvable parts of a problem instance so that only a small hard *kernel* of the instance remains to be solved by an exponential-time algorithm. Moreover, we want the

resulting kernel to be equivalent to the input instance in a sense that, from an optimal solution for the kernel, we want to recover an optimal solution for the original instance.

This leads to the notion of *problem kernelization*—a concept also pioneered by Downey and Fellows [DF99] and which we will define formally in Chapter 2: in polynomial time, an input instance is transformed into an equivalent instance—the *problem kernel*—whose size is bounded from above by $f(k)$ for some function f in some parameter k. The function f is called the *size* of the problem kernel.

Naturally, it is desirable that the size of the problem kernel is a slowly growing function of the parameter. Hence, the field of problem kernelization got involved into races of lowering the sizes of problem kernels and finding smaller parameters with respect to which problems allow for polynomial-size problem kernels [KN12]. Works focusing on the running time of problem kernelization, like those of Chor, Fellows, and Juedes [CFJ05] and Protti, Dantas da Silva, and Szwarcfiter [PDS09], have been an exception. The topic of linear-time kernelization has only recently regained attention starting with works of van Bevern et al. [Bev+12] and Hagerup [Hag12], although this aim is natural: the goal of data reduction is shrinking *large* input instances. This clearly limits the applicability of data reduction algorithms that run in quadratic or cubic time.

Linear-time problem kernels, in contrast, do not only lead to fixed-parameter linear-time algorithms. Beyond that, they can make kernelization algorithms applicable that do *not* run in linear time. For example, one could first compute a size-$O(k^2)$ problem kernel in linear-time, and from this equivalent, reduced instance compute a size-$O(k)$ problem kernel in cubic time. For this reason, races for the fastest and the smallest problem kernels can take place independently and yet perfectly complement each other. We will see an example for this in Chapter 5.

1.3 Overview

The thesis starts with an introduction to the theoretical foundations of discrete mathematics, graph theory, and computational complexity theory in Chapter 2.

We then proceed with the four main chapters, Chapter 3 to Chapter 6, which are based on two different strategies: Chapter 3 to Chapter 5 are devoted to the design and the experimental evaluation of fixed-parameter linear-time algorithms and linear-time data reduction for *concrete* graph and hypergraph problems. In contrast, Chapter 6 presents a *method* for deriving fixed-parameter linear-time algorithms for hypergraph problems. Although the method alone does not yield

practically applicable algorithms, the obtained algorithms may guide the way to future research.

Each of the four main chapters is devoted to one of the problems described in the beginning of this introductory chapter. We defer the definitions and overviews of the precise results to the respective chapters:

Chapter 3 shows fixed-parameter linear-time algorithms and problem kernels for generalizations and special cases of 2-Union Independent Set.

Chapter 4 presents fixed-parameter linear-time algorithms and linear-time data reduction for DAG Partitioning. While the presented data reduction does not yield problem kernels, experiments show that it significantly speeds up our algorithms.

Chapter 5 concentrates on linear-time computable problem kernels for d-Hitting Set, a problem that is already well-studied in terms of fixed-parameter algorithms and problem kernelization.

Chapter 6 presents a method that can be used to derive fixed-parameter linear-time algorithms for hypergraph problems. We illustrate it by showing that Hypergraph Cutwidth is fixed-parameter linear.

Finally, Chapter 7 discusses possible directions of future research.

2 Preliminaries

This chapter gives an introduction to the theoretical foundations of this thesis. We start by defining basic mathematical concepts in Section 2.1.

Then, since we will consider problems on graphs and hypergraphs, Section 2.2 gives an introduction to graph and hypergraph theory.

In Section 2.3, we recall the basic concepts of computational complexity theory, like decision problems, polynomial-time many-one reductions, and NP-hardness.

Section 2.4 gives a brief introduction to parameterized complexity theory, which focuses on analyzing the complexity of a problem not only with respect to the size of input instances, but also in additional parameters. Among other things, this section will formally define the concepts of fixed-parameter tractability and problem kernelization.

Finally, Section 2.5 provides tools to classify graph problems as fixed-parameter linear with respect to the graph parameter "treewidth."

2.1 Sets, relations, and functions

Throughout this thesis, we use the following identifiers to refer to frequently used sets of objects.

$\mathbb{N}$ is the set $\{0, 1, 2, \ldots\}$ of natural numbers.

$\mathbb{Z}$ is the set $\{\ldots, -1, 0, 1, \ldots\}$ of integers.

$\mathbb{R}$ is the set of real numbers.

$[c]$ for some $c \in \mathbb{N}$ is the subset $\{1, 2, \ldots, c\}$ of the natural numbers.

Σ is an arbitrary finite *alphabet* of *letters*, for example, $\Sigma = \{0, 1\}$.

Σ^* is the set of all finite words formed by the letters in the alphabet Σ. A subset of Σ^* is a *language*. For example, $\{0, 1\}^*$ is the set of all 0-1-strings of finite length.

2^A for some set A is the set of all subsets of A, also called *power set* of A.

$|\cdot|$ is the *cardinality* or *size* of a set or the *length* of a word.

$A \times A$ is the set of all pairs of elements of a set A.

A^ℓ is the set of all ℓ-tuples of elements of a set A.

A *partition* of a set A is a collection of non-empty, pairwise disjoint subsets of A whose union is A.

A *minimal* (*maximal*) set with a property P is a set, no proper subset (superset) of which has property P.

A *minimum* (*maximum*) set with a property P has the least (greatest) cardinality among all sets with property P.

The difference between *minimal* and *minimum* sets is crucial: clearly, a minimum set is minimal, but we will frequently encounter situations where minimal sets are easy to compute, while minimum sets are not. The same, of course, applies to maximal and maximum sets.

Relations. For some sets A, B, a *(binary) relation* between A and B is a subset $R \subseteq A \times B$. Instead of $(x, y) \in R$ for $x \in A$ and $y \in B$, we will write aRb and call b an *image* under R. A relation $R \subseteq A \times B$ is

total if, for all $x \in A$, there is a $y \in B$ with xRy,

surjective if, for all $y \in B$, there is an $x \in A$ with xRy,

injective if, for all $x, x' \in A$ and $y, y' \in B$ such that $x \neq x'$, xRy, and $x'Ry'$, we have $y \neq y'$, and

bijective if it is surjective and injective.

Example 2.1. The relation "=" is a total and injective relation between $\mathbb{N}$ and $\mathbb{Z}$, but it is not surjective and, hence, not bijective.

Equivalence relations. Instead of relations between two sets, we often consider relations over *one* set. An important special case of such relations are the equivalence relations, which will play a crucial role in Chapter 6. An *equivalence relation* $R \subseteq A \times A$ over a set A is a relation that is

reflexive — for all $x \in A$, we have xRx,

symmetric — for all $x, y \in A$, we have xRy if and only if yRx, and

transitive — for all $x, y, z \in A$, it holds that from xRy and yRz follows xRz.

Henceforth, we will denote equivalence relations by $\sim$ instead of R.

An *equivalence class* of an equivalence relation $\sim$ over a set A is a maximal set $X \subseteq A$ such that, for all $x, y \in X$, it holds that $x \sim y$.

Every element of A is in some equivalence class and two equivalence classes are either equal or disjoint.

The *index* of an equivalence relation is the number of its equivalence classes.

A *refinement* of an equivalence relation $\sim$ over A is an equivalence relation $\approx$ over A such that, for all $x, y \in A$, from $x \approx y$ follows $x \sim y$.

If $\approx$ is a refinement of $\sim$, then every equivalence class X of $\approx$ is fully contained in some equivalence class of $\sim$. Hence, the index of $\sim$ is at most the index of $\approx$.

Example 2.2. The relation "was born on the same weekday as" is an equivalence relation over the set of all people. Its index is 7. A refinement is, for example, the relation "was born on the same weekday and in the same hour as."

Functions. A relation $R \subseteq A \times B$ is a *function* or *map* from A to B, written $R : A \to B$, if for each $x \in A$, there is at most one $y \in B$ such that xRy. In this case, we write $R(x) = y$ or $R : x \mapsto y$.

In Chapter 3, we will exploit that, for two sets A and B, it holds that $|A| \geq |B|$ if and only if there is a surjective function $R : A \to B$.

To compare the asymptotic growth of functions, we use the O-notation. For two functions $f, g : \mathbb{N} \to \mathbb{N}$, we write

$g \in O(f)$ if $\exists c, n_0 \in \mathbb{N} : \forall n > n_0 : g(n) \leq cf(n)$, that is, if g grows as most as fast as f,

$g \in \Omega(f)$ if $f \in O(g)$, that is, if g grows at least as fast as f,

$g \in \Theta(f)$ if $g \in O(f)$ and $g \in \Omega(f)$, that is, if g and f grow equally fast, and

$g \in o(f)$ if $\forall c > 0 : \exists n_0 \in \mathbb{N} : \forall n > n_0 : g(n) \leq cf(n)$, that is, g grows strictly slower than f.

2.2 Graph theory

This section gives an introduction to the concepts of graph theory. We closely follow the graph theory books of Diestel [Die10] and West [Wes01].

Section 2.2.1 defines directed graphs, undirected graphs, and hypergraphs.

Section 2.2.2 defines graph relations like being a subgraph or being isomorphic, and operations like deleting vertices or edges.

Section 2.2.3 introduces graph structures like matchings and independent sets, and some graph-topological notions like connectivity and distances.

Finally, Section 2.2.4 introduces special graphs, like complete graphs, and graph classes like forests, bipartite graphs, and directed acyclic graphs.

2.2.1 Graphs and hypergraphs

In this thesis, we will meet different types of graphs. In Chapter 3, we will consider undirected graphs. In Chapter 4, we will consider directed graphs. In Chapter 5 as well as in Chapter 6, we will consider hypergraphs.

Undirected graphs. An *(undirected) graph* is a pair $G := (V, E)$ and

$V(G)$ is the set V of *vertices*,

$E(G)$ is the set $E \subseteq \{\{v, w\} \mid v, w \in V, v \neq w\}$ of *edges*,

n is the number $|V|$ of vertices,

m is the number $|E|$ of edges,

$N_G(v)$ is the *open neighborhood* of the vertex $v \in V$—the set of all vertices $u \in V$ that are *adjacent to* or *neighbors of* v, that is, for which $\{v, u\} \in E$,

$N_G[v]$ is the *closed neighborhood* $N_G(v) \cup \{v\}$ of the vertex $v \in V$,

$N_G[U]$ for a vertex set $U \subseteq V$ is $\bigcup_{v \in U} N_G[v]$, and

$\deg_G(v)$ is the *degree* $|N_G(v)|$ of the vertex $v \in V$.

If the graph G is understood from the context, then we drop the subscript G. Note that we do not allow edges to be present multiple times, that is, the edge set E is not a multiset.

Directed graphs. A *directed graph* is a pair $G := (V, A)$ and

$V(G)$ is the set V of *vertices*,

$A(G)$ is the set $A \subseteq \{(v,w) \mid v,w \in V, v \neq w\}$ of *arcs*,

n is the number $|V|$ of vertices,

m is the number $|A|$ of arcs,

$N_G^{\text{out}}(v)$ is the set $\{u \in V \mid (v,u) \in A\}$ of *out-neighbors* of a vertex $v \in V$,

$N_G^{\text{in}}(v)$ is the set $\{u \in V \mid (u,v) \in A\}$ of *in-neighbors* of a vertex $v \in V$,

$\deg_G^{\text{out}}(v)$ is the *outdegree* $|N_G^{\text{out}}(v)|$ of a vertex $v \in V$,

$\deg_G^{\text{in}}(v)$ is the *indegree* $|N_G^{\text{in}}(v)|$ of a vertex $v \in V$,

$\deg_G(v)$ is the *degree* $\deg_G^{\text{out}}(v) + \deg_G^{\text{in}}(v)$ of a vertex $v \in V$, and

the *underlying undirected graph* of G is a graph on the vertex set V and the edge set $E := \{\{u,v\} \mid (u,v) \in A\}$.

If the directed graph G is understood from the context, then we drop the subscript G. Again, the set A is not a multiset, that is, each arc is present at most once.

Note that an arc $(u,v) \neq (v,u)$ in a directed graph has a direction, while an edge $\{u,v\} = \{v,u\}$ of an undirected graph has not.

Hypergraphs. A *hypergraph* is a pair $H = (V, E)$ and

$V(H)$ is the set V of *vertices*,

$E(H)$ is the set $E \subseteq 2^V$ of *hyperedges*,

n is the number $|V|$ of vertices,

m is the number $|E|$ of hyperedges, and

the *incidence graph* of H is the undirected graph $\mathscr{I}(H) := (V', E')$, where $V' := V \uplus E$ and, for each vertex $v \in V$ and each hyperedge $e \in E$, there is an edge $\{v, e\} \in E'$ if and only if $v \in e$.

In Chapter 6, we will allow E to be a *multiset*, that is, to contain each hyperedge several times.

Graph and hypergraph representations. Since our main concern is solving problems in time linear in the input size (given that certain parameters are constant), it is crucial to agree on the graph and hypergraph representations we expect as input.

We assume all graphs to be represented using an *adjacency list*, that is, as a doubly-linked list of vertices, each being associated with a doubly-linked list of its out-neighbors and a doubly-linked list of its in-neighbors. For undirected graphs, these two lists will be the same. Adjacency lists allow for iterating over all vertices in $O(n)$ time, iterating over all out-neighbors of some vertex v in $O(\deg^{\text{out}}(v))$ time, and iterating over all in-neighbors of some vertex v in $O(\deg^{\text{in}}(v))$ time. Hence, iterating over all neighbors of all vertices works in $\sum_{v \in V} \deg(v) = O(n + m)$, that is, in linear time.

We assume hypergraphs to be given as a list of vertices and a *hyperedge list*, that is, as a doubly-linked list of hyperedges, each being a doubly-linked list of the vertices it contains. A hypergraph given as hyperedge list is linear-time transformable into an adjacency list of its incidence graph and vice versa.

2.2.2 Relations and operations on graphs

Two undirected graphs, directed graphs, or hypergraphs can be compared to each other using graph relations or combined with each other using graph operations.

Graph relations. Let $G = (V, E)$ be an undirected graph, directed graph, or hypergraph.

A *supergraph*	of G is a graph $H = (V', E')$ with $V' \supseteq V$ and $E' \supseteq E$.
A *subgraph*	of G is a graph $H = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$.
An *induced subgraph*	of G is a graph $H = (V', E')$ with $V' \subseteq V$ and $E' = \{e \in E \mid \forall v \in e : v \in V'\}$ in case of undirected graphs or hypergraphs, or $E' = \{(u, v) \in E \mid u, v \in V'\}$ in case of directed graphs.
An *isomorphism*	between G and an undirected graph or hypergraph $H = (V', E')$ is a bijective function $f : V \to V'$ such that e is an edge of G if and only if $\{f(v) \mid v \in e\}$ is an edge of H (with the same multiplicity). For an isomorphism f between G and a directed graph H, we require that (v, u) is an arc of G if and only if $(f(v), f(u))$ is an arc of H.
$G \cong H$	holds if there is an isomorphism between G and H, in other words, if G and H are *isomorphic*. Note that $\cong$ is an equivalence relation.

Graph operations. Let $G = (V, E)$ and $H = (V', E')$ be undirected graphs, directed graphs, or hypergraphs.

$G \cup H$	is the *union* of G and H, that is, $G \cup H := (V \cup V', E \cup E')$. If $V = V'$, then we say that $G \cup H$ is the *edge-wise union* of G and H.
$G \uplus H$	is the *disjoint union* of G and H, which is the same as the union but the elements of V and V' are relabeled so that $V \cap V' = \emptyset$.
$G[X]$	for some vertex set $X \subseteq V$ is the induced subgraph of G on the vertex set X.
$G - X$	for some vertex set X is the induced subgraph $G[V \setminus X]$, that is, G with the vertices in X and the incidence edges, hyperedges, or arcs removed.
$G \setminus X$	for some set X of edges, hyperedges, or arcs is the graph G with the edges, hyperedges, or arcs in X removed.

Subdividing an edge $\{u,v\}$ in an undirected graph means replacing $\{u,v\}$ by two edges $\{u,w\}$ and $\{w,v\}$ for a newly introduced vertex w.

2.2.3 Structures, topology, and parameters

We will often consider graphs of special structure or graphs in which certain parameters are small. These structural properties and graph parameters are presented in the following.

Graph structures.

An *isolated vertex* is a vertex v with $\deg(v) = 0$.

A *sink* is a vertex v with $\deg^{\text{out}}(v) = 0$.

A *leaf* is a vertex with $\deg(v) = 1$.

An *independent set* is a set of mutually nonadjacent vertices.

A *matching* is a set of mutually disjoint edges or hyperedges.

A *hitting set* is a set of vertices containing at least one element of each edge or hyperedge.

A *vertex cover* is a hitting set in a graph.

Graph topology.

A *path* in an undirected graph $G = (V,E)$ from a vertex v_1 to a vertex v_ℓ is an ℓ-tuple $(v_1, v_2, \ldots, v_\ell)$ of vertices with $\{v_i, v_{i+1}\} \in E$ for $i \in [\ell - 1]$. Its *length* is its number $\ell - 1$ of edges.

A *directed path* in a directed graph $G = (V,A)$ from a vertex v_1 to a vertex v_ℓ is an ℓ-tuple $(v_1, v_2, \ldots, v_\ell)$ of vertices such that $(v_i, v_{i+1}) \in A$ for $i \in [\ell - 1]$. Its *length* is $\ell - 1$.

An *undirected path* in a directed graph is a path in its underlying undirected graph.

A *cycle* in an undirected graph is a path from a vertex v to itself that uses each edge or arc at most once. Directed and undirected cycles in directed graphs are defined analogously.

The *distance* between two vertices v and w is the length of a shortest (undirected) path between v and w.

Connectivity. Two vertices v and w of an undirected graph are *connected* if there is a path between v and w.

Two vertices in a directed graph are *connected* if they are connected in the underlying undirected graph. In the literature, this is also called *weakly connected*.

Reachability. A vertex v *can reach* a vertex w or w *is reachable from* v in a directed graph if there is a directed path from v to w.

A *connected component* is a maximal set of pairwise connected vertices. In the literature, this is also called *weakly connected component* for directed graphs.

Graph parameters. Let G be a directed or undirected graph.

The *chromatic index* of G is the minimum number of colors required in a coloring of the edges such the edges of each color form a matching.

The *clique size* of a graph G is the maximum size of any clique in G.

The *diameter* of G is the maximum distance between any two vertices.

The *maximum degree* of G is the maximum degree of any vertex of G.

The *treewidth* of G is the minimum possible width of a tree decomposition for (the underlying undirected graph of) G (see Section 2.5).

The *pathwidth* of G is the minimum possible width of a path decomposition for (the underlying undirected graph of) G (see Section 2.5).

The *incidence treewidth* of a hypergraph H is the treewidth of its incidence graph $\mathscr{I}(H)$.

2.2.4 Graph classes and special graphs

This section introduces central graph classes as well as special graphs together with their shorthand identifiers.

As a *graph class*, we understand a set $\mathscr{C}$ of undirected graphs, directed graphs, or hypergraphs that is *closed under isomorphism*, that is, we have $H \in \mathscr{C}$ for all graphs $H \cong G$ with $G \in \mathscr{C}$. A graph class is

closed under disjoint unions if $G \uplus H \in \mathscr{C}$ for any $G, H \in \mathscr{C}$, and

closed under induced subgraphs if $G[X] \in \mathscr{C}$ for any graph $G = (V, E) \in \mathscr{C}$ and any $X \subseteq V$.

Graph classes.

A *bipartite graph* is an undirected or directed graph whose vertex set can be partitioned into two independent sets.

A *clique* is an undirected graph consisting pairwise adjacent vertices.

A *cluster graph* is an undirected graph in which every connected component is a clique.

A *d-uniform hypergraph* is a hypergraph in which every hyperedge has cardinality d. In particular, 2-uniform hypergraphs are graphs.

A *directed acyclic graph* is a directed graph that does not contain a directed cycle. Directed acyclic graphs are also known as *DAGs*.

An *$\mathscr{F}$-free* graph is a graph that contains none of the graphs in the set $\mathscr{F}$ as induced subgraph.

A *forest* is an undirected graph that does not contain a cycle.

A *tree* is a connected forest.

A *rooted tree* is a tree with one distinguished vertex called *root*. Non-root and non-leaf vertices are *inner vertices*. The neighbors of a vertex v of a rooted tree are called *children* if their distance to the root is larger than that of v and *parents* otherwise.

A *rooted n-ary tree* is a rooted tree in which every node has at most n children.

An *interval graph* is a graph, each vertex v of which can be represented as a closed interval $[v_s, v_e]$ of real numbers such that two vertices v and w are adjacent if and only if the intervals $[v_s, v_e]$ and $[w_s, w_e]$ intersect.

A *proper interval graph* is an interval graph that allows for an interval representation such no interval properly contains another. Equivalently, a proper interval graph is a $K_{1,3}$-free interval graph [BLS99, Theorem 7.1.9].

A *tournament* is a directed graph such that, for every pair of vertices u, v, there is either the arc (u, v) or the arc (v, u).

For more information on graph classes, we refer to the book of Brandstädt, Le, and Spinrad [BLS99].

Special graphs.

K_k is a clique on k vertices.

$K_{i,j}$ is the *complete undirected bipartite graph* whose vertex set can be partitioned into two independent sets V and W such that $|V| = i$ and $|W| = j$ and such that each vertex in V is adjacent to each vertex in W. The graph $K_{1,3}$ is also called *claw*.

P_k is a tree on k vertices and maximum degree two (a *path*).

2.3 Computational complexity

This section gives an introduction to computational complexity theory—the theory of measuring the difficulty of problems. For a more detailed introduction to computational complexity theory, we refer to the books of Arora and Barak [AB09], Garey and Johnson [GJ79], and Papadimitriou [Pap94].

Section 2.3.1 defines the notion of "time" and the model of computation that we will use—the unit-cost random access machine.

Section 2.3.2 introduces the notion of optimization problems and decision problems.

Section 2.3.3 introduces the most fundamental open problem of computational complexity theory: the P-NP-problem.

Finally, Section 2.3.4 discusses what it means for a problem to be NP-hard and how to prove that a problem is NP-hard.

2.3.1 What is time?

Since our main concern is solving problems in time linear in the input size (given that certain parameters are constant), it is crucial not only to agree on the graph and hypergraph representations we expect as input, but also on a notion of "time." To this end, we use the computational model of the random access machine [Pap94, Section 2.6].

Definition 2.1. A *random access machine (RAM)* has an unbounded array of *memory cells* $R_1, R_2, \ldots$, each of which holds an integer. For any $i, j \in \mathbb{N}$, the RAM uses one unit of time for each of

putting	some constant value into R_i,
copying	the content of R_i to R_j, or, using *indirect addressing*,
copying	the content of R_k to R_j, where k is the number in R_i,
copying	the content of R_i to R_k, where k is the number in R_j,
adding	the content of R_i to R_j,
subtracting	the content of R_i from R_j, or
comparing	the values of R_i and R_j and, based on the result, jumping to another line of the program.

In the case of indirect addressing, we also say that R_i and R_j, respectively, contain *pointers* to R_k. The random access machine gets its input from a finite input array and puts its output into R_0.

One usually assumes all memory cells of the RAM to be initially set to zero. However, on a real computer, initializing large portions of memory can take a significant amount of time [Ben86, Section 1.6]. In order to make sure that our algorithms run in linear time not only on RAMs but also on real computers, we will make *no assumptions* on the initial content of memory cells. That is, our algorithms will only read memory cells that we have explicitly filled with some value before.

Definition 2.2. We say that an algorithm runs

in *t(n)* time	if, on any input of length n, it takes at most $t(n)$ units of time on a random access machine.
in *linear time*	if it runs in $O(n)$ time, and
in *polynomial time*	if it runs in $p(n)$ time for some polynomial p. We will also write that it runs in $n^{O(1)}$ time.

2.3.2 Optimization problems versus decision problems

In the four example problems described in the beginning of Chapter 1, one is usually interested in solving *optimization problems* like the following:

MINIMUM d-HITTING SET
Input: A hypergraph $H = (V, E)$ with hyperedges whose cardinality is bounded from above by a constant d.
Output: A minimum-size *hitting set* S for H, that is, a set S containing at least one element of every hyperedge in E.

In optimization problems, we are searching for some object—in this case this object is a vertex set—that minimizes or maximizes some objective function. Here, the objective function is the size of the sought vertex set.

Computational complexity theory, however, usually analyzes the difficulty of *decision problems*—problems, the answer to which is either "yes" or "no." As an example, consider the decision problem corresponding to MINIMUM d-HITTING SET:

d-HITTING SET
Input: A hypergraph $H = (V, E)$ with hyperedges whose cardinality is bounded from above by a constant d, and a natural number k.
Question: Is there a hitting set S of size at most k for H?

Clearly, if MINIMUM d-HITTING SET is solvable in polynomial time, then d-HITTING SET is also solvable in polynomial time: we simply solve MINIMUM d-HITTING SET and then check whether the found hitting set has size at most k.

In this thesis, we focus on decision problems: if we show that d-HITTING SET is not solvable in polynomial time, then MINIMUM d-HITTING SET is not solvable in polynomial time either. Moreover, almost all algorithms that we present for decision problems will also solve the corresponding optimization problems, the only exception being Chapter 6.

Formally, a *decision problem* is, given some word $x \in \Sigma^*$ over some alphabet Σ, to decide whether $x \in L$ for some language $L \subseteq \Sigma^*$. If some $x \in \Sigma^*$ is in L, we say that x is a *yes-instance* of L. Otherwise, x is a *no-instance*.

Example 2.3. Since hypergraphs as well as integers can be represented as binary strings, in fact, d-HITTING SET is simply the decision problem whether some $x \in \{0,1\}^*$ is contained in the language

$$\begin{aligned} L_{d\text{HS}} := \{(H,k) \mid\ & H \text{ is a hypergraph with hyperedges whose cardinality} \\ & \quad \text{is bounded from above by a constant } d \\ & \text{and } k \text{ is an integer} \\ & \text{and } H \text{ has a hitting set of size at most } k\} \\ \subseteq \{0,1\}^*. & \end{aligned}$$

2.3.3 P versus NP

The question "P = NP?" is one of the major open questions in theoretical computer science. We now define the involved problem classes P and NP formally.

Definition 2.3.

P can be thought of as the class of efficiently solvable decision problems. Formally, P is the class of all languages $L \subseteq \Sigma^*$ for which there is an algorithm that decides $x \in L$ in polynomial time for any $x \in \Sigma^*$.

NP can be thought of as the class of decision problems whose solutions are efficiently verifiable. Formally, NP is the class of all languages $L \subseteq \Sigma^*$ for which there is a *verifier* algorithm such that an *input word* $x \in \Sigma^*$ is in L if and only if there is a *solution* or *certificate* $y \in \Sigma^*$ with length polynomial in x such that the algorithm answers "yes" given x and y.

In the literature, the classes P and NP are usually defined with respect to polynomial-time on the computational model of a Turing machine. However, the definition with respect to random access machines is equivalent [Pap94, Section 2.6].

Example 2.4. Clearly, there is a polynomial-time algorithm that, given a hypergraph H, a natural number k, and some set S, checks whether S is a hitting set of size k for H. Thus, this decision problem is in P.

We show that d-HITTING SET is in NP. As verifier algorithm for d-HITTING SET, we use the above polynomial-time algorithm. Clearly, (H,k) is a yes-instance

for d-HITTING SET if and only if the verifier algorithm answers "yes" for some certificate, that is, for some set S.

It holds that $\mathrm{P} \subseteq \mathrm{NP}$, since a verifier algorithm for any language $L \in \mathrm{P}$ can simply run the polynomial-time algorithm for L and ignore the certificate given to it. Hence, $L \in \mathrm{NP}$.

Any problem in NP can be solved in exponential time by a simple *exhaustive search* algorithm that simply tries all possible certificates of polynomial length for some input $x \in \Sigma^*$ and, for each certificate, runs the verifier algorithm in polynomial time. The P-NP-problem naturally arises from the question whether this exhaustive search can be avoided:

"P = NP?" asks whether all problems in NP are solvable in polynomial time, or, more philosophically, whether finding solutions is as easy as verifying them.

The widely accepted conjecture is that $\mathrm{P} \subsetneq \mathrm{NP}$. The reason for this wide acceptance is given in the following Section 2.3.4.

2.3.4 NP-hardness and completeness

Many problems within the complexity class NP can be translated into each other in polynomial time and, hence, a polynomial-time algorithm for one problem would yield polynomial-time algorithms for many other problems:

Definition 2.4. Let $K, L \subseteq \Sigma^*$ be two languages. A *polynomial-time many-one reduction* from K to L is a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that $x \in K$ if and only if $f(x) \in L$.

Now, if we have a polynomial-time algorithm for the language L, we obtain a polynomial-time algorithm for K as follows: given x, the algorithm for K just runs the polynomial-time algorithm for L on $f(x)$. It follows that, if there is no polynomial-time algorithm for K, then there is none for L. This leads to the following concepts.

Definition 2.5. A language $L \subseteq \Sigma^*$ is

NP-hard if, for all languages $K \in \mathrm{NP}$, there is a polynomial-time many-one reduction from K to L and

NP-complete if L is NP-hard and contained in NP.

Since the relation "is polynomial-time many-one reducible to" is transitive, in order to show that some language L is NP-hard, it is sufficient to show that there is a polynomial-time many-one reduction from an *arbitrary* NP-hard problem K to L.

Implications of NP-hardness. By Definition 2.5, either all or no NP-complete problem is solvable in polynomial time. Moreover, if any NP-hard problem is solvable in polynomial time, then all problems in NP, including the NP-complete ones, are solvable in polynomial time.

However, there are hundreds of NP-complete problems (for examples, we refer to the book of Garey and Johnson [GJ79]) for which no polynomial-time algorithms are known. Among these are our four problems described in Chapter 1.

2.4 Parameterized complexity

We have seen in Section 2.3.4 that there are good reasons to assume that NP-hard problems are not solvable in time polynomial in the *input size*.

This section gives a basic introduction to parameterized complexity, which tries to overcome this obstacle by measuring the difficulty of problems not only in the size of the input. For a more detailed introduction to parameterized complexity theory, we refer to the books of Downey and Fellows [DF13], Flum and Grohe [FG06], and Niedermeier [Nie06].

Section 2.4.1 introduces the notion of parameterized languages and the parameterized complexity analogs of the classes P and NP.

Section 2.4.3 formally defines the concept of a problem kernel, which is a formalization of efficient and effective data reduction.

2.4.1 Fixed-parameter tractability

With the aim of developing efficient algorithms for NP-hard problems, parameterized complexity theory measures complexity not only in the size of the input, but in an additional *parameter*. Instead of languages over Σ^*, it considers parameterized languages:

Definition 2.6. A *parameterized language* is a set $L \subseteq \Sigma^* \times \mathbb{N}$. The part x of a pair $(x,k) \in \Sigma^* \times \mathbb{N}$, is called *input*, whereas the part k is called *parameter*. A parameterized language $L \subseteq \Sigma^* \times \mathbb{N}$ is

fixed-parameter tractable if there is an algorithm deciding $(x,k) \in L$ in $f(k) \cdot |x|^c$ time for a computable function f and some constant c independent of k, and

fixed-parameter linear if there is an algorithm deciding $(x,k) \in L$ in $f(k) \cdot |x|$ time for some computable function f.

The corresponding algorithms are called *fixed-parameter algorithms* or *fixed-parameter linear-time algorithms*, respectively. Moreover,

FPT is the class of all fixed-parameter tractable languages.

In the literature, one can find various definitions of parameterized languages: Downey and Fellows [DF13] and Niedermeier [Nie06] define them as subsets of $\Sigma^* \times \Sigma^*$; Flum and Grohe [FG06] define them as subsets of Σ^* associated with a polynomial-time computable function that maps input instances to some parameter value $k \in \mathbb{N}$. Our definition follows that of Downey and Fellows [DF13] and Niedermeier [Nie06]. However, for our considerations, parameters drawn from $\mathbb{N}$ instead of Σ^* are sufficient.

Example 2.5. The language $L_{d\text{HS}} \subseteq \Sigma^*$ from Example 2.3 corresponding to d-HITTING SET can be considered as parameterized language in several ways:

The problem d-HITTING SET parameterized by k is simply the parameterized language $L^k_{d\text{HS}} := \{(H,k,k) \mid (H,k) \in L_{d\text{HS}}\}$. This parameterization will be subject of our studies in Chapter 5.

Likewise, one could consider d-HITTING SET parameterized by the incidence treewidth $\text{iw}(H)$ of the input hypergraph H, that is, the parameterized language $L^{\text{iw}}_{d\text{HS}} := \{(H,k,\text{iw}(H)) \mid (H,k) \in L_{d\text{HS}}\}$.

The problem d-HITTING SET parameterized by the number n of vertices in the input hypergraph is trivially fixed-parameter linear: we can simply try each of the 2^n vertex subsets for being a hitting set of size at most k.

The example illustrates that different parameterizations make a problem more or less complex in the sense of parameterized complexity theory: all decidable problems are easy parameterized by the size of the input.

We also consider problems parameterized by *combined parameters* of the form $(k_1, k_2, \ldots, k_j)$. Formally, this means that we parameterize the problem by $k := k_1 + k_2 + \cdots + k_j$. However, for a more fine-grained running time analysis, we will not state running times in terms of the single parameter k but in terms of the parameters $k_1, k_2, \ldots, k_j$ independently.

2.4.2 Parameterized hardness

The class FPT is the parameterized complexity analog to the class P from classical complexity theory: it can be thought of as containing the "efficiently solvable" problems. Unsurprisingly, there are classes of parameterized problems that are presumably not fixed-parameter tractable.

The first such class is, of course, the class of parameterized languages that are NP-hard even for constant parameter values. Such problems cannot be fixed-parameter tractable unless P = NP.

Moreover, there is a whole hierarchy of classes of presumably not fixed-parameter tractable problems—known as the W-hierarchy

$$\mathrm{FPT} \subseteq \mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \cdots \subseteq \mathrm{XP}.$$

Herein, XP is the class of parameterized languages that are solvable in polynomial time when the parameter is a constant. Note, however, that the degree of the polynomial may depend on the parameter.

Since this thesis is mainly concerned with developing fixed-parameter algorithms and not with proving parameterized hardness results, we will not formally define the W-classes. However, one can, analogously to classical complexity theory, define the concept of *parameterized reductions* and, for each $t \in \mathbb{N}$, the concept of *W[t]-hardness* and *W[t]-completeness* such that either all or no W[t]-complete problems are fixed-parameter tractable.

Since there is a variety of W[1]-complete and W[2]-complete problems for which no fixed-parameter algorithms are known, it is conjectured that W[1]-hard problems are not fixed-parameter tractable, that is, $\mathrm{FPT} \subsetneq \mathrm{W}[1]$.

2.4.3 Problem kernelization

Problem kernelization is a concept formally describing efficient and effective data reduction. This section introduces problem kernelization as well as the tools to prove that problem kernelization is presumably ineffective for certain parameterized problems.

Definition 2.7. Let $L \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized language. A *problem kernelization* is an algorithm that takes as input an instance $(x,k) \in \Sigma^*$ and, in time polynomial in $|x| + k$, outputs an instance (x',k') such that

$$|x'| \leq f(k) \qquad \text{for some computable function } f,$$

$k' \leq g(k)$ for some computable function g, and

$(x,k) \in L \Leftrightarrow (x',k') \in L$.

We call the last property the *correctness* of the data reduction and say that (x,k) and (x',k') are *equivalent* instances.

By *problem kernel* we refer to the instance (x',k').

The *size* of the problem kernel is the function f.

Note that, no matter how large the input instance to a kernelization algorithm is, the output will be an equivalent instance whose size *only* depends on the parameter.

In order to be able to kernelize large instances, we are interested in developing kernelization algorithms that not only run in polynomial time, but in linear time. Moreover, since we are interested in effective data reduction, we are interested in problem kernels of polynomial size.

Obviously, if a decidable problem has a problem kernel with respect to some parameter k, then it is fixed-parameter tractable with respect to the parameter k: just transform it into an equivalent instance of size $f(k)$ in polynomial time and decide that instance in arbitrary time. It can be shown that also the reverse direction holds.

2.4.4 Lower bounds on the problem kernel size

Bodlaender, Jansen, and Kratsch [BJK14] introduced a special form of polynomial-time many-one reduction, called "cross-composition." Using cross-compositions, one can show that some parameterized problems do not allow for polynomial-size problem kernels unless *the polynomial-time hierarchy* collapses. In computational complexity theory, a collapse of the polynomial-time hierarchy would be about as surprising as P = NP. We give an introduction to cross-compositions in the following.

Definition 2.8. A *polynomial equivalence relation* $\sim$ is an equivalence relation over Σ^* such that

i) there is an algorithm that decides $x \sim y$ in polynomial time for any two instances $x, y \in \Sigma^*$, and such that

ii) the index of $\sim$ over any *finite* set $S \subseteq \Sigma^*$ is polynomial in $\max_{x \in S} |x|$.

Example 2.6. Let $\sim$ be the relation under which all strings in Σ^* that are not graphs are equivalent and under which two graphs are equivalent if and only if they have the same number of vertices.

Then, $\sim$ is a polynomial-time equivalence relation: $G \sim H$ is checkable in polynomial time. Moreover a set $S \subseteq \Sigma^*$ that does not contain any graphs has only one equivalence class. Otherwise, if the largest graph in S has n vertices, then $\sim$ partitions S into at most $n+1$ equivalence classes (one equivalence class contains the words in Σ^* that are not graphs).

Definition 2.9. A language $K \subseteq \Sigma^*$ *cross-composes* into a parameterized language $L \subseteq \Sigma^* \times \mathbb{N}$ if there is a polynomial-time algorithm, called *cross-composition*, that, given a sequence $x_1, \ldots, x_s$ of s instances that are equivalent under some polynomial equivalence relation, outputs an instance (x^*, k) such that

i) k is bounded by a polynomial in $\max_{i=1}^{s} |x_i| + \log s$ and

ii) $(x^*, k) \in L$ if and only if there is an $i \in \{1, \ldots, s\}$ such that $x_i \in K$.

Observe that requiring the s input instances to be equivalent under some polynomial equivalence relation is no restriction: we can simply choose the polynomial equivalence relation under which all instances are equivalent. However, we can as well choose the equivalence relation from Example 2.6. Then, knowing that all input graphs have the same number of vertices can be exploited in the construction of cross-compositions.

Theorem 2.1 (Bodlaender, Jansen, and Kratsch [BJK14])**.** *If some NP-hard language $K \subseteq \Sigma^*$ cross-composes into the parameterized language $L \subseteq \Sigma^* \times \mathbb{N}$, then there is no polynomial-size problem kernel for L unless the polynomial-time hierarchy collapses.*

2.5 Fixed-parameter linear-time algorithms for graphs of bounded treewidth

This section gives an introduction to techniques for obtaining fixed-parameter linear-time algorithms for graph problems parameterized by the treewidth of the input graph. Intuitively, the treewidth is a measure for the tree-likeness of a graph. Considering this parameter is motivated by the fact that many NP-complete problems are linear-time solvable on trees.

We now formally define the treewidth of a graph. To this end, we define the relatively technical concept of a *tree decomposition*. Note, however, that we will not develop algorithms that directly work on tree decompositions and, therefore, refer to the book of Kloks [Klo94] for more details on tree decompositions.

Definition 2.10. A *tree decomposition* (T, β) for an undirected graph $G = (V, E)$ consists of a tree T and a function $\beta\colon V(T) \to 2^V$ of each *node* x of the tree T to a subset $V_x := \beta(x) \subseteq V$, called *bag*, such that

i) for each vertex $v \in V$, there is a node x of T with $v \in V_x$,

ii) for each edge $\{u, w\} \in E$, there is a node x of T with $\{u, w\} \subseteq V_x$,

iii) for each vertex $v \in V$, the nodes x of T with $v \in V_x$ induce a subtree.

A *path decomposition* of a graph is a tree decomposition (T, β) where T is a path instead of a tree.

The *width* of a tree decomposition is the maximum size of any bag minus one.

The *treewidth* of a graph G is the minimum possible width of any tree decomposition for G.

The *pathwidth* of a graph G is the minimum possible width of any path decomposition for G.

We use the same terminology for directed graphs, but understand all notions in terms of the underlying undirected graphs.

Trees have treewidth one. It is possible to decide in $f(t) \cdot n$ time whether a graph has treewidth at most t and, if it has, to construct a tree decomposition of width t in the same time [Bod96]. How tree decompositions help in algorithm design, we describe in the following.

Section 2.5.1 introduces the approach of formulating graph problems in monadic second-order logic of graphs.

Section 2.5.2 introduces the approach of solving graph problems on graphs of bounded treewidth using tree automata—an approach that we will generalize to hypergraphs in Chapter 6.

2.5.1 Monadic second-order logic

Monadic second-order logic is a powerful tool to quickly show that a graph problem is fixed-parameter linear parameterized by the treewidth of the input graph. The book of Courcelle and Engelfriet [CE12] gives a detailed introduction into the possibilities of monadic second-order logic. Here, we only give an introduction to the basic tools we will exploit.

Monadic second-order logic is a logic that can express graph properties or properties of structures in graphs. The variant of monadic second-order logic described by Courcelle and Engelfriet [CE12] and used by us allows the vertices of a graph to have colors and allows for testing whether a vertex has a certain color.

Definition 2.11. A formula φ in monadic second-order logic for (directed or undirected) graphs may consist of the logic operators $\vee, \wedge, \neg, \iff, \implies$ (or, and, negation, equivalence, and implication), vertex variables, edge variables, set variables, quantifiers $\exists$ and $\forall$ over vertices, edges, and sets, and the predicates

$x \in X$ for a vertex or edge variable x and a set X,

$\text{inc}(e,v)$, being true if e is an edge or arc incident to the vertex v,

$\text{adj}(v,w)$, being true if v and w are joined by an edge $\{v,w\}$ or by an arc (v,w),

$\text{col}_i(v)$, being true if v is a vertex and has color i, and

equality of vertex variables, edge variables, and set variables.

We use upper-case letters for set variables and lower-case letters for vertex and edge variables.

Example 2.7. The vertex set of a graph can be partitioned into three independent sets if and only if the graph satisfies the monadic second-order logic formula

$$\begin{aligned}\exists X,Y,Z \forall u,v[\text{adj}(u,v) \implies &(u \in X \vee u \in Y \vee u \in Z) \wedge (v \in X \vee v \in Y \vee v \in Z)\\ &\wedge \neg(v \in X \wedge u \in X)\\ &\wedge \neg(v \in Y \wedge u \in Y)\\ &\wedge \neg(v \in Z \wedge u \in Z)].\end{aligned}$$

That is, there are sets X, Y, and Z such that, for any pair of adjacent vertices u and v, we have that u and v are in one of the sets X, Y, and Z but not in the same.

The problem in Example 2.7 is also known as 3-COLORING. It is NP-hard and, by the following theorem, fixed-parameter linear parameterized by treewidth.

Theorem 2.2 (Courcelle's theorem [CE12, Theorem 6.4])**.** *Given a graph G and a constant-size formula φ in monadic second-order logic, checking whether φ holds in G is fixed-parameter linear parameterized by treewidth.*

Theorem 2.2 allows us to check graph properties in linear time on graphs of constant treewidth. There is a variant of Theorem 2.2 that allows us, for example, to search for structures like minimum vertex covers in graphs:

Example 2.8. A set S is a vertex cover for some graph if and only if it satisfies the monadic second-order logic predicate

$$\mathrm{vc}(S) \equiv \forall v \forall w [\mathrm{adj}(v,w) \implies v \in S \vee w \in S].$$

That is, S is a vertex cover if and only if, for each pair of adjacent vertices v and w, at least one of them is contained in S.

From Example 2.8 and the following theorem, it follows that a minimum vertex cover in a graph of constant treewidth is computable in linear time.

Theorem 2.3 ([CE12, Theorem 6.56])**.** *Given a graph G and a constant-size monadic second-order predicate $\varphi(S)$, finding a minimum set S satisfying $\varphi(S)$ is fixed-parameter linear parameterized by treewidth.*

Note that, although the algorithms resulting from Theorems 2.2 and 2.3 run in $f(t) \cdot n$ time on graphs of treewidth t, the function f often grows so fast that the resulting algorithms become practically inapplicable. Nevertheless, Kneis, Langer, and Rossmanith [KLR11] and Abseher et al. [Abs+14] have recently undertaken efforts towards implementing such algorithms.

2.5.2 Tree automata

Graph problems parameterized by treewidth can be reformulated as a "tree language" that can be decided in linear time by a "tree automaton." Indeed, this is the key to prove Courcelle's theorem, which is roughly proven as follows and as illustrated in Figure 2.1 [CE12, Theorem 6.4]:

Let L be some graph problem parameterized by the treewidth t of the input graph and let φ_L be a constant-size monadic second-order logic formula that a graph G satisfies if and only if $G \in L$. Then,

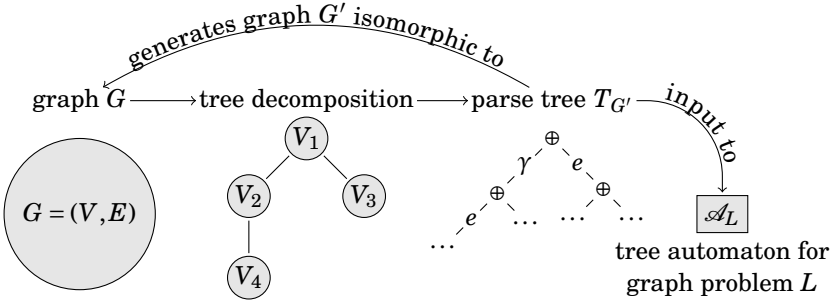


Figure 2.1: Solving a graph problem L using a tree automaton: from a graph G with constant treewidth, a minimum-width tree decomposition is computed in linear time. The tree decomposition is turned into a size-O(n) expression over a finite set of operators in linear time such that the value of the expression is a graph G' isomorphic to G. The parse or expression tree $T_{G'}$ of the expression is fed to a tree automaton $\mathscr{A}_L$ that accepts $T_{G'}$ in O(n) time if and only if $G' \in L$.

- given a graph G, it can be converted into an expression over a constant-size set of operators and, consequently, into a rooted binary expression tree T_G in $f(t) \cdot n$ time, and
- φ_L can be turned into a *tree automaton* $\mathscr{A}_L$ in constant time such that
- $\mathscr{A}_L$ processes T_G in $f(t) \cdot n$ time and accepts T_G if and only if $G \in L$.

We now formally define tree languages and tree automata. Then, we give an example for a tree automaton that accepts all true Boolean expressions.

Definition 2.12. Let Σ^{**} denote the set of all rooted trees whose vertices are labeled using letters from the alphabet Σ. A *tree language* is a set $L \subseteq \Sigma^{**}$. A *tree automaton* is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where

Q is a finite set of *states*,

Σ is a finite *alphabet*,

$q_0 \in Q$ is the *start state*,

$F \subseteq Q$ is the set of *final states*, and, finally

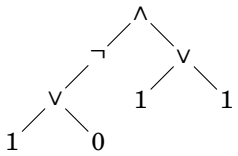


Figure 2.2: The parse tree of the Boolean expression $(\neg(1 \vee 0)) \wedge (1 \vee 1)$. The parse tree is in Σ^{**} for $\Sigma := \{\wedge, \vee, \neg, 0, 1\}$.

$\delta\colon (\Sigma \times Q) \cup (\Sigma \times Q \times Q) \to Q$ is the *transition function*.

A tree automaton processes a tree $T \in \Sigma^{**}$ starting at its leaves in order to determine the state at the root node of T as follows:

- The state at a leaf node x of T with label $a \in \Sigma$ is determined by $\delta(a, q_0)$.
- The state at a node x of T with label a and a single child node y is determined by $\delta(a, q_y)$, where $q_y \in Q$ is the state at y.
- The state at a node x of T with label a and two child nodes y and z is determined by $\delta(a, q_y, q_z)$, where $q_y, q_z \in Q$ are the states at y and z, respectively.

A tree automaton *accepts* a tree $T \in \Sigma^{**}$ if its state at the root node of T is in F. A tree automaton A *recognizes* a tree language $L \subseteq \Sigma^{**}$ if, for every tree $T \in \Sigma^{**}$, the automaton A accepts T if and only if $T \in L$.

Note that a tree automaton that operates on rooted *unary* trees (that is, on paths), can be understood as working on words in Σ^*. Thus, as a special case of a tree automaton, we obtain a *finite automaton*.

Example 2.9. Figure 2.2 shows the *expression tree* or *parse tree* of the Boolean expression $(\neg(1 \vee 0)) \wedge (1 \vee 1)$.

The following tree automaton $\mathscr{A} := (Q, \Sigma, \delta, q_{\text{init}}, \{q_{\text{acc}}\})$ accepts the parse trees of all true Boolean expressions over the operators $\wedge$, $\vee$, $\neg$, and the symbols 1 (true) and 0 (false), where

$$Q := \{q_{\text{acc}}, q_{\text{rej}}, q_{\text{init}}\}, \qquad \Sigma := \{\wedge, \vee, \neg, 0, 1\},$$
$$\delta(1, q_{\text{init}}) := q_{\text{acc}}, \qquad \delta(0, q_{\text{init}}) := q_{\text{rej}},$$
$$\delta(\neg, q_{\text{acc}}) := q_{\text{rej}}, \qquad \delta(\neg, q_{\text{rej}}) := q_{\text{acc}},$$
$$\delta(\wedge, q_{\text{acc}}, q_{\text{acc}}) := q_{\text{acc}}, \qquad \delta(\wedge, q_{\text{acc}}, q_{\text{rej}}) := q_{\text{rej}},$$
$$\delta(\wedge, q_{\text{rej}}, q_{\text{acc}}) := q_{\text{rej}}, \qquad \delta(\wedge, q_{\text{rej}}, q_{\text{rej}}) := q_{\text{rej}},$$
$$\delta(\vee, q_{\text{acc}}, q_{\text{acc}}) := q_{\text{acc}}, \qquad \delta(\vee, q_{\text{acc}}, q_{\text{rej}}) := q_{\text{acc}},$$
$$\delta(\vee, q_{\text{rej}}, q_{\text{acc}}) := q_{\text{acc}}, \text{ and} \qquad \delta(\vee, q_{\text{rej}}, q_{\text{rej}}) := q_{\text{rej}}.$$

That is, $\mathscr{A}$ accepts the tree language $L := \{T \in \Sigma^{**} \mid T \text{ is the parse tree of a true Boolean expression}\}$.

Which graph problems can be solved by tree automata? By Courcelle's theorem, the existence of a constant-size monadic second-order logic formula φ_L for a graph problem L is a sufficient condition for the existence of a tree automaton $\mathscr{A}_L$ that accepts the parse tree T_G for G if and only if $G \in L$.

Downey and Fellows [DF13, Section 12.7] give a sufficient and *necessary* condition for the existence of a tree automaton $\mathscr{A}_L$ for a graph problem L in form of the so-called "Myhill-Nerode theorem for graphs." We will give an introduction to the Myhill-Nerode technique in Chapter 6 in form of a generalization to hypergraphs.

3 Job Interval Selection and 2-Union Independent Set

Numerous problems in resource allocation, telecommunication, aircraft maintenance, and steel manufacturing can be modeled using the NP-hard INDEPENDENT SET problem [Bar+06; HKML11; KLPS07]: given an undirected graph and an integer k, find a set of at least k pairwise nonadjacent vertices.

Herein, one encounters variants like 2-UNION INDEPENDENT SET—the INDEPENDENT SET problem on edge-wise unions of two interval graphs—and its special case JOB INTERVAL SELECTION—the INDEPENDENT SET problem on edge-wise unions of an interval graph and a cluster graph. Since both variants are NP-hard, this chapter analyzes their parameterized complexity with a focus on providing fixed-parameter linear-time algorithms.

1. We show a complexity dichotomy, which shows that both problems are presumably not fixed-parameter tractable with respect to a variety of graph parameters and polynomial-time solvable only in very restricted cases.

2. We chart the possibilities and limits of problem kernelization for JOB INTERVAL SELECTION and 2-UNION INDEPENDENT SET.

3. We extend Halldórsson and Karlsson's [HK06] fixed-parameter linear-time algorithm for JOB INTERVAL SELECTION parameterized by the number of jobs (the number of connected components in the cluster graph) to the parameter k. Moreover, we generalize their algorithm to 2-UNION INDEPENDENT SET.

Our investigations benefit from a "compactness" interval graph parameter and from a unified view on 2-UNION INDEPENDENT SET and JOB INTERVAL SELECTION in form of a more general problem on vertex-colored interval graphs: COLORFUL INDEPENDENT SET WITH LISTS.

3.1 Introduction

Many scheduling problems can be modeled as finding maximum independent sets in generalizations of interval graphs [KLPS07]. Intuitively, finding a maximum independent set corresponds to scheduling a maximum number of pairwise non-conflicting jobs (represented by time intervals) on a limited set of machines.

In this context, we consider two popular generalizations of interval graphs, namely 2-union graphs [Bar+06] and strip graphs [HK06]: a graph $G = (V, E)$ is a *2-union graph* if it can be represented as the edge-wise union of two interval graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on the same vertex set V, that is, $G = G_1 \cup G_2 = (V, E_1 \cup E_2)$, where an *interval graph* is a graph whose vertices one-to-one correspond to intervals on the real line and there is an edge between two vertices if their intervals intersect. If one of the two interval graphs G_1 and G_2 even is a *cluster graph*, that is, a disjoint union of cliques, then G is a *strip graph*.

Examples of scheduling problems that can be modeled as (weighted) INDEPENDENT SET on 2-union graphs include resource allocation scenarios [Bar+06] and coil coating—a process in steel manufacturing [HKML11]. Formally, we are interested in solving the following problem:

2-UNION INDEPENDENT SET
Input: Two interval graphs G_1 and G_2 on the same vertex set and a natural number k.
Question: Is there an independent set of size at least k in $G_1 \cup G_2$?

A variant of the problem is where each vertex of the input graph has a weight and an independent set of total weight at least k is sought. Moreover, if one of the two input interval graphs is a cluster graph, then the problem is known as JOB INTERVAL SELECTION [Spi99].

We illustrate 2-UNION INDEPENDENT SET using a simplified version of the scheduling task of Höhn et al. [HKML11].

Example 3.1. Consider pieces of long metal sheets that have to be painted. Each painting job is to be executed by one of three machines. Moreover, there is a crew that sets up the machines for each specific job (by cleaning machines from remains of old paint and filling tanks with paint of the appropriate new color). Each machine can execute only one painting job at a time and the setup crew can only do setup work for one job at a time. We model this task using 2-UNION INDEPENDENT SET as illustrated in Figure 3.1.

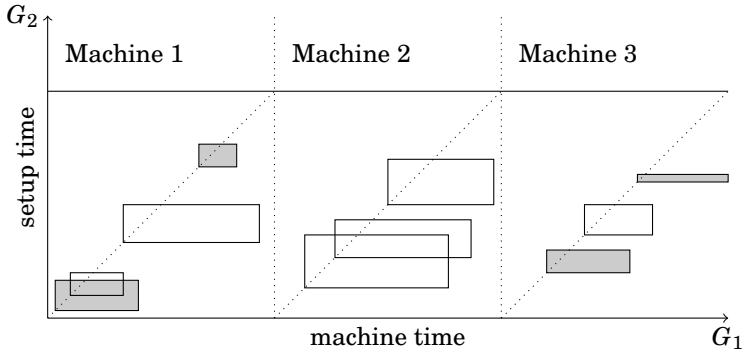


Figure 3.1: Jobs of painting sheet metal that can be performed by one of three machines and that require machine setup work by a setup crew. Each job is represented as a rectangle: its horizontal extent is an interval in an interval graph G_1 and represents the time it occupies on some machine; its vertical extent is an interval in an interval graph G_2 and represents the time it occupies the setup crew. Multiple machines are simply modeled as pairwise disjoint segments of the machine time axis. An independent set in $G_1 \cup G_2$ now corresponds to a set of rectangles whose projections onto neither time axis intersect, that is, to a set of jobs that neither occupy the setup crew nor the same machine at the same time. An example of an independent set is shown in form of the gray rectangles.

3.1.1 Known results

This section summarizes known results for 2-UNION INDEPENDENT SET as well as for JOB INTERVAL SELECTION.

Known results for 2-Union Independent Set. Already checking whether a graph is a 2-union graph is NP-hard [GW95; Jia13]. Therefore, we require two separate interval graphs as input to 2-UNION INDEPENDENT SET.

To date, a number of polynomial-time approximation algorithms for 2-UNION INDEPENDENT SET has been devised. Bar-Yehuda et al. [Bar+06] showed that 2-UNION INDEPENDENT SET with vertex weights admits a polynomial-time ratio-4 approximation. That is, in polynomial time, one can compute an independent set whose weight is at least 25% of the weight of a maximum-weight independent set. For the special case of $K_{1,5}$-free graphs (which comprises the case that both input graphs are proper interval graphs), Bafna, Narayanan, and Ravi [BNR96] provided a ratio-3.25 approximation.

In the context of steel manufacturing, Höhn et al. [HKML11] showed NP-hardness of 2-UNION INDEPENDENT SET on so-called M-composite 2-union graphs (which arise in their application), and showed a dynamic programming algorithm running in polynomial time for constant M, where the degree of the polynomial depends on M. They additionally provided heuristic algorithms for the problem, and, using these, optimized the production plans of a major German steel producer by lowering their production times by about 13%. Using lower bounds on the time required to produce the requested items, Höhn et al. [HKML11] showed that their heuristic solved the instances at hand to within 10% of the optimum.

Regarding parameterized complexity, Jiang [Jia10] proved that 2-UNION INDEPENDENT SET is W[1]-hard parameterized by the size k of the sought independent set, thus diminishing hopes for fixed-parameter tractability with respect to the parameter k. Jiang's [Jia10] W[1]-hardness result holds even when both input graphs are proper interval graphs.

Known results for Job Interval Selection. JOB INTERVAL SELECTION was introduced by Nakajima and Hakimi [NH82]. Spieksma [Spi99] showed that, unless P $\neq$ NP, there is no algorithm that, for each fixed $\varepsilon > 0$, computes in polynomial time an independent set with size within a factor of $(1-\varepsilon)$ of a maximum independent set (such algorithms are also known as *polynomial-time approximation schemes (PTAS)*. Spieksma [Spi99] also provided a ratio-2 approximation algorithm. Chuzhoy, Ostrovsky, and Rabani [COR06] improved this to a

ratio-1.582 approximation algorithm. Halldórsson and Karlsson [HK06] showed fixed-parameter tractability results for JOB INTERVAL SELECTION parameterized by the "maximum number of live jobs" or the "total number of jobs." Moreover, they showed that it is NP-hard to check whether a graph is a strip graph.

3.1.2 Our results

We provide a refined computational complexity analysis of 2-UNION INDEPENDENT SET with a focus on developing fixed-parameter linear-time algorithms. To this end, we make two main conceptual contributions:

1. We introduce COLORFUL INDEPENDENT SET WITH LISTS, a problem on vertex-colored interval graphs that is a natural generalization of 2-UNION INDEPENDENT SET and JOB INTERVAL SELECTION and allows for a unified view on both problems.

2. We bring the "compactness" interval graph parameter into parameterized complexity considerations: an interval graph is *c-compact* if its intervals are representable using at most c distinct start and end points. That is, c is the "number of numbers" required in an interval representation. Similar "number of numbers" parameters have previously been exploited to obtain fixed-parameter algorithms for problems unrelated to interval graphs [FGR12].

Moreover, we provide a complexity dichotomy that shows that all variants of 2-UNION INDEPENDENT SET considered in this chapter remain NP-hard.

By embedding JOB INTERVAL SELECTION and 2-UNION INDEPENDENT SET into COLORFUL INDEPENDENT SET WITH LISTS, we obtain the following results.

Results for Job Interval Selection. We complement known polynomial-time approximability results [COR06; Spi99] for JOB INTERVAL SELECTION with parameterized complexity results and extend the tractability results by Halldórsson and Karlsson [HK06] in several ways:

1. We generalize their fixed-parameter algorithm for JOB INTERVAL SELECTION parameterized by the "maximum number of live jobs" to COLORFUL INDEPENDENT SET WITH LISTS and, consequently, to a fixed-parameter linear-time algorithm for 2-UNION INDEPENDENT SET. Moreover, we show that their algorithm for JOB INTERVAL SELECTION can be turned into a

randomized fixed-parameter linear-time algorithm with respect to the parameter k—the number of intervals to be selected into an independent set.

2. We prove that, unless the polynomial-time hierarchy collapses, JOB INTERVAL SELECTION has no polynomial-size problem kernel with respect to k and other parameters, like the maximum clique size ω.

3. We show that, if the input graph is the edge-wise union of a cluster graph and a *proper* interval graph, then JOB INTERVAL SELECTION *does* have a polynomial-size kernel, more specifically, a problem kernel comprising $4k^2\omega$ intervals that can be computed in linear time.

Results for 2-Union Independent Set. Since 2-UNION INDEPENDENT SET is W[1]-hard with respect to the parameter k [Jia10] and NP-hard even when natural structural graph parameters like maximum clique size ω or maximum vertex degree are constants (which is implied by our complexity dichotomy), 2-UNION INDEPENDENT SET is unlikely to be fixed-parameter tractable for any of these parameters.

However, the *compactness* parameter arises naturally in our studies. We use $c_\forall$ to denote a number such that *both* input interval graphs are $c_\forall$-compact and $c_\exists$ to denote a number such that at least *one* of the input interval graphs is $c_\exists$-compact. We obtain the following results:

1. We give a simple data reduction rule for 2-UNION INDEPENDENT SET. The analysis of its effectiveness naturally leads to the compactness parameter: the reduction rule yields a $c_\forall^3$-vertex problem kernel for 2-UNION INDEPENDENT SET. This improves to a $2c_\forall^2$-vertex problem kernel if one of the input graphs is a proper interval graph.

2. The problem kernel with respect to $c_\forall$ shows that 2-UNION INDEPENDENT SET is fixed-parameter tractable with respect to $c_\forall$. By generalizing Halldórsson and Karlsson's [HK06] fixed-parameter algorithm from JOB INTERVAL SELECTION to 2-UNION INDEPENDENT SET, we improve this to a time-$O(2^{c_\exists} c_\exists \cdot n)$ fixed-parameter linear-time algorithm for the parameter $c_\exists \leq c_\forall$.

Our results are summarized in Table 3.1. We provide experimental results that show that 2-UNION INDEPENDENT SET instances with up to $5 \cdot 10^5$ intervals are solvable in less than five minutes if one of the two input interval graphs is 15-compact.

Table 3.1: Overview of parameterized complexity results for JOB INTERVAL SELECTION, where G_2—one of the two input graphs—is a cluster graph, and 2-UNION INDEPENDENT SET, where G_2 is any interval graph. Shown are results for various graph classes of G_1—the other input graph. The complexity dichotomy in Theorem 3.1 shows that all these problem variants remain NP-hard.

Class of G_1	JOB INTERVAL SELECTION	2-UNION INDEPENDENT SET
interval	randomized FPT algorithm: $O(5.5^k k \cdot n)$ time (Theorem 3.4)	FPT algorithm: $O(2^{c_\exists} c_\exists \cdot n)$ time (Theorem 3.7)
	No polynomial-size kernels w. r. t. k and ω (Theorem 3.5)	problem kernel: $c_\forall^3$ vertices in $O(n \log^2 n)$ time (Theorem 3.8)
proper interval	problem kernel: $4k^2\omega$ vertices in linear time (Theorem 3.6)	problem kernel: $2c_\forall^2$ vertices in $O(n \log^2 n)$ time (Theorem 3.8)

3.1.3 Chapter outline

Section 3.2 formally introduces COLORFUL INDEPENDENT SET WITH LISTS, a problem more general than JOB INTERVAL SELECTION and 2-UNION INDEPENDENT SET.

Section 3.3 presents a computational complexity dichotomy and its implications to the parameterized complexity of JOB INTERVAL SELECTION as well as of 2-UNION INDEPENDENT SET.

Section 3.4 formally introduces compact representations of interval graphs and shows that they are computable in linear time. The remaining sections will assume c-compact representations for minimum c as input.

Section 3.5 presents fixed-parameter algorithms for JOB INTERVAL SELECTION, lower bounds on problem kernel sizes, and polynomial-size problem kernels for special cases.

Section 3.6 contains fixed-parameter and kernelization algorithms for the more general problem 2-UNION INDEPENDENT SET.

Finally, we present experimental results in Section 3.7.

3.2 Colorful independent sets

Many of the results presented in this chapter benefit from a natural embedding of JOB INTERVAL SELECTION and 2-UNION INDEPENDENT SET into a more general problem: COLORFUL INDEPENDENT SET WITH LISTS. Herein, instead of considering the input as two interval graphs, we consider it as a single interval graph whose vertices may have one or multiple colors and search for an independent set such that the selected vertices have pairwise disjoint color sets.

3.2.1 A colorful version of Job Interval Selection

The foundation of COLORFUL INDEPENDENT SET WITH LISTS is an alternative formulation of the classical scheduling problem JOB INTERVAL SELECTION.

The task in JOB INTERVAL SELECTION is to execute a maximum number of jobs out of a given set, where each job has multiple possible execution intervals, each job is executed at most once, and a machine can only execute one job at a time. We formally state this problem in terms of colored interval graphs, where the colors correspond to jobs and intervals of one color correspond to multiple possible execution times of the same job:

JOB INTERVAL SELECTION
Input: An interval graph G with a coloring $\mathrm{col}\colon V(G) \to [\gamma]$ and a natural number k.
Question: Is there a colorful independent set of size at least k in G?

Herein, *colorful* means that no two vertices of the independent set have the same color.

As an example for a scheduling task that can be modeled using JOB INTERVAL SELECTION, consider the following fictional aircraft maintenance problem, which resembles the airport scheduling scenarios described by Kolen and Kroon [KK91; KK93; KK94].

Example 3.2. Consider an aircraft maintenance contractor that has orders for inspecting 15 airplanes the upcoming day. The contractor has three hangars, in each of which one airplane can be inspected at a time. For each hangar, there are three inspection crews, so that potentially each hangar can be used 24 hours a day in shift work.

However, there are constraints: not every crew has the license to inspect every type of aircraft, some hangars might be to small for large airplanes, some aircraft types take longer to inspect, experienced inspection crews work faster

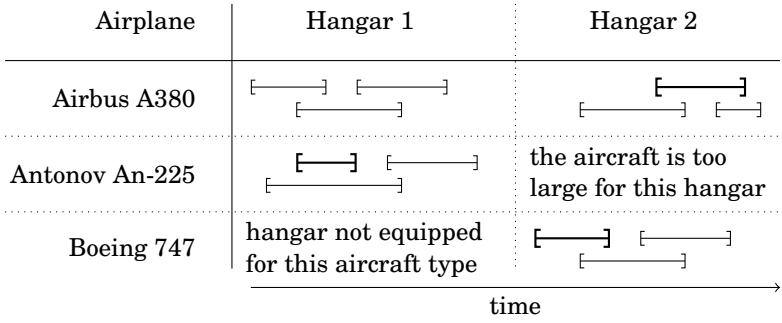


Figure 3.2: A model of the aircraft maintenance problem. Each airplane is interpreted as a color. Each interval has the color of the airplane to be inspected and represents a time segment at which a qualified inspection crew is available for inspecting the airplane in a given hangar. Herein, multiple hangars are simply modeled as pairwise disjoint segments of the time axis. It follows that two intervals may be chosen into one colorful independent set of the corresponding colored interval graph if and only if they do not inspect the same airplane and do not block the same hangar at the same time. A colorful independent set of size three is shown in the form of the thick intervals.

than others, and for unattractive working times like night shifts a bonus is paid to the employees.

This can be modeled as JOB INTERVAL SELECTION as illustrated in Figure 3.2. Then, finding a maximum colorful independent set in the resulting colored interval graph allows the aircraft maintenance contractor to serve as many orders as possible. Moreover, if we additionally assign to each interval the profit that the contractor will make by inspecting an airplane in that interval (recall that night shifts are more expensive), then finding a maximum-weight colorful independent set maximizes the profit for the contractor.

Observe that the number γ of airplanes to be inspected as well as the length of an inspection crew's work day can be small compared to the number of potential maintenance intervals: assume that there are 15 maintenance jobs, each of which can be accomplished by an inspection crew within at most eight hours. Moreover, we have three inspection crews for each of three hangars. For the purpose of illustration, let us assume the exaggerated scenario that each inspection crew can start inspection at any minute of the day, given that they are paid a high

enough bonus for unpopular working times. Since a day has 1440 minutes, we end up with $15 \cdot 3 \cdot 3 \cdot 1440 \approx 2 \cdot 10^5$ intervals, $\gamma = 15$ colors, and a maximum interval length of $8 \cdot 60 = 480$ minutes. Thus, parameterizing JOB INTERVAL SELECTION by the number γ of colors or the maximum interval length makes sense. Our experiments in Section 3.7 will show that a maximum-profit schedule even in this exaggerated scenario is computable within about 100 seconds.

Note that the colored formulation of JOB INTERVAL SELECTION provided above is indeed equivalent to the known formulation by Spieksma [Spi99], where JOB INTERVAL SELECTION is special case of 2-UNION INDEPENDENT SET with one input interval graph being a cluster graph:

JOB INTERVAL SELECTION∗
Input: An interval graph G_1, a cluster graph G_2 on the same vertex set, and a natural number k.
Question: Is there an independent set of size at least k in $G_1 \cup G_2$?

In this second formulation, the maximal cliques of G_2 correspond to jobs and the intervals in G_1 that belong to the same maximal clique of G_2 correspond to multiple possible execution times of the same job. That is, the maximal cliques in G_2 one-to-one correspond to the colors in the colorful problem formulation.

In Section 3.5.2, we will restate the fixed-parameter algorithms for JOB INTERVAL SELECTION by Halldórsson and Karlsson [HK06] in terms of our colorful formulation. This formulation allows for a more combinatorial point of view, rather than the geometric point of view taken by Halldórsson and Karlsson [HK06]. Exploiting this, we turn Halldórsson and Karlsson's [HK06] fixed-parameter linear-time algorithm for the total number of jobs (which translates to the number γ of colors in our formulation) into a randomized fixed-parameter linear-time algorithm for JOB INTERVAL SELECTION parameterized by the number $k \leq \gamma$ of jobs to be executed.

3.2.2 Colorful Independent Set with Lists

An advantage of formulating Halldórsson and Karlsson's [HK06] fixed-parameter algorithm for JOB INTERVAL SELECTION in our colorful model is that it easily generalizes to COLORFUL INDEPENDENT SET WITH LISTS—a canonical generalization of JOB INTERVAL SELECTION that we will observe to also capture 2-UNION INDEPENDENT SET.

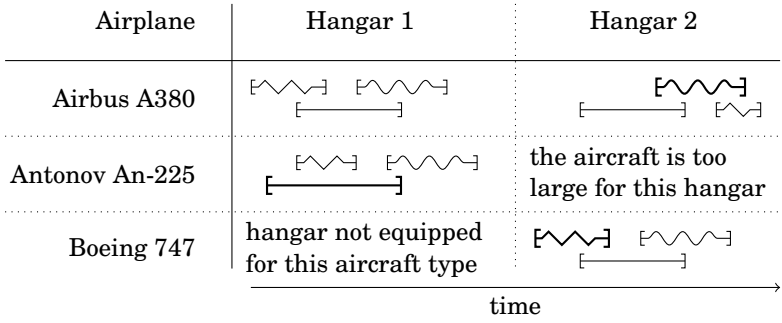


Figure 3.3: A modification of the aircraft maintenance problem from Figure 3.2. Now, airplanes as well as crews (each crew is represented by a distinct interval shape) are interpreted as colors and, thus, each potential inspection interval now has *two* colors: one for the airplane to be inspected and one for the crew to be inspecting it. It follows that two intervals may be chosen into one colorful independent set of the corresponding list-colored interval graph if and only if they do not block the same hangar at the same time, if they do not inspect the same airplane, and if they do not involve the same inspection crew. A maximum colorful independent set, here again shown as the thick intervals, allows the aircraft maintenance contractor to serve as many orders as possible.

COLORFUL INDEPENDENT SET WITH LISTS
Input: An interval graph G with a *list-coloring* $\mathrm{col}\colon V(G) \to 2^{[\gamma]}$ and a natural number k.
Question: Is there a colorful independent set of size at least k in G?

Herein, *colorful* means that the color sets of any two vertices in the independent set are disjoint.

We will see that COLORFUL INDEPENDENT SET WITH LISTS is an even more general problem than 2-UNION INDEPENDENT SET. For example, using the latter problem, it is difficult to model the following scheduling task, while it is easy using COLORFUL INDEPENDENT SET WITH LISTS.

Example 3.3. Consider again our aircraft maintenance problem from Example 3.2. This time, we do not want the maintenance crews to be tied to their hangars, so that if we have an expert crew for some type of aircraft, then that

crew can use any free hangar. Moreover, each crew should inspect only one plane per day, since inspecting two planes is considered infeasible within one shift.

We can model this problem using Colorful Independent Set with Lists as illustrated in Figure 3.3. Like in Example 3.2, the number of colors may be significantly smaller than the number of intervals to select from.

3.2.3 Advantages and limitations of the model

The colorful formulation of Job Interval Selection will help us to transform Halldórsson and Karlsson's [HK06] fixed-parameter linear-time algorithm for the parameter "number γ of colors" into a randomized fixed-parameter linear-time algorithm for the parameter "size $k \leq \gamma$ of the sought colorful independent set." Moreover, the algorithm for the colorful formulation of Job Interval Selection easily generalizes to Colorful Independent Set with Lists and, as we will see, to 2-Union Independent Set.

The advantage of Colorful Independent Set with Lists over 2-Union Independent Set is that one can concentrate on a single given interval graph instead of two merged ones, thus making the numerous structural results on interval graphs applicable. Possibly, the colorful view on finding independent sets and scheduling might be useful in further studies.

Not always, however, the colorful viewpoint gives an advantage. This is because Colorful Independent Set with Lists is a more general problem than 2-Union Independent Set and it is cumbersome to formulate *precisely* 2-Union Independent Set in terms of Colorful Independent Set with Lists. Thus, when exploiting the specific combinatorial properties of 2-Union Independent Set, as we will do it in the kernelization algorithm in Section 3.6, the colored model is not helpful. Moreover, in the following Section 3.3, we prove hardness results for 2-Union Independent Set and Job Interval Selection that apply to larger graph classes than only to list-colored interval graphs. Hence, the colorful model is not exploited in the upcoming hardness proof.

3.3 A complexity dichotomy

In this section, we determine the computational complexity of Independent Set on edge-wise unions of graphs in dependence of the allowed input graph classes. Formally, we define the considered problem as follows:

COMMON INDEPENDENT SET
Input: Two graphs G_1 and G_2 on the same vertex set and a natural number k.
Question: Is there an independent set of size at least k in $G_1 \cup G_2$?

Note that COMMON INDEPENDENT SET contains 2-UNION INDEPENDENT SET as special case. Indeed, the only difference is that it does not restrict the two input graphs to be interval graphs.

If we assume that the input graphs G_1 and G_2 are members in a graph class that is closed under induced subgraphs and disjoint unions, like interval graphs, cluster graphs, and forests, then we can use the main result of this section to precisely state for which classes COMMON INDEPENDENT SET is NP-hard and for which it is polynomial-time solvable, thus giving a *complexity dichotomy* for the problem.

Now, we first precisely state the dichotomy theorem. Since it is quite technical, we immediately illustrate it by some examples in form of implications to the complexity of JOB INTERVAL SELECTION and 2-UNION INDEPENDENT SET. We conclude the section with the proof of the theorem.

Theorem 3.1. *Let $\mathscr{C}_1$ and $\mathscr{C}_2$ be graph classes such that*

- *$\mathscr{C}_1$ and $\mathscr{C}_2$ are closed under disjoint unions and induced subgraphs, and*
- *$\mathscr{C}_1$ and $\mathscr{C}_2$ each contain at least one graph that has an edge.*

Then, COMMON INDEPENDENT SET *restricted to input graphs $G_1 \in \mathscr{C}_1$ and $G_2 \in \mathscr{C}_2$*

i) is solvable in $O(n^{1.5})$ time if $\mathscr{C}_1$ and $\mathscr{C}_2$ only contain cluster graphs, and

ii) NP-hard otherwise.

We illustrate Theorem 3.1 using two examples. First, choose $\mathscr{C}_1 = \mathscr{C}_2$ to be the class of interval graphs that have maximum degree two and maximum clique size two. Since P_3 is an interval graph but not a cluster graph in $\mathscr{C}_1$, we obtain the following NP-hardness result for 2-UNION INDEPENDENT SET:

Corollary 3.1. 2-UNION INDEPENDENT SET *is NP-hard even if both*

- *the maximum degree of each input interval graph is two, and*
- *the maximum clique size of each input interval graph is two.*

Now, choose $\mathscr{C}_1$ to be the class of disjoint unions of paths of length at most two and $\mathscr{C}_2$ to be the class of cluster graphs of clique size at most two. That is, the graphs in $\mathscr{C}_2$ only consist of pairwise disjoint edges and isolated vertices. Then, we have $P_3 \in \mathscr{C}_1$, which is not a cluster graph and, since disjoint unions of paths are interval graphs, we obtain the result that 2-Union Independent Set is NP-hard even if one input interval graph is a cluster graph consisting of cliques of size two and if the other is a disjoint union of paths of length at most two. Transferring this to the colored model as described in Section 3.2, we obtain:

Corollary 3.2. Job Interval Selection *is NP-hard even if the input interval graph is a disjoint union of paths of length at most two and contains each color at most two times.*

It remains to prove Theorem 3.1. For the first part of the theorem, Bar-Yehuda et al. [Bar+06] and Halldórsson and Karlsson [HK06] mentioned that Common Independent Set is polynomial-time solvable if both input graphs are cluster graphs. We prove here an explicit upper bound on the running time.

Lemma 3.1. Common Independent Set *is solvable in* $O(n^{1.5})$ *time if both input interval graphs are cluster graphs.*

Proof. Let G_1 and G_2 be cluster graphs on the same size-n vertex set and $G := G_1 \cup G_2$. Let C_i denote the set of connected components (cliques) of G_i for $i \in \{1,2\}$. Define a bipartite graph $H = (C_1 \uplus C_2, E_H)$ with edge set E_H such that there is an edge $\{U_1,U_2\} \in E_H$ if and only if there is a vertex $v \in V$ that occurs in clique U_1 of G_1 and in clique U_2 of G_2. We claim that G has an independent set of size k if and only if H has a matching of size k.

First, let I be an independent set of size k for G. We construct a matching M in H. To this end, for each $v \in I$, add an edge $\{U_1,U_2\}$ to M, where U_1 is the clique in G_1 that contains v and U_2 is the clique in G_2 that contains v. To verify that M is a matching in H, observe that each clique of G_1 or G_2 can contain only one vertex of I. Therefore, the edges in M are pairwise disjoint and $|M| = |I| = k$.

Second, let M be a matching of size k in H. We construct an independent set I for G with $|I| = k$ as follows: for each edge $\{U_1,U_2\} \in M$, include an arbitrary vertex contained in both U_1 and U_2 in I. Since each clique of G_1 or G_2 is incident to at most one edge of M, we have chosen at most one vertex per clique of G_1 and G_2, respectively. Hence, I is an independent set. Furthermore, this implies that we did not choose any vertex twice. Hence, $|I| = |M| = k$. This completes the proof of the claim.

It remains to prove the running time. We can verify in linear time that a graph is a cluster graph. Moreover its connected components can be listed in linear time using depth first search. Hence, in linear time, we can build a list L that, for each vertex, holds the index of its connected component in G_1 and the index of its connected component in G_2. This allows us to compute the bipartite graph H in linear time: as the vertices of H, we use indices of connected components. We now iterate over the list L and, for each vertex v, add to H an edge between the indices of the two connected components containing v. Adding an edge to H works in constant time since, for computing a maximum matching in H, we do not care whether we add edges to H multiple times. Now, by construction, the graph H has at most n edges, and, therefore, we can compute a maximum matching in H in $O(n^{1.5})$ time using the algorithm of Hopcroft and Karp [Sch03, Theorem 16.4]. □

We point out that Lemma 3.1 generalizes to finding a maximum-*weight* independent set if the vertices in the input cluster graphs have weights; to this end, we compute a *weighted* maximum matching in the auxiliary bipartite graph, where each edge is assigned the maximum weight of the vertices occurring in both clusters it connects.

To prove the second part of Theorem 3.1, we will employ a polynomial-time many-one reduction from 3-SAT.

3-SAT
Input: A Boolean formula φ in conjunctive normal form with at most three literals per clause.
Question: Does φ have a satisfying assignment?

In fact, we will see two very similar reductions from 3-SAT to COMMON INDEPENDENT SET, where the second is an extension of the first. This has the following benefits. The first reduction is an adaption of a simple NP-hardness reduction by Garey, Johnson, and Stockmeyer [GJS76] and, while not sufficient to show Theorem 3.1, it proves the following lemma, which we will exploit to exclude polynomial-size problem kernels for JOB INTERVAL SELECTION in Section 3.5.3:

Lemma 3.2. *In polynomial time, a* 3-SAT *instance φ can be reduced to a* COMMON INDEPENDENT SET *instance (G_1, G_2, k) such that*

i) G_1 consists of pairwise disjoint paths of length at most two and

ii) G_2 consists of k connected components, each being a triangle or an edge.

Moreover, k is proportional to the number of clauses in φ.

This first reduction, which we will use to prove Lemma 3.2, can then be modified to prove the following lemma, which we will exploit to show Theorem 3.1. Note that, in comparison to Lemma 3.2, the following lemma restricts G_2 to consist only of isolated edges and vertices. A polynomial-time many-one reduction that *additionally* ensures G_2 to have k connected components, like in Lemma 3.2, does not exist unless P = NP: in this case, COMMON INDEPENDENT SET is polynomial-time solvable using 2-SAT.

Lemma 3.3. *In polynomial time, a* 3-SAT *instance* φ *can be reduced to a* COMMON INDEPENDENT SET *instance* (G_1, G_2, k) *such that*

i) G_1 *consists of pairwise disjoint paths of length at most two,*

ii) G_2 *consists only of isolated edges and vertices, and*

iii) $G_1 \cup G_2$ *has chromatic index three.*

Moreover, k *is proportional to the number of clauses in* φ*.*

Proof of Lemmas 3.2 and 3.3. We first show Lemma 3.2. To this end, we transform a 3-SAT formula φ into three graphs G'_1, G'_2, and G'_3. The "three graphs approach" will easily allow us to show that the edge-wise union $G_1 := G'_1 \cup G'_3$ consists of pairwise disjoint paths of length at most two and that $G_2 := G'_2$ consists of k pairwise disjoint triangles and edges.

We now give the details of the construction. Let φ be a formula in conjunctive normal form with the clauses $C_1, \ldots, C_m$, each of which contains at most three variables from the variable set $\{x_1, \ldots, x_n\}$. For a variable x_i, let m_i denote the number of clauses in φ that contain x_i. For each clause C_j in φ, create the gadget shown in Figure 3.4, where an edge labeled $\ell \in \{1,2,3\}$ belongs to G'_ℓ. We call the non-triangle vertices *leaves* and the non-triangle edges *antennas*. With each variable x_i in C_j, we associate a leaf vertex $C_j(x_i)$.

For each variable x_i, create a cycle gadget X_i (as illustrated in Figure 3.5), with $2m_i$ edges, alternatingly labeled 2 and 3, and $2m_i$ vertices, which we alternatingly call *T-vertex* and *F-vertex*.

For each variable x_i of a clause C_j, we merge the leaf vertex $C_j(x_i)$ with a vertex of the variable gadget X_i for the variable x_i as follows: if x_i appears non-negated in C_j, then we identify $C_j(x_i)$ with an F-vertex of X_i, otherwise, we identify $C_j(x_i)$ with a T-vertex of X_i. Since X_i has m_i pairwise disjoint edges with label 3, each of which has one F- and one T-vertex, we can realize all these connections such that no edge with label 3 shares vertices with more than one antenna. The connections

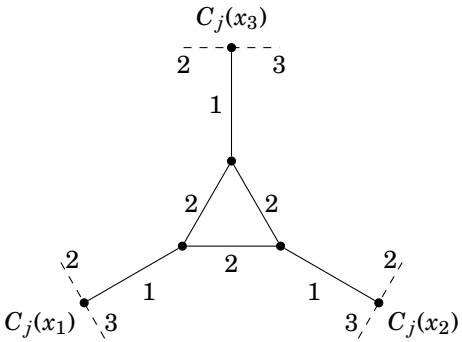


Figure 3.4: Solid edges form the gadget for a clause C_j containing the variables x_1, x_2, and x_3. With each variable x_i, we associate a leaf vertex $C_j(x_i)$, which will be merged with either a T- or an F-vertex in the gadget for variable x_i (Figure 3.5), depending on whether C_j contains the variable x_i negated or not. The dashed edges represent edges of the variable gadgets. Edges labeled by a number ℓ belong to the graph G'_ℓ.

are illustrated by dashed lines in Figures 3.4 and 3.5. We set $G_1 := G'_1 \cup G'_3$ and $G_2 := G'_2$ and output the COMMON INDEPENDENT SET instance (G_1, G_2, k), where $k := m + \sum_{i=1}^{n} m_i$.

It remains to show that the construction is correct and that G_1 and G_2 satisfy the required properties. Indeed, G_1 consists of pairwise disjoint paths of lengths at most two, since it only contains the edges labeled 1 or 3, of which each family forms a matching, and no edge with label 3 is ever connected to two edges labeled 1 and vice versa (only antennas are labeled 1). Moreover, G_2 consists of k isolated triangles and edges (labeled 2): it contains one isolated triangle for each of the m clauses and m_i isolated edges for each variable x_i. It remains to establish the correctness of the reduction by showing that φ is satisfiable if and only if $G_1 \cup G_2$ has an independent set of size k.

First, let I be an independent set of $G_1 \cup G_2$ with $|I| \geq m + \sum_{i=1}^{n} m_i = k$. Note that, for each variable x_i, the variable gadget X_i is a cycle of $2m_i$ vertices. Clearly, I contains at most half of these vertices. Moreover, I contains at most one triangle vertex of each of our m clause gadgets. Hence, $|I| \geq m + \sum_{i=1}^{n} m_i$ implies that

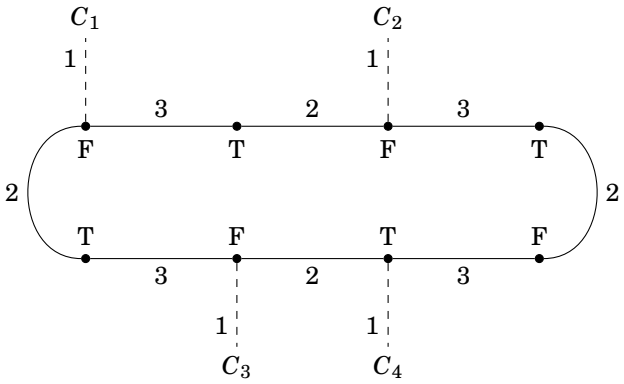


Figure 3.5: The solid edges form the gadget for a variable contained in the clauses C_1, C_2, and C_3 non-negated, but in C_4 negated. Dashed edges belong to the clause gadgets of the clauses $C_1, \ldots, C_4$. Edges labeled by a number ℓ belong to the graph G'_ℓ.

I contains a triangle vertex of each of the m clause gadgets and m_i vertices of each variable gadget X_i. Equivalently,

- for each variable gadget X_i, either all T-vertices or all F-vertices are contained in I, and
- for each clause gadget, one of its leaf vertices is *not* contained in I.

Equivalently, in each clause C_j, we find at least one of the following situations:

- C_j contains a positive literal x_i (then $C_j(x_i)$ is an F-vertex in X_i) and I contains all T-vertices of X_i, or
- C_j contains a negative literal $\bar{x}_i$ (then $C_j(x_i)$ is a T-vertex in X_i) and I contains all F-vertices of X_i.

Therefore, setting a variable x_i to true if and only if I contains the T-vertices of X_i yields a satisfying assignment for φ.

Now, assume that we have a satisfying assignment for φ. Then, putting into I all T-vertices of X_i if x_i is true and all F-vertices otherwise allows us to add a triangle vertex for each clause gadget to I and, thus, to construct an independent set I with $|I| \geq k$ for $G_1 \cup G_2$. Thus, we have proven Lemma 3.2.

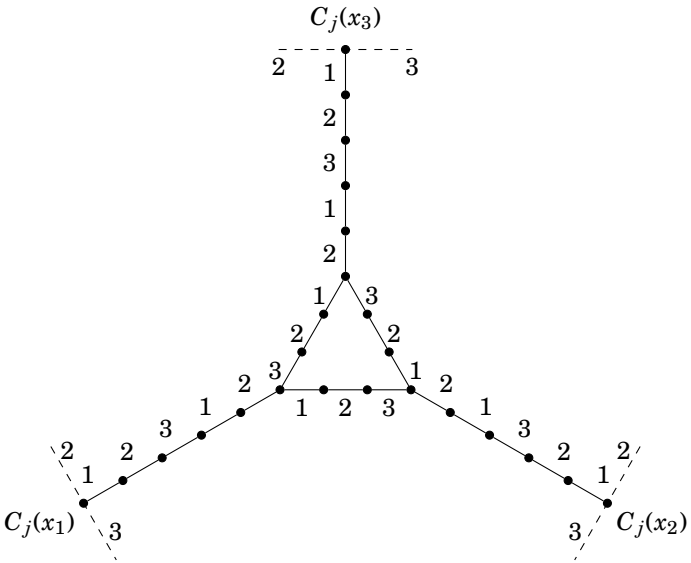


Figure 3.6: Advanced gadget for a clause C_j containing the variables x_1, x_2, and x_3. With each variable x_i, we associate a leaf vertex $C_j(x_i)$, which is merged with either a T- or an F-vertex in the gadget for variable x_i (Figure 3.5), depending on whether C_j contains the variable x_i negated or not. The dashed edges represent edges of the variable gadgets. Edges labeled by a number ℓ belong to the graph G'_ℓ.

We can now easily turn this reduction into a reduction that also proves Lemma 3.3. To this end, note that subdividing an edge of a graph twice increases the size of the graph's maximum independent set by exactly one. Thus, instead of using the clause gadget in Figure 3.4, we can use the gadget shown in Figure 3.6 and ask for an independent set of size $k := 10m + \sum_{i=1}^{n} m_i$: indeed, the gadget in Figure 3.6 is obtained from the simpler one in Figure 3.4 by subdividing each triangle edge twice and each antenna four times. Thus, we increase the maximum independent set size by $3 + 3 \cdot 2 = 9$ per clause gadget and, hence, ask for $k := 10m + \sum_{i=1}^{n} m_i$ instead of $m + \sum_{i=1}^{n} m_i$.

The benefit of replacing the gadget in Figure 3.4 by the gadget in Figure 3.6 is that G_2 now consists only of isolated edges and vertices instead of triangles.

Moreover, the resulting graph $G_1 \cup G_2 = G'_1 \cup G'_2 \cup G'_3$ has chromatic index three: the edges of each label form a matching. Thus, using Figure 3.6, we proved Lemma 3.3. □

Using Lemma 3.3, it is now easy to prove Theorem 3.1.

Proof of Theorem 3.1. Statement (i) immediately follows from Lemma 3.1. It remains to show (ii). To this end, observe that, without loss of generality, $\mathscr{C}_1$ contains not only cluster graphs. Since a graph is a cluster graph if and only if it is P_3-free, $\mathscr{C}_1$ contains a graph that has a P_3 as induced subgraph. Since $\mathscr{C}_1$ is closed under induced subgraphs and disjoint unions, it follows that $\mathscr{C}_1$ contains all graphs that consist of pairwise disjoint paths of length at most two. With the same argument and exploiting that $\mathscr{C}_2$ contains at least one graph with an edge, we obtain that $\mathscr{C}_2$ contains all graphs consisting of isolated vertices and edges. Hence, NP-hardness follows from Lemma 3.3. □

Concluding this section, we derive one further hardness result for Job Interval Selection from Lemma 3.2. It exploits the Exponential Time Hypothesis stated by Impagliazzo, Paturi, and Zane [IPZ01], which implies that n-variable 3-Sat cannot be solved in $2^{o(n)} \cdot n^{O(1)}$ time. The hypothesis is justified by the fact that a subexponential-time algorithm for 3-Sat would lead to subexponential-time algorithms for a large class of other NP-complete problems (also see the survey of Lokshtanov, Marx, and Saurabh [LMS11]). Essentially, the following corollary shows that Job Interval Selection cannot be solved in a time that grows subexponentially in k or γ, even if we sacrifice our aim for running times that grow only linearly in the input size.

Corollary 3.3. *Even when restricted to instances*

- *with an input graph that consists of disjoint paths of length at most two, and*
- *that ask for a colorful independent set of size k with k being equal to the number γ of input colors,*

Job Interval Selection *remains*

i) NP-hard, and

ii) cannot be solved in $2^{o(k)} \cdot n^{O(1)}$ time unless the Exponential Time Hypothesis fails.

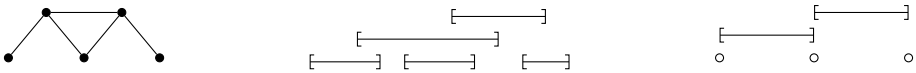


Figure 3.7: From left to right: an interval graph, an interval representation thereof, and an equivalent 3-compact interval representation. The shape ⊢⊣ represents a closed interval with distinct start and end points, while the shape ∘ represents a closed interval containing only one point.

Proof. Statement (i) is the simple translation of Lemma 3.2—that COMMON INDEPENDENT SET remains NP-hard even if one input graph is a cluster graph with k connected components—into our colored model.

It remains to prove (ii). By Lemma 3.2, a 3-SAT instance with n variables and m clauses is reduced to an instance of COMMON INDEPENDENT SET with $k \in O(m)$. Thus, a $2^{o(k)} \cdot n^{O(1)}$-time algorithm for COMMON INDEPENDENT SET would yield a $2^{o(m)} \cdot n^{O(1)}$-time algorithm for 3-SAT. This, in turn, by the so-called *Sparsification Lemma* of Impagliazzo, Paturi, and Zane [IPZ01, Corollary 2], would imply a $2^{o(n)} \cdot n^{O(1)}$ time algorithm for 3-SAT, which contradicts the Exponential Time Hypothesis. □

3.4 Compact interval graphs

This section introduces an interval graph parameter that we will see to highly influence the computational complexity of 2-UNION INDEPENDENT SET: the *compactness* of an interval graph, which is the minimum number of distinct numbers required in its interval representation. Similar "number of numbers" parameters have in the past been successfully exploited to obtain fixed-parameter algorithms for problems unrelated to interval graphs [FGR12].

A related interval graph parameter is the *interval count*, which is the minimum number of distinct interval lengths needed to represent an interval graph. While polynomial-time algorithms for computing the interval count are known only for subclasses of interval graphs and it is open whether the problem is NP-hard [COS11], we will see that the compactness of an interval graph is computable in linear time.

We now formally define compactness and give some simple observations on this graph parameter that we will exploit throughout this chapter. The following definition is illustrated in Figure 3.7.

Definition 3.1. An interval representation is *c-compact* if the start and end point of each interval lies in $[c]$ and the intervals are sorted by their start points.

An interval graph is *c-compact* if it admits an interval representation that is c-compact.

In this chapter, we state all running times under the assumption that interval graphs are given in a *c-compact interval representation* for some c. An n-compact representation of an interval graph given as adjacency list is computable in $O(n+m)$ time [COS09, Section 8].

Observation 3.1. Any n-vertex interval graph is n-compact.

Of course, since we aim to use the compactness of an interval graph as a parameter for fixed-parameter algorithms, we are interested in the *minimum* c for which an interval graph is c-compact. Its exact value is given by the number of maximal cliques in the interval graph:

Observation 3.2. Let G be an interval graph and c be the minimum integer such that G is c-compact. Then, G has exactly c maximal cliques.

For the proof of this observation, recall from Section 2.2.4 that each vertex v of an interval graph is represented by an interval $[v_s, v_e]$ with start point v_s and end point v_e and that there is an edge between two vertices v and w if and only if the intervals $[v_s, v_e]$ and $[w_s, w_e]$ intersect.

Proof of Observation 3.2. Let c' be the number of maximal cliques in G. We show $c = c'$ by proving $c' \leq c$ and $c \leq c'$ independently.

First, it is easy to see that a c-compact interval graph has at most c maximal cliques: each start point v_s or end point v_e gives rise to at most one maximal clique, which consists of the intervals containing the point v_s or v_e, respectively. Hence, $c' \leq c$.

Second, the interval graph G allows for an ordering of its c' maximal cliques such that the cliques containing an arbitrary vertex occur consecutively in the ordering [FG65]. Hence, each vertex v can be represented by the interval $[v_s, v_e]$, where v_s is the number of the first maximal clique containing v and v_e is the number of the last maximal clique containing v. It follows that $c \leq c'$. □

Finally, we give an $O(n+m)$-time algorithm that computes a c-compact representation with minimum c in $O(n+m)$ time from an interval graph given as adjacency list.

Observation 3.3. Given an interval graph G as adjacency list, in $O(n+m)$ time, one can compute a c-compact representation for G such that

i) at each position in $[c]$, there is an interval start point *and* an interval end point, and

ii) c is the minimum number such that G is c-compact.

Proof. In the following proof, we will not consider start or end points (henceforth both called *event points*) to be mere integer positions; instead, an event point e is an object that has an integer position $p(e)$ and is associated with the interval it starts or ends. Thus, when we "move" event points, we actually modify the corresponding intervals accordingly. We now first describe the algorithm and then show (i) and (ii).

First, using an $O(n+m)$-time algorithm by Corneil, Olariu, and Stewart [COS09, Section 8], we obtain an n-compact interval representation of G. In linear time, we iterate over this list of intervals and, for each interval $[v_s, v_e]$, add the start point v_s to the beginning of a list L and the end point v_e to the end of the list L. As a result, L contains all interval start points before all interval end points.

We now sort the event points in L by increasing positions. Since there are at most n event points, all of which have positions not exceeding n, they can be sorted in $O(n)$ time using counting sort [CLRS01, Section 8.2]. Since counting sort is a stable sorting algorithm, for each position $i \in [n]$, the sorted list L contains first all start points and then all end points at position i. We will exploit this fact and maintain it as an invariant.

Now, we iterate once over the list L, that is, over all event points by increasing positions, and move event points from higher positions to lower positions while maintaining all pairwise interval intersections. Equivalently, for any start point v_s and any end point u_e, we will have $p(v_s) \leq p(u_e)$ after movement if and only if it holds before movement. For each event point e in L, we consider the event point e' that directly precedes e in L and proceed according to one of the following four cases, all of which are illustrated in Figure 3.8.

Case 1) Assume that e' is an end point and that e is a start point. If $p(e')+1 < p(e)$, then our invariant ensures that there are no end points at the positions $\{p(e')+1, \ldots, p(e)-1\}$. Therefore, we can move e to position $p(e')+1$. Thereafter, e is the only event point at position $p(e')+1$; hence, our invariant is maintained.

(a) Case 1. (b) Case 2. (c) Case 3. (d) Case 4.

Figure 3.8: The four cases for the event point e and its direct predecessor e' in the proof of Observation 3.3.

Case 2) Assume that both e' and e are end points. If $p(e') < p(e)$, then our invariant ensures that there are no start points at the positions $\{p(e')+1,\dots,p(e)\}$. Thus, we can move e to $p(e')$. Thereafter, e is an end point at the same position as the end point e' and occurs after e' in L; hence, our invariant is maintained.

Case 3) Assume that e' is a start point and that e is an end point. Observe that it might be the case that they are the start and end points of the same interval. If $p(e') < p(e)$, then our invariant ensures that there are no start points at the positions $\{p(e')+1,\dots,p(e)\}$. Thus, we can move e to $p(e')$. Thereafter, the end point e is the last vertex in L at the position of e'; hence, our invariant is maintained.

Case 4) Finally, assume that both e' and e are start points. If $p(e') < p(e)$, then our invariant ensures that there are no end points at the positions $\{p(e'),\dots,p(e)-1\}$. Thus, we can move e to $p(e')$. Since there are no end points at $p(e')$, our invariant is maintained.

It is now easy to prove (i) and (ii).

(i) Each interval start point v_s is also an end point for some interval: otherwise, we would have moved an event point e (possibly, $e = v_e$) whose position directly follows that of v_s to the position of v_s. Second, each interval end point v_e is also a start point for some interval: otherwise, we would have moved v_e to the position of an event point whose position directly precedes that of v_e.

(ii) From (i), it follows that, for any two positions i, j of event points, the set of intervals containing i and the set of intervals containing j are distinct and, therefore, i and j give rise to distinct maximal cliques in G. Thus, our algorithm computes a c'-compact representation of G with $c' \leq c$, where c is the number of maximal cliques in G. From Observation 3.2, it follows that c' is the minimum number such that G is c'-compact. □

The algorithm given in this proof also constructs c-compact representations for minimum c when interval graphs are given in form of interval representations rather than in form of adjacency lists. However, then the algorithm takes $O(n \log n)$ time to initially sort the event points.

3.5 Job Interval Selection

In this section, we investigate the parameterized complexity of Job Interval Selection. As a warm-up for working with the colored model and in order to illustrate one of the most basic techniques for obtaining fixed-parameter algorithms, Section 3.5.1 first gives a simple search tree algorithm that solves Job Interval Selection in linear time if the size k of the sought colorful independent set and the maximum number Γ of colors in any maximal clique of G are constant.

Section 3.5.2 then proceeds with a reformulation of the fixed-parameter algorithm of Halldórsson and Karlsson [HK06] with respect to the parameter "maximum number of live jobs" into our colored model, which makes it easy for us to generalize the algorithm to Colorful Independent Set with Lists and also to show that Job Interval Selection is fixed-parameter tractable parameterized by k (as opposed to parameterizing it by both k and Γ as in the search tree algorithm we are going to see first).

We conclude our findings for Job Interval Selection in Section 3.5.3 by showing that the problem has no polynomial-size problem kernel in general, but on proper interval graphs.

3.5.1 A simple search tree algorithm

As a warm-up for working with the colored formulation of Job Interval Selection, and also to illustrate one of the most basic techniques for obtaining fixed-parameter algorithms, this section presents a simple search tree algorithm leading to the following theorem:

Theorem 3.2. Job Interval Selection *is solvable in* $O(\Gamma^k \cdot n)$ *time, where* Γ *is the maximum number of colors occurring in any maximal clique.*

Only for $\Gamma < 6$ the worst-case running time of Theorem 3.2 can compete with our generalizations of the dynamic program of Halldórsson and Karlsson [HK06] that we will see in Section 3.5.2. However, as opposed to the dynamic programs, the space requirements of the search tree algorithm are polynomial.

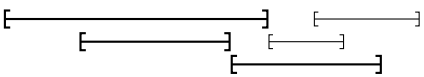


Figure 3.9: The thick intervals start no later than any interval ends.

The first ingredient in our search tree algorithm is the following lemma, which shows that there is a maximum colorful independent set containing one of the "first" intervals of the interval graph. This is illustrated in Figure 3.9.

Lemma 3.4. *Let K be the set of intervals that start no later than any interval in G ends. Then, there is a maximum colorful independent set that contains exactly one interval of K.*

Proof. Let I be a maximum colorful independent set for G with $I \cap K = \emptyset$ and let v^* be the interval in G that ends first. Obviously, $v^* \in K$ and any interval $v \in I$ intersecting v^* is in K. Hence, since $I \cap K = \emptyset$, I contains no interval intersecting v^*. It follows that I contains an interval w such that $\mathrm{col}(w) = \mathrm{col}(v^*)$, otherwise $I \cup \{v^*\}$ would be a larger colorful independent set. Now, $I' = (I \setminus \{w\}) \cup \{v^*\}$ is a colorful independent set for G with $|I'| = |I|$ and $v^* \in I' \cap K$.

Finally, note that I cannot contain more than one interval of K, since the intervals in K pairwise intersect. □

The second ingredient in our search tree algorithm is the following lemma, which shows that knowing the color of the interval in K that is to be included in a maximum colorful independent set is sufficient to make an optimal choice among the intervals in K.

Lemma 3.5. *Let K be the set of intervals that start no later than any interval in G ends. Moreover, assume that there is a maximum colorful independent set containing an interval of color c from K.*

Then, there is a maximum colorful independent set that contains the interval of color c from K that ends first.

Proof. Let I be a maximum colorful independent set, let $v \in K \cap I$ and let $\mathrm{col}(v) = c$. Moreover, let v^* be the interval in K with $\mathrm{col}(v^*) = c$ that ends first. By Lemma 3.4, I contains at most one interval of K. Then, since v intersects all intervals that intersect v^*, we know that $I' = (I \setminus \{v\}) \cup \{v^*\}$ is a colorful independent set with $|I'| = |I|$. □

Using Lemma 3.4 and Lemma 3.5, it is easy to solve Job Interval Selection in $O(\Gamma^k \cdot n)$ time and, thus, to prove Theorem 3.2:

Proof of Theorem 3.2. The algorithm computes a colorful independent set of size at least k recursively. First, it chooses K to be the set of intervals that start no later than any interval in G ends and $C := \bigcup_{v \in K} \mathrm{col}(v)$ to be the set of colors occurring in K. Note that, given a c-compact representation for minimum c of the input interval graph, these computations can be executed in $O(n)$ time by sorting the event points by increasing positions in $O(n)$ time like we did it in the proof of Observation 3.3. Since the intervals in K form a clique, it follows that $|C| \leq \Gamma$.

By Lemma 3.4 and Lemma 3.5, it is now sufficient, for each color $c \in C$ and the first-ending interval v with $\mathrm{col}(v) = c$, to try choosing v for inclusion into a colorful independent set and to recursively apply the algorithm to search for a colorful independent set of size at least $k-1$ in the interval graph G' obtained from G by removing all intervals of color c and all intervals intersecting v. If any of these recursive calls returns a colorful independent set of size at least $k-1$ for G', we know that adding v to it yields a colorful independent set of size at least k for G.

Recursion stops when k reaches 0, that is, when the algorithm searches for a colorful independent set of size at least $k = 0$. Thus, the recursion depth is bounded by k. Each recursive call causes at most Γ new recursive calls. It follows that there are $O(\Gamma^k)$ recursive calls in total. For each recursive call, we compute the sets K and C and delete intervals that intersect a previously chosen interval or have the same color as that. All of this can be done in $O(n)$ time, yielding the total $O(\Gamma^k \cdot n)$ running time of the algorithm. □

3.5.2 Generalizations of an algorithm of Halldórsson and Karlsson

In this section, we first present a dynamic program for Job Interval Selection by Halldórsson and Karlsson [HK06] in terms of our colored model. Based on this presentation, we show modifications in order to lower its space requirements, then we generalize it from Job Interval Selection to Colorful Independent Set with Lists and, finally, transform it into a randomized fixed-parameter linear-time algorithm with respect to the parameter k.

It is easy to see that the dynamic programs in this section can be straightforwardly generalized to the problem variant where each interval has assigned a weight and we search for a colorful independent set of maximum weight, rather than a colorful independent set of maximum size.

A dynamic program for the parameter "number γ of colors." Let (G,k) be an instance of Job Interval Selection, where G is given in c-compact representation for minimum c. For $i \in [c+1]$ and $C \subseteq [\gamma]$, we use $T[i,C]$ to denote the size of a maximum colorful independent set in G that uses only intervals whose start point is at least i and whose color is in C. Obviously, for $i = c+1$ and any $C \subseteq [\gamma]$, we have $T[i,C] = 0$. Knowing $T[i,C]$ for some $i \in [c+1]$ and all $C \subseteq [\gamma]$, we can easily compute $T[i-1,C]$ for all $C \subseteq [\gamma]$, since there are only two cases:

Case 1) If there is a maximum colorful independent set of intervals with start point at least $i-1$ and colors belonging to C that contains an interval v with start point $v_s = i-1$, then $T[i-1,C] = 1 + T[v_e+1, C \setminus \{\mathrm{col}(v)\}]$, where v_e is the end point of v.

Case 2) Otherwise, $T[i-1,C] = T[i,C]$.

It follows that we can compute the size $T[1,[\gamma]]$ of a maximum colorful independent set in G using the recurrence

$$T[i-1,C] = \max \begin{cases} T[i,C], \\ 1 + \max\limits_{\substack{v \in V, v_s = i-1 \\ \mathrm{col}(v) \in C}} T[v_e+1, C \setminus \{\mathrm{col}(v)\}]. \end{cases} \tag{DP-γ}$$

This is an alternative formulation of the dynamic program of Halldórsson and Karlsson [HK06] using colored interval graphs. Since we have a c-compact representation of G, we can evaluate recurrence (DP-γ) in fixed-parameter linear time by iterating over the n intervals in G in order of decreasing start points and by, for each interval, iterating over the 2^γ subsets of $[\gamma]$. In this way, we first encounter all intervals with start point c, then those with start point $c-1$ and so on, so that we compute the table entries for decreasing start points $i \in [c+1]$. Herein, the c-compact representation not only ensures that the intervals are sorted by their start points, but also that, for each $i \in [c]$, some interval starts in i and, therefore, that the table entry $T[i,C]$ indeed gets filled for all $C \subseteq [\gamma]$. This algorithm yields an alternative proof of a result due to Halldórsson and Karlsson [HK06]:

Lemma 3.6 (Halldórsson and Karlsson [HK06])**.** Job Interval Selection *is solvable in* $O(2^\gamma \gamma \cdot n)$ *time.*

Halldórsson and Karlsson [HK06] actually state a running time of $O(2^\gamma \cdot n)$ for Lemma 3.6, since they assume table look-ups for sets of arbitrary size to

work in constant time. We decide to include the extra factor of γ to account for these table look-ups: if each subset $C \subseteq [\gamma]$ is represented as a binary length-γ string $j_\gamma j_{\gamma-1} \dots j_2 j_1$ with $j_\ell = 1$ for $\ell \in C$ and $j_\ell = 0$ otherwise, the value associated with C can be looked up in $O(\gamma)$ time in a *trie* [AHU83, Section 5.3].

Improving to the parameter "maximum number Q of live colors." Halldórsson and Karlsson [HK06] improved recurrence (DP-γ) by bounding its exponential running time part not in the number γ of colors but in the number $Q \leq \gamma$ of "live colors," which formally is defined as follows:

Definition 3.2. Let G be an interval graph given in c-compact representation and with vertex colors in $[\gamma]$. For each $i \in [c+1]$, let

$L_i \subseteq [\gamma]$ be the set of colors that appear on intervals with start point at most i (note that $L_{c+1} = [\gamma]$), and

$R_i \subseteq [\gamma]$ be the set of colors that appear on intervals with start point at least i (note that $R_{c+1} = \emptyset$).

Then $Q := \max_{i \in [c+1]} |L_i \cap R_i|$ is the maximum number of *live colors*. That is, a color c is *live* at a point i if there is an interval with color c that starts no later than i as well as an interval with color c that starts no earlier than i.

Example 3.4. To get an intuition for this parameter, reconsider our aircraft maintenance problem in Figure 3.10. Since each maintenance job corresponds to one color, we have $\gamma = 3$. However, the color for the Boeing 747 is only "live" in Hangar 2, while the color for the Antonov An-225 is "live" only in Hangar 1. Thus, the maximum number Q of live colors in this example is two.

We now describe the fixed-parameter algorithm for Job Interval Selection parameterized by Q. First, observe that, given an interval graph G in c-compact representation, the sets R_i and L_i for all $i \in [c+1]$ are computable in $O(\gamma n)$ total time in two passes: first, we iterate over all intervals by increasing start points and compute each L_i from L_{i-1}. Then, we iterate over all intervals by decreasing start points and compute each R_i from R_{i+1}.

Using Definition 3.2, we first observe that, when searching for a maximum colorful independent set containing only intervals with start point at least $i \in [c+1]$, then it is safe to allow this independent set to contain all colors of $\bar{L}_i := [\gamma] \setminus L_i$: this is because an interval with start point less than i cannot have a color in $\bar{L}_i$. Hence, we are only interested in the values $T[i, C]$ for $i \in [c+1]$ and $\bar{L}_i \subseteq C \subseteq [\gamma]$.

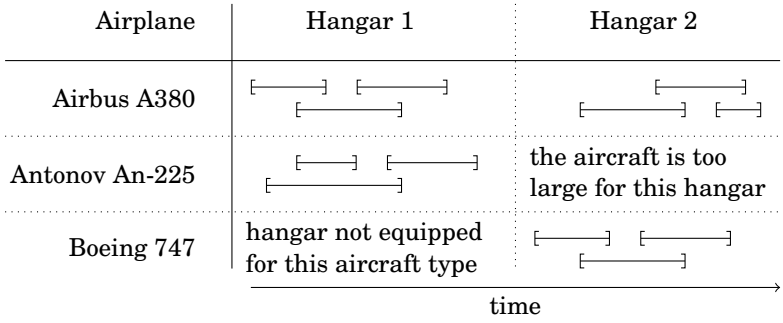


Figure 3.10: A model of the aircraft maintenance problem. Each airplane is interpreted as a color. Each interval has the color of the airplane to be inspected and represents a time segment at which a qualified inspection crew is available for inspecting the airplane in a given hangar. Herein, multiple hangars are simply modeled as pairwise disjoint segments of the time axis.

Second, a colorful independent set that only contains intervals with start point at least $i \in [c+1]$ only contains intervals with colors in R_i. Therefore, it is safe to allow only colors contained in R_i. We see that we are only interested in the values $T[i, C]$ for $i \in [c+1]$ and $\bar{L}_i \subseteq C \subseteq R_i$. There are at most 2^Q such subsets, since, for each C with $\bar{L}_i \subseteq C \subseteq R_i$, we have $C \setminus \bar{L}_i \subseteq L_i \cap R_i$.

Exploiting these observations in (DP-γ), we can compute $T[i-1, C]$ for all C with $\bar{L}_{i-1} \subseteq C \subseteq R_{i-1}$ as

$$T[i-1,C] = \max \begin{cases} T[i, (C \cup \bar{L}_i) \cap R_i], \\ 1 + \max\limits_{\substack{v \in V, v_s = i-1 \\ \mathrm{col}(v) \in C}} T[v_e + 1, (C \cup \bar{L}_{v_e+1}) \cap (R_{v_e+1} \setminus \{\mathrm{col}(v)\})]. \end{cases} \qquad \text{(DP-}Q\text{)}$$

As we have not changed the semantics of a table entry compared to (DP-γ), the size of a maximum colorful independent set in G is, as before, $T[1, [\gamma]]$. Hence, the improved dynamic program of Halldórsson and Karlsson [HK06] also works in our colored model:

Lemma 3.7 (Halldórsson and Karlsson [HK06])**.** Job Interval Selection *is solvable in* $O(2^Q \gamma \cdot n)$ *time, where* Q *is the maximum number of live colors as defined in Definition 3.2.*

Halldórsson and Karlsson [HK06] actually state a running time of $O(2^Q \cdot n)$ for Lemma 3.7, accounting constant time for table look-ups and the involved set operations. Again, we include the extra factor γ to account for the running time of look-ups, set unions, intersections, and subtractions involving sets of size γ. Thus, in fact, (DP-Q) only yields a fixed-parameter linear-time algorithm for the parameter Q when measuring the running time like Halldórsson and Karlsson [HK06].

Improving the space complexity. Having stated the dynamic programs of Halldórsson and Karlsson [HK06] in terms of our colored model, we now build upon these algorithms. The dynamic programming table of recurrence (DP-Q) has $2^Q \cdot (c+1)$ entries when a c-compact interval graph is given. We improve it to $2^Q \cdot (\ell+2)$, where ℓ is the length of the longest interval in the input interval graph. That is, if Q and ℓ are constant, or at least small like in our aircraft maintenance problem from Example 3.2 (also see Figure 3.10), then we can solve arbitrarily large input instances using a dynamic programming table of constant size.

The space complexity improvement is based on a simple observation: when computing $T[i-1, C]$ in (DP-Q), there is a largest possible $i' > i-1$ and some color set $C' \subseteq [\gamma]$ for which we access $T[i', C']$. By definition of T, $i' = v_e + 1$ for some interval v with end point v_e and start point $v_s = i-1$. We have $i' - 1 = v_e \leq v_s + \ell = i - 1 + \ell$, and, hence, $i' \leq i + \ell$. It follows that we only need $2^Q \cdot (\ell+2)$ table entries, since the entry $T[i-1, C]$ does not need the value $T[i+\ell+1, C]$ and can therefore reuse the space previously occupied by $T[i+\ell+1, C]$. This we simply achieve by storing $T[i, C]$ for $i \in [c+1]$ and $C \subseteq [\gamma]$ in a table $T'[i \bmod (\ell+2), C]$ that has only $\ell+2$ entries in the first coordinate. Having shrunken the dynamic programming table in this way, we obtain the following lemma:

Lemma 3.8. Job Interval Selection *is solvable in* $O(2^Q \gamma \cdot n)$ *time and* $O(2^Q \ell + \gamma c)$ *space when the input graph is given in* c*-compact representation,* ℓ *is the maximum interval length, and* Q *is the maximum number of live colors as defined in Definition 3.2.*

Herein, $O(2^Q \ell)$ space is used by the dynamic programming table and $O(\gamma c)$ space is used to hold the sets L_i and R_i from Definition 3.2, which we used to speed up the dynamic programming.

Generalization to Colorful Independent Set with Lists. We now generalize (DP-Q) to Colorful Independent Set with Lists. That is, vertices are now allowed to have multiple colors instead of just one and we search for a

maximum independent set that is colorful in the sense that no pair of vertices may have common colors. The algorithm for Colorful Independent Set with Lists will allow us to solve 2-Union Independent Set in Section 3.6.1.

Exploiting the formulation of (DP-Q) in our colored model, the generalization to Colorful Independent Set with Lists turns out to be easy. Again, we assume that the input graph G is given in c-compact representation for minimum c. For $i \in [c+1]$ and $C \subseteq [\gamma]$, we use $T[i,C]$ to denote the size of a maximum colorful independent set in G that uses only intervals with start point at least i and whose color list is a *subset* of C.

Completely analogously to (DP-Q) for Job Interval Selection, we can compute the size $T[1,[\gamma]]$ of a maximum colorful independent set by computing $T[i-1,C]$ for each $i \in [c+1]$ and all color sets C with $\bar{L}_{i-1} \subseteq C \subseteq R_{i-1}$ as

$$T[i-1,C] = \max \begin{cases} T[i,(C \cup \bar{L}_i) \cap R_i], \\ 1 + \max\limits_{\substack{v \in V, v_s = i-1 \\ \mathrm{col}(v) \subseteq C}} T[v_e+1, (C \cup \bar{L}_{v_e+1}) \cap (R_{v_e+1} \setminus \mathrm{col}(v))]. \end{cases} \qquad \text{(DP-}Q^*\text{)}$$

Note that the space complexity improvement demonstrated for (DP-Q) also works for (DP-Q*). Hence, we can merge Lemmas 3.6–3.8 into the following theorem:

Theorem 3.3. *Given an interval graph with γ colors and maximum interval length ℓ in c-compact interval representation,* Colorful Independent Set with Lists *is solvable in $O(2^Q \gamma \cdot n)$ time and $O(2^Q \ell + \gamma c)$ space, where Q is the maximum number of live colors as defined in Definition 3.2.*

The parameter "size k of the sought colorful independent set." In the following, we improve recurrence (DP-γ) for Job Interval Selection parameterized by the number γ of colors to a fixed-parameter algorithm for Job Interval Selection parameterized by the size $k \leq \gamma$ of the sought colorful independent set. Our first step is providing a randomized fixed-parameter linear-time algorithm for Job Interval Selection. The algorithm correctly answers if a no-instance of Job Interval Selection is given. In contrast, it rejects yes-instances with a given error probability ε. The randomized algorithm can be derandomized to show the following theorem:

Theorem 3.4. Job Interval Selection *can be solved with error probability ε in $O(5.5^k k \cdot |\ln \varepsilon| \cdot n)$ time. The algorithm can be derandomized to deterministically solve* Job Interval Selection *in $O(12.8^k k\gamma \cdot n)$ time.*

Note that, in practical applications, the randomized algorithm is probably preferable over the derandomized one, since the error probability can be chosen low without increasing the running time significantly. However, the derandomized algorithm shows that Job Interval Selection is fixed-parameter tractable parameterized by k.

To prove Theorem 3.4, we use the color-coding technique by Alon, Yuster, and Zwick [AYZ95] to reduce the number γ of colors in the given instance to k. Thereafter, recurrence (DP-γ) can be evaluated in $O(2^k k \cdot n)$ time. Depending on whether we reduce the number of colors randomly or deterministically, this method will yield the first or the second running time stated in Theorem 3.4.

Proof of Theorem 3.4. Let (G, col, k) be an instance of Job Interval Selection. In a first step, we assign each color in $[\gamma]$ a color in $[k]$ uniformly at random. Let $\delta\colon [\gamma] \to [k]$ denote this recoloring and let (G, col', k) denote the resulting instance with $\mathrm{col}'(v) = \delta(\mathrm{col}(v))$ for all vertices v. Note that, in general, δ is not injective. Then, we use (DP-γ) to compute a size-k colorful independent set in the resulting instance. Since the resulting instance has only k colors, this works in $O(2^k k \cdot n)$ time.

We now first analyze the probability that a colorful independent set for (G, col, k) is also a colorful independent set for (G, col', k) and vice versa. Then, we analyze how often we have to repeat the procedure of recoloring and computing recurrence (DP-γ) in order to achieve the low error probability ε.

First, assume that the recolored instance (G, col', k) is a yes-instance. Then, there is a colorful independent set I with $|I| \geq k$. The set I is a colorful independent set also for the original instance (G, col, k), since each color in col is mapped to only one color in col'. It follows that (G, col, k) is a yes-instance.

Now, assume that the original instance (G, col, k) is a yes-instance. We analyze the probability of the recolored instance (G, col', k) being a yes-instance. Let I be a colorful independent set for (G, col, k). The set I is a colorful independent set for (G, col', k) if the vertices in I have pairwise distinct colors with respect to col'. Since the vertices in I have pairwise distinct colors with respect to col and we assign each color in $[\gamma]$ a color in $[k]$ uniformly at random, the colors of the vertices of I with respect to col' are also chosen uniformly at random and independently from each other. Thus, the probability of I being colorful with respect to col' is $p := k!/k^k$: out of k^k possible ways of coloring the k vertices in I with k colors, there are $k!$ ways of doing so in a colorful manner. Hence, the probability of (G, col', k) also being a yes-instance, is $p := k!/k^k$.

In order to lower the error probability of not finding a colorful independent set if it exists to ε, we repeat the process of recoloring and running recurrence (DP-γ) $t(\varepsilon)$ times. That is, we want

$$(1-p)^{t(\varepsilon)} \leq \varepsilon.$$

Exploiting that $1+x \leq e^x$ holds for all $x \in \mathbb{R}$, the above inequality is satisfied by any number $t(\varepsilon)$ of recoloring trials that satisfies

$$e^{-pt(\varepsilon)} \leq \varepsilon.$$

Taking the logarithm on both sides and rearranging terms, we get

$$t(\varepsilon) \geq \ln \varepsilon \cdot \frac{1}{-p} = |\ln \varepsilon| \cdot \frac{k^k}{k!}.$$

Using Stirling's lower bound for the factorial, one obtains $k^k/k! \in \mathrm{O}(e^k)$. To conclude the proof, it is now enough to put together the observations that each run of recurrence (DP-γ) with k colors takes $\mathrm{O}(2^k k \cdot n)$ time and that we have to repeat it only $t(\varepsilon) \in \mathrm{O}(|\ln \varepsilon| \cdot e^k)$ times to get an error probability of ε. Thus, the overall procedure takes $\mathrm{O}(|\ln \varepsilon| \cdot (2e)^k k \cdot n)$ time.

We now derandomize the presented algorithm: instead of repeatedly choosing random recolorings $\delta : [\gamma] \to [k]$, we deterministically enumerate the recolorings according to a *k-color coding scheme* [CLSZ07]: a k-color coding scheme $\mathscr{F}$ is a set of recolorings such that, for each subset $C \subseteq [\gamma]$ with $|C| = k$, there is a recoloring $\delta \in \mathscr{F}$ such that the colors in C will be mapped to pairwise distinct colors by δ. That is, whatever colors a colorful independent set I of size k in G might have, there is one recoloring in $\mathscr{F}$ such that I is colorful after recoloring. Thus, the dynamic program (DP-γ) will find it.

A k-color coding scheme $\mathscr{F}$ can be computed in $\mathrm{O}(6.4^k \cdot \gamma)$ time [CLSZ07]. Moreover, it consists of $\mathrm{O}(6.4^k \cdot \gamma)$ recolorings. That is, in $\mathrm{O}(6.4^k \gamma \cdot 2^k k n)$ time, we can run (DP-γ) for each recoloring in $\mathscr{F}$, thus proving (ii). □

Many algorithms that are based on the color-coding technique can be sped up using algebraic techniques [KW09]. It would be interesting to see whether these can also be used to speed up the running time of Theorem 3.4 (at least in the asymptotic sense).

Concluding this section, we note that the color-coding technique as used in Theorem 3.4 for Job Interval Selection could in the same way be applied to

Colorful Independent Set with Lists. However, the result will not be a fixed-parameter algorithm with respect to the parameter "size k of the sought colorful independent set," but with respect to the total number of colors found in the lists of the colorful independent set vertices. Colorful Independent Set with Lists is W[1]-hard parameterized by k, which, as we will see, follows from 2-Union Independent Set being W[1]-hard parameterized by k [Jia10].

3.5.3 Problem kernelization

In this section, we chart the possibilities of kernelization for Job Interval Selection. First, we show that Job Interval Selection does not allow for polynomial-size problem kernels unless the polynomial-time hierarchy collapses. However, we then show that Job Interval Selection does allow for polynomial-size problem kernels when we restrict the colored input graph to be a *proper* interval graph.

Non-existence of polynomial-size problem kernels. In the following, we show that Job Interval Selection does not allow for polynomial-size problem kernels for any of various parameters unless the polynomial-time hierarchy collapses. Specifically, we prove the following theorem:

Theorem 3.5. Job Interval Selection *does not admit a polynomial-size problem kernel with respect to the combined parameter "number γ of colors" and "maximum clique size ω" unless the polynomial-time hierarchy collapses.*

In particular, there are no polynomial-size problem kernels for the combined parameters (ω, k) or (ω, Q), where Q is the number of "live colors" (Definition 3.2).

The second part of the theorem follows from the first part since both k and Q are at most γ.

Proof. To prove the theorem, we employ the cross-composition framework introduced in Section 2.4.4: we combine s instances $x_0, x_1, \ldots, x_{s-1}$ of an NP-hard problem into an instance x^* of Job Interval Selection that is a yes-instance if and only if one of the x_i is. As the NP-hard source problem for the cross-composition we, more specifically, use Job Interval Selection with the restriction that the size k of the sought colorful independent set equals the number γ of colors. We saw in Corollary 3.3 in Section 3.3 that this restriction remains NP-hard.

By Definition 2.9, we may assume that all of the s input instances x_i are members of the same equivalence class of a polynomial equivalence relation.

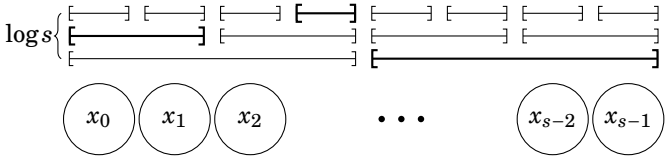


Figure 3.11: Schematic view of the cross-composition for JOB INTERVAL SELECTION. Circles at the bottom represent the s input instances. Above them are $\log s$ rows of auxiliary intervals spanning over the input instances. Here, each of the $\log s$ rows stands for a new color. A colorful independent set of size $k+\log s$ must select one interval in each of the $\log s$ rows (the thick intervals) and the remaining k intervals from the only uncovered of the s input instances (x_2 in this example).

That is, we may assume that all of the s input instances x_i have the same value for k and, thus, each instance uses the same color set $[k]$. We assume, without loss of generality, that s is a power of two (otherwise, we add some "no"-instances to the list of input instances). The steps of the cross-composition are as follows (see Figure 3.11):

1. Place the start and end points of the n intervals of each input instance x_i into the integer range $[i \cdot n, (i+1) \cdot n - 1]$. Herein, we assume each of the input instances to be given in an n-compact representation.

2. Introduce $\log s$ extra colors $k+1, k+2, \ldots, k+\log s$; the resulting instance then asks for a colorful independent set of size $k+\log s$, that is, it has to contain *all* colors.

3. For each $1 \leq i \leq \log s$, introduce 2^i auxiliary intervals $v_0, v_1, \ldots, v_{2^i-1}$ with color $k+i$ such that the auxiliary interval v_j spans exactly over the instances x_ℓ with

$$j \cdot \frac{s}{2^i} \leq \ell \leq (j+1) \cdot \frac{s}{2^i} - 1.$$

To show that this construction is indeed a cross-composition for the parameters "number γ of colors" and "maximum clique size ω," observe that $\gamma, \omega \leq \max_i |x_i| + \log s$. It remains to prove that the constructed instance $(G, k+\log s)$ is a yes-instance if and only if one of the input instances is a yes-instance.

First, if the constructed graph G has a colorful independent set I of size $k+\log s$, then I contains an interval of each color. In particular, I contains an auxiliary interval of each of the colors $k+1$ to $k+\log s$. We show that all of the k non-auxiliary intervals of I are from the same input instance. To this end, note that, for each $1 \leq i \leq \log s$, each auxiliary interval v_j of color $k+i$ spans over exactly

$$(j+1) \cdot \frac{s}{2^i} - j \cdot \frac{s}{2^i} = \frac{s}{2^i} \quad \text{instances.}$$

Since, for any two auxiliary intervals v, w in I, the sets of instances spanned by v and spanned by w are disjoint, the $\log s$ auxiliary intervals in I span over exactly

$$\sum_{i=1}^{\log s} \frac{s}{2^i} = s - 1 \quad \text{instances.}$$

Hence, *exactly one* input instance is *not* spanned, implying that all k non-auxiliary intervals in I are chosen from this very instance.

Second, let x_ℓ be a yes-instance, that is, there is a colorful independent set of size k in x_ℓ that contains the colors $[k]$. We extend this to a colorful independent set of size $k+\log s$ for G. To this end, it is sufficient to add the intervals from a $(\log s)$-*separating* colorful independent set, where a colorful independent set I is i-*separating* for some integer i if

- no interval in I spans x_ℓ,
- I has size i and contains all colors $\{k+1, \ldots, k+i\}$, and
- there is a single interval of color $k+i$ that is not in I and spans over all instances not spanned by the intervals in I.

Obviously, there is a 1-separating colorful independent set, since there are only two intervals with color $k+1$, each spanning over exactly one half of all instances. To complete the proof, it remains to enlarge this 1-separating independent set to be $(\log s)$-separating. To this end, we use induction.

Assume that I is an i-separating colorful independent set for some $1 \leq i < \log s$. We show how to enlarge it to be $(i+1)$-separating. The auxiliary intervals in I span exactly

$$\sum_{j=1}^{i} \frac{s}{2^j} = s - \frac{s}{2^i} \quad \text{instances.}$$

That is, $s/2^i$ instances are not spanned by I but, since I is i-separating, by a single interval of color $k+i$ that is not in I. Since each interval with color $k+i+1$ spans $s/2^{i+1}$ instances and is contained in an interval of color $k+i$, there are precisely two intervals of color $k+i+1$ that span the instances not spanned by I. Since they are disjoint, one of them does not span x_ℓ; add this interval to I. □

Polynomial-size problem kernel on proper interval graphs. Given the result from Theorem 3.5 that Job Interval Selection does not have polynomial-size problem kernels even for the combined parameter (ω, k), where ω is the maximum clique size in the input graph, we now consider Job Interval Selection with the input being restricted to proper interval graphs. This special case remains NP-hard, as shown in Section 3.3. Here, the negative result of Theorem 3.5 collapses and the following data reduction rule will yield a problem kernel containing $4k^2 \cdot \omega$ intervals.

Reduction Rule 3.1. For a graph G and a color c, let $G[c]$ denote the subgraph of G induced by the intervals of color c.

If $G[c]$ has an independent set of size at least $2k-1$, then remove all intervals of color c from G and decrease k by one.

The intuition behind Reduction Rule 3.1 is that, if a color occurs sufficiently often in the input proper interval graph, then we can assume one of the intervals of that color to be part of a maximum colorful independent set. This is formally proven by the following lemma.

Lemma 3.9. *Reduction Rule 3.1 is correct, that is, the produced instance is a yes-instance if and only if the input instance is a yes-instance. Furthermore, Reduction Rule 3.1 can be applied exhaustively in* $O(n)$ *time.*

Proof. Let $(G', k-1)$ denote the instance produced by Reduction Rule 3.1 from an instance (G,k) by removing all intervals of a color c from G. Clearly, a colorful independent set I for G is also a colorful independent set for G' if we remove from I the interval with color c. Hence, if (G,k) is a yes-instance, then so is $(G', k-1)$.

Now, let $(G', k-1)$ be a yes-instance, let I be a colorful independent set of size at least $k-1$ for G' and let I_c be an independent set of size $2k-1$ in $G[c]$. If $|I| \geq k$, then (G,k) is a yes-instance. Otherwise, $|I| \leq k-1$. Furthermore, I does not contain an interval with color c. Consider an interval $u \in I$. If $N_G(u) \cap I_c$ contains at least three intervals x, y, z, then $G[\{u,x,y,z\}]$ is a $K_{1,3}$. Since proper interval

graphs are $K_{1,3}$-free [BLS99, Theorem 7.1.9], this contradicts G being a proper interval graph. Therefore, each interval in I overlaps at most two intervals in I_c. Hence, the intervals in I overlap at most $2|I| \leq 2(k-1) < |I_c|$ intervals, implying $I_c \setminus N_G[I] \neq \emptyset$. Thus, there is an interval in I_c that can be added to I and we obtain a colorful independent set of size at least k for G.

It remains to argue the claimed running time. First, since we assume G to be given in a compact representation, in particular with intervals sorted by their start points, we can compute all graphs $G[c]$ for all colors c in $O(n)$ total time such that their intervals are also sorted by start points. To this end, we just need a single iteration over the intervals of G and copy an interval with color c to the appropriate graph $G[c]$. If the number of vertices in $G[c]$ is n_c, then, for each color c, a maximum independent set in $G[c]$ can be computed in $O(n_c)$ time since $G[c]$ is a *monochromatic* proper interval graph: repeatedly choose an interval v that starts first and delete all intervals that start before v has ended. Hence, computing maximum independent sets for the graphs $G[c]$ for all colors c can be done in $O(n)$ total time. It then remains to count the sizes of the resulting maximum independent sets and delete intervals with color c from G when $G[c]$ has a maximum independent set of size at least $2k-1$.

Since applying Reduction Rule 3.1 to one color does not affect its applicability to other colors, one execution of Reduction Rule 3.1 to all colors is exhaustive, that is, Reduction Rule 3.1 is inapplicable to the resulting instance. □

In order to prove the problem kernel bound, we, moreover, need the following trivial "data reduction rule" that returns a yes-instance if we can greedily find a colorful independent set of size at least k.

Reduction Rule 3.2. Let I be a maximal colorful independent set of G. If $|I| \geq k$, then return a trivial yes-instance.

Lemma 3.10. *Reduction Rule 3.2 is correct and can be applied in* $O(n)$ *time.*

Proof. The correctness of Reduction Rule 3.2 is obvious. It remains to prove the running time.

A maximal colorful independent set of G can be found by repeatedly picking an interval v of a color that has not yet been used and that starts first, marking its color as used, and deleting all intervals that start before v has ended. To check and store in constant time whether a color is used, we employ a size-γ array whose i-th entry is 1 if color i is already used. Hence, the whole procedure can be executed in $O(n)$ time given that the intervals are sorted. □

We can show that these two data reduction rules yield a problem kernel.

Theorem 3.6. Job Interval Selection *on proper interval graphs admits a problem kernel with at most* $4k^2 \cdot \omega$ *intervals that is computable in* $O(n)$ *time. Herein,* ω *is the maximum clique size of the input graph.*

Proof. To show the problem kernel bound, consider an instance (G,k) of Job Interval Selection that is reduced with respect to Reduction Rule 3.1 and to which Reduction Rule 3.2 has been applied. It follows that there is a maximal colorful independent set I of G with $|I| < k$. Since G is a *proper* interval graph, the neighborhood of each interval v can be partitioned into two cliques: one clique contains all intervals intersecting the start point of v, the other clique contains all intervals intersecting the end point of v. Thus, each interval has at most $2\omega - 1$ neighbors and, hence, we can bound $|N[I]| \leq 2k\omega$.

Now, let X denote the set of intervals in G that are not neighbors of any interval in I and let $G' := G[X]$. Then, since I is maximal, all intervals in X have a color that appears in I, of which there are at most $k-1$. For each color c of these, let $G'[c]$ denote the subgraph of G' that is induced by all intervals of color c and let I_c denote a maximum independent set of $G'[c]$. Since G is reduced with respect to Reduction Rule 3.1, $|I_c| \leq 2(k-1)$. Again, since $G'[c]$ is a proper interval graph, each interval $u \in I_c$ has at most $2\omega - 1$ neighbors in $G'[c]$. Thus, the total number of intervals in $G'[c]$ is at most $4(k-1)(\omega-1)$. Since G' contains at most $k-1$ colors, we can bound $|V(G')| \leq 4(k-1)^2(\omega-1)$, implying a bound of $|V(G)| + |N[I]| \leq 4(k-1)^2(\omega-1) + 2k\omega$ for the number of intervals in G.

The running time bound follows from Lemma 3.9 and Lemma 3.10. □

3.6 2-Union Independent Set

In Section 3.5, we studied the Job Interval Selection problem, the special case of 2-Union Independent Set where one of the two input interval graphs is a cluster graph. In this section, we investigate the parameterized complexity 2-Union Independent Set.

Our complexity dichotomy has already shown that 2-Union Independent Set is NP-hard even if the maximum vertex degree of both input interval graphs is at most two (Corollary 3.1). Moreover, it is known that 2-Union Independent Set is W[1]-hard with respect to the standard parameter k [Jia10]. Hence, with respect to these three parameters, 2-Union Independent Set is unlikely to be fixed-parameter tractable.

In contrast, this section shows how the compactness of the input interval graphs affects the computational complexity of 2-Union Independent Set. To this end, as before, we assume that both input interval graphs are $c_\forall$-compact and that at least one of both input interval graphs is $c_\exists$-compact for some integers $c_\forall$ and $c_\exists$ (see Definition 3.1).

First, in Section 3.6.1, we show a fixed-parameter linear-time algorithm with respect to the parameter $c_\exists$. The algorithm is an adaption of our algorithm for Colorful Independent Set with Lists (Theorem 3.3) to 2-Union Independent Set. In the analysis of its complexity, the parameter $c_\exists$ naturally arises as complexity measure.

Second, in Section 3.6.2, we show a simple polynomial-time data reduction rule for 2-Union Independent Set. Again, in the analysis of its effectiveness, the parameter $c_\forall$ naturally arises as complexity measure.

Since in both cases, the compactness parameters arise quite naturally, we suspect that the parameter may be useful in the development of fixed-parameter algorithms for other NP-hard problems on interval graphs.

3.6.1 A dynamic program for compact interval graphs

We describe an algorithm that solves 2-Union Independent Set in $O(2^{c_\exists} c_\exists \cdot n)$ time. To this end, we reformulate 2-Union Independent Set as a special case of Colorful Independent Set with Lists and then solve the resulting instance using the dynamic program (DP-Q^*) from Section 3.5.2 (Theorem 3.3).

An instance of 2-Union Independent Set can be solved by an algorithm for Colorful Independent Set with Lists as follows: without loss of generality, assume that, of the input interval graphs, G_2 is $c_\exists$-compact. We interpret each number in $[c_\exists]$ as a color and give the input graph G_1 as input to Colorful Independent Set with Lists such that each vertex v from G_1 gets the colors corresponding to the numbers contained in the interval that represents v in G_2. Then, a solution for Colorful Independent Set with Lists is a solution for 2-Union Independent Set and vice versa:

- Two vertices v and w may be together in a solution of 2-Union Independent Set if and only if their intervals neither intersect in G_1 nor in G_2.
- Two vertices v and w may be together in a solution of Colorful Independent Set with Lists if and only if neither their intervals in G_1 intersect nor their color lists intersect (which are precisely their intervals in G_2).

We stated earlier that COLORFUL INDEPENDENT SET WITH LISTS is a more general problem than 2-UNION INDEPENDENT SET. This now becomes clear: whereas COLORFUL INDEPENDENT SET WITH LISTS allows arbitrary color lists, the instances generated from 2-UNION INDEPENDENT SET only use intervals of natural numbers as color lists.

To execute the transformation from 2-UNION INDEPENDENT SET to COLORFUL INDEPENDENT SET WITH LISTS, we just take each interval of G_2 and add the numbers that it contains to the color list of the corresponding vertex in G_1. Since each interval in G_2 contains at most $c_\exists$ numbers, the transformation from 2-UNION INDEPENDENT SET to COLORFUL INDEPENDENT SET WITH LISTS is executable in $O(c_\exists \cdot n)$ time. The resulting COLORFUL INDEPENDENT SET WITH LISTS instance has $c_\exists$ colors and, by Theorem 3.3, is solvable in $O(2^{c_\exists} c_\exists \cdot n)$ additional time.

Theorem 3.7. 2-UNION INDEPENDENT SET *is solvable in* $O(2^{c_\exists} c_\exists \cdot n)$ *time when at least one of the input interval graphs is* $c_\exists$*-compact.*

Note that our transformation from 2-UNION INDEPENDENT SET to COLORFUL INDEPENDENT SET WITH LISTS is a polynomial-time many-one reduction that does not change the value of the value k of the sought solution. We call polynomial-time many-one reductions of this form *parameterized reductions* with respect to k. Since 2-UNION INDEPENDENT SET is W[1]-hard parameterized by k [Jia10], this parameterized reduction implies that COLORFUL INDEPENDENT SET WITH LISTS is also W[1]-hard parameterized by k.

3.6.2 Problem kernelization

In this section, we present a polynomial-time data reduction rule for 2-UNION INDEPENDENT SET. It will turn out that the presented data reduction rule yields a polynomial-size problem kernel for the parameter $c_\forall$, assuming that both input interval graphs G_1, G_2 are $c_\forall$-compact.

The intuition behind the data reduction rule is simple: assume that we have a vertex that is represented by the interval v in the first input interval graph G_1 and by v' in the second interval graph G_2. Moreover, assume that there is another vertex represented by the intervals u in G_1 and u' in G_2. Then, if $v \subseteq u$ and $v' \subseteq u'$, we would never choose the vertex represented by u and u' into a maximum independent set, as it "blocks" a superset of vertices for inclusion into a maximum independent set compared to the vertex represented by v and v'. Hence, we delete the intervals u and u'.

In order to lead this intuitive idea to a problem kernel, we introduce the concept of the *signature* of a vertex, give a data reduction rule that bounds the number of vertices having a given signature, and bound the number of signatures in a 2-union graph.

Definition 3.3. Let (G_1, G_2, k) denote an instance of 2-Union Independent Set and let v be a vertex of G_1 and G_2. The *signature* $\text{sig}(v)$ of v is a four-dimensional vector $(-v_s, v_e, -v'_s, v'_e)$, where v_s and v_e are v's start and end points in G_1, and v'_s and v'_e are its start and end points in G_2.

Reduction Rule 3.3. Let (G_1, G_2, k) denote an instance of 2-Union Independent Set. For each pair of vertices u, v of G_1 and G_2 such that $\text{sig}(v) \leq \text{sig}(u)$ (component-wise), delete u from G_1 and G_2.

Lemma 3.11. *Reduction Rule 3.3 is correct and can be applied in* $O(n \log^2 n)$ *time.*

Proof. Let (G_1, G_2, k) be an instance of 2-Union Independent Set and let u, v be vertices of G_1 and G_2 such that $\text{sig}(v) \leq \text{sig}(u)$. Observe that this implies $v_s \geq u_s$, $v_e \leq u_s$, $v'_s \geq u'_s$, and $v'_e \leq u'_e$. Hence, $N_{G_1}[v] \subseteq N_{G_1}[u]$ and $N_{G_2}[v] \subseteq N_{G_2}[u]$ and, therefore, $N_{G_1 \cup G_2}[v] \subseteq N_{G_1 \cup G_2}[u]$. Hence, instead of choosing u into an independent set, we can always choose v. Therefore, it is safe to delete u.

Regarding the running time, Kung, Luccio, and Preparata [KLP75] have shown that the set of maxima of n vectors in d dimensions can be computed in $O(n \log^{d-2} n)$ time. □

Theorem 3.8. *For both input interval graphs being* $c_\forall$*-compact,* 2-Union Independent Set *admits a problem kernel with* $c_\forall^3$ *vertices that can be computed in* $O(n \log^2 n)$ *time.*

The problem kernel size reduces to $2c_\forall^2$ *vertices if one of the input graphs is a proper interval graph.*

Proof. Let (G_1, G_2, k) be an instance of 2-Union Independent Set. We assume that G_1 and G_2 have been preprocessed according to Observation 3.3, that is, at each position of the interval representations of G_1 and G_2, there is an interval start point as well as an interval end point. The problem kernel (G_1^*, G_2^*, k) is then obtained from (G_1, G_2, k) by applying Reduction Rule 3.3 to (G_1, G_2, k). By definition of G_1^* and G_2^*, the graphs G_1^* and G_2^* contain at most one vertex of each signature. Hence, it is sufficient to show that there are at most $c_\forall^3$ different signatures corresponding to vertices in the new instance (G_1^*, G_2^*, k).

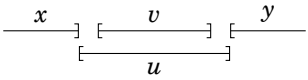


Figure 3.12: The constellation $u_s = x_e < v_s$ and $v_e < y_s = u_e$ inducing a $K_{1,3}$ in G_1.

Consider the set $\mathscr{S}_{i,j}$ of all signatures $s = (-v_s, v_e, -v'_s, v'_e)$ with $v_s = i$ and $v'_s = j$ such that v remains in G_1^* and G_2^*. If $|\mathscr{S}_{i,j}| > c_\forall$, then we find $s_1, s_2 \in \mathscr{S}_{i,j}$ such that s_1 and s_2 agree in the second or fourth coordinate, since there are at most $c_\forall$ possible values for each of them. Since then s_1 and s_2 agree in three coordinates, it follows that either $s_1 \leq s_2$ or $s_2 \leq s_1$, contradicting the assumption that Reduction Rule 3.3 has been applied to (G_1^*, G_2^*, k). Obviously, there are at most $c_\forall^2$ sets of signatures $\mathscr{S}_{i,j}$ and, thus, there are at most $c_\forall^3$ signatures in total.

We now show that the described instance has at most $2c_\forall^2$ vertices if G_1 is a proper interval graph. We will use two relations R_1, R_2 between pairs $(i,j) \in [c_\forall] \times [c_\forall]$ and signatures corresponding to vertices of G_1^*. We then show that every signature is an image under one of R_1 and R_2 and that both relations are in fact functions, that is, they map each pair to at most one signature. This proves that there are at most $2 \cdot |[c_\forall] \times [c_\forall]| = 2c_\forall^2$ signatures corresponding to vertices in G_1^*. Since G_1^* contains at most one vertex per signature, the theorem will follow.

We define the relations R_1 and R_2 as follows: for a pair (i,j), the relation R_1 associates (i,j) with all signatures $s = \text{sig}(v) = (-v_s, v_e, -v'_s, v'_e)$ of $\mathscr{S}_{i,j}$ that minimize v'_e. For all signatures $s = \text{sig}(w) = (-w_s, w_e, -w'_s, w'_e)$ with w remaining in G_1^* and that are *not* images under R_1, the relation R_2 associates (w_e, w'_s) with s. By definition, every signature is the image of some pair under *either* R_1 or R_2.

Observe that R_1 is a function: since G_1^* and G_2^* are reduced with respect to Reduction Rule 3.3, for each pair $(i,j) \in [c_\forall] \times [c_\forall]$, there is at most one signature $s = (-v_s, v_e, -v'_s, v'_e) \in \mathscr{S}_{i,j}$ that minimizes v'_e.

It remains to show that R_2 is also a function. Towards a contradiction, assume that R_2 maps some pair to two signatures. Then, there are distinct vertices v and w in G_1^* with signatures $s_1 := \text{sig}(v) = (-v_s, v_e, -v'_s, v'_e)$ and $s_2 := \text{sig}(w) = (-w_s, w_e, -w'_s, w'_e)$ such that $(v_e, v'_s) = (w_e, w'_s)$. Since the vertices v and w are in G_1^*, we know that $v_s \neq w_s$, since otherwise $s_1 \leq s_2$ or $s_2 \leq s_1$. By symmetry, let $w_s < v_s$. Since $s_2 \in \mathscr{S}_{w_s, w'_s}$, and, therefore, $\mathscr{S}_{w_s, w'_s}$ is nonempty, there is a signature $s_3 := R_1(w_s, w'_s) = \text{sig}(u) = (-u_s, u_e, -u'_s, u'_e)$. Since s_2 is an image under R_2, it is not an image under R_1 and, thus, we have $s_2 \neq s_3$. By definition of R_1, $u_s = w_s$, $u'_s = w'_s$, and $u'_e \leq w'_e$. Therefore, we have $u_e > w_e$ since, otherwise, $s_3 \leq s_2$.

Since $u_s = w_s < v_s$ and $v_e = w_e < u_e$, we now have a constellation $u_s < v_s \leq v_e < u_e$ of intervals that contradicts G_1 being a proper interval graph: since G_1 has been preprocessed according to Observation 3.3, the start point u_s is also the end point x_e of some interval x and the end point u_e is also the start point y_s of some interval y. This, as depicted in Figure 3.12, implies that G_1 contains a $K_{1,3}$ as induced subgraph, which contradicts G_1 being a proper interval graph [BLS99, Theorem 7.1.9]. The $K_{1,3}$ consists of the central vertex u and the leaves v, x, y. □

We can generalize Theorem 3.8 for the problem of finding an independent set of *weight* at least k: we only have to keep the vertex for each signature in the graph that has the highest weight. Since there are at most $c_\forall^4$ different signatures, we obtain a problem kernel with $c_\forall^4$ vertices for the weighted variant of 2-Union Independent Set.

3.7 Experimental evaluation

In this section, we demonstrate to which extent instances of Colorful Independent Set with Lists are solvable within five minutes. Herein, we chose Colorful Independent Set with Lists (see Section 3.2.2 for the definition) since it is the most general problem studied in this chapter and algorithms for it also solve 2-Union Independent Set and Job Interval Selection.

We implemented the dynamic programming algorithm (DP-Q*) from Section 3.5.2 that solves Colorful Independent Set with Lists in $O(2^Q \gamma \cdot n)$ time and $O(2^Q \ell + \gamma c)$ space on c-compact interval graphs, where ℓ is the maximum length of an interval and Q is the parameter "maximum number of live colors" (Definition 3.2 of Section 3.5.2). We applied the implemented algorithm to randomly generated instances.

Note that we abstained from implementing our data reduction rules (Sections 3.5.3 and 3.6.2), since they do not apply to the most general form of Colorful Independent Set with Lists.

Implementation details. The implementation of the algorithm is based on recurrence (DP-Q*) from Section 3.5.2, but allows the vertices to have weights and finds a colorful independent set of maximum weight. Moreover, all set operations and table look-ups have been implemented to take constant time instead of $O(\gamma)$ time by representing each set as a 64 bit integer and implementing set oper-

ations by bitwise logical operations on these integers. Thus, the implementation is limited to COLORFUL INDEPENDENT SET WITH LISTS with $\gamma \leq 64$ colors.

The source code uses about 700 lines of C++ and is freely available.[1] The experiments were run on a computer with a 3.6 GHz Intel Xeon processor and 64 GB RAM under Linux 3.2.0, where the source code has been compiled using the GNU C++ compiler in version 4.7.2 and using the highest optimization level (-O3).

Data. In order to test the influence of various parameters on the running time and memory usage of the algorithm, we evaluated the algorithm on artificial, randomly generated data. To generate random interval graphs, we use a model resembling that of Scheinerman [Sch88]. However, while Scheinerman [Sch88] chooses integer interval endpoints uniformly at random without repetitions from $[2n]$, we choose integer interval endpoints uniformly at random from $[c]$, where c is a maximum compactness chosen in advance. It then remains to assign colors and weights to the vertices.

In detail, to generate a random interval graph, we fix a maximum compactness c, a maximum number γ of colors, and a number n of intervals to generate. We then randomly generate n intervals: for each interval v, we choose a start point v_s and an end point v_e uniformly at random from $[c]$. Then, we add each color in $[\gamma]$ to the color list of v with probability 1/2 and uniformly at random assign to v a weight from one to ten.

To interpret the experimental results, it is important to make some structural observations about the data generated by this random process.

1. The maximum interval length ℓ is at most $c-1$. Moreover, with a growing number n of generated intervals, the probability $(1-1/c^2)^n$ of *not* generating an interval that indeed has length $c-1$ approaches zero. That is, for large input graphs, we expect the chosen parameter $c \approx \ell + 1$ to linearly influence the memory usage of the algorithm (Theorem 3.3).

2. The maximum number of live colors Q is at most the number γ of colors. However, since every interval contains each color with equal probability, with increasing number n of intervals, we will have $Q \approx \gamma$. Hence, we expect γ to exponentially influence the running time and memory usage of the algorithm (Theorem 3.3).

3. The sizes of the generated vertex color lists follow a binomial distribution. The expected color list size is $\gamma/2$.

[1] http://fpt.akt.tu-berlin.de/cis/

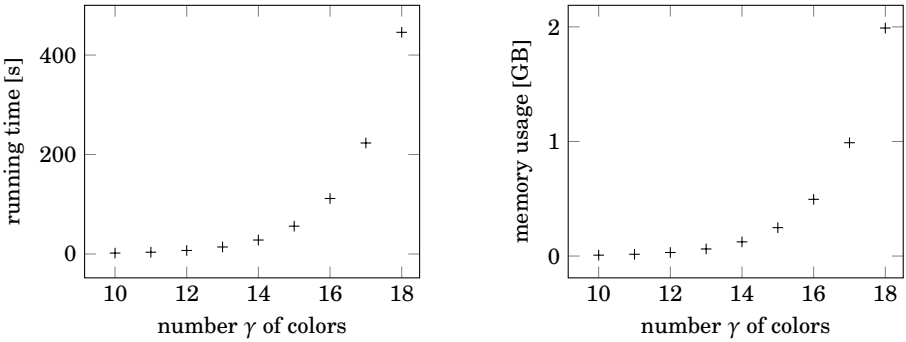


Figure 3.13: Dependence of running time and space requirements of the dynamic program (DP-Q*, Section 3.5.2) for Colorful Independent Set with Lists on the number γ of colors in the input interval graph, each having 10^5 intervals and being 10^3-compact.

Experimental results. In all plots to be shown, each point is obtained from a single run of our algorithm; the running times and memory usage are not averaged in any way. Although our input graphs being random, we will see almost no fluctuation in running times and memory usage of the algorithm, since the algorithm is quite insensitive to the structure of the input data: for each input interval, it iterates over all subsets of live colors—it has no "early exit strategy." In Chapter 4, we will see that our search tree algorithm for DAG Partitioning is significantly more sensitive to the structure of the input data.

For the experiments, we generated three data sets by varying each time one of the parameters $\{n, \gamma, c\}$ and keeping the other two parameters constant.

For the first data set, we let the number γ of colors vary between 10 and 18 and fixed $n = 10^5$ and $c = 10^3$. Figure 3.13 clearly exhibits the exponential dependence of running time and memory usage on the number γ of colors. We see that, in this setup, we can solve Colorful Independent Set with Lists within a time frame of five minutes for $\gamma \leq 17$.

For the second data set, we let the number n of intervals vary between 10^5 and $8 \cdot 10^5$. We again fixed $c = 10^3$. We chose $\gamma = 15$ as number of colors. As expected, Figure 3.14 shows a linear dependence of the running time on the number n of intervals. Moreover, the memory usage is almost constantly about 250 MB with a slight increase at the beginning, since we left the compactness c

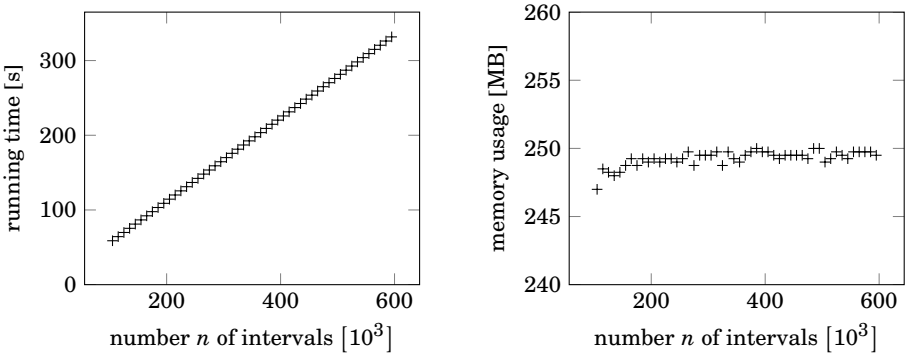


Figure 3.14: Dependence of running time and space requirements of the dynamic program (DP-Q*, Section 3.5.2) for Colorful Independent Set with Lists on the number n of intervals in the input interval graph, each being colored with subsets of $\{1,\dots,15\}$ and being 10^3-compact.

constant and, with increasing number n of intervals, the maximum interval length ℓ approaches the maximum compactness c.

For the third data set, we finally let the compactness c vary between 100 and 1000. We again fixed $\gamma = 15$ and $n = 10^5$. Figure 3.15 shows the linear dependence of the memory usage on the compactness $c \approx \ell + 1$. In contrast, the running time remains roughly constant with increasing c. The observed local minima of the running time are exactly at those values of c where c is a power of two, where we observed table look-ups to work more efficiently.

Summary. The running time and memory usage of the algorithm on randomly generated data very reliably behave as predicted by Theorem 3.3 and most likely scale to larger data. We have seen that, on moderate values of $\gamma \leq 15$, the algorithm can solve instances with up to $5.5 \cdot 10^5$ intervals in a time frame of about five minutes. Instances of this scale can model complex scheduling tasks like those in our aircraft maintenance problems in Examples 3.2 and 3.3. However, in the steel manufacturing instances described by Höhn et al. [HKML11] and sketched in Example 3.1, the number of colors in the resulting instances of Colorful Independent Set with Lists exceeds one hundred.

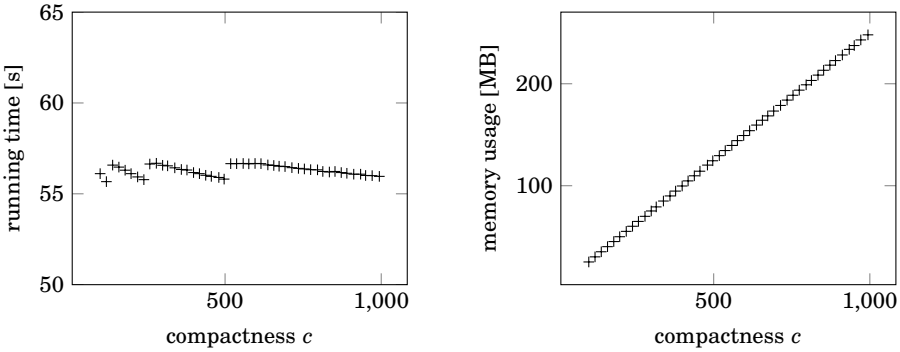


Figure 3.15: Dependence of running time and space requirements of the dynamic program (DP-Q*, Section 3.5.2) for COLORFUL INDEPENDENT SET WITH LISTS on the compactness c of the input interval graph, each having 10^5 intervals colored using subsets of $\{1,\ldots,15\}$.

In order to solve instances with many colors using our algorithm efficiently, it is crucial that these instances have a low maximum number Q of "live colors", that is, these instances must be more structured than our randomly generated interval graphs. This case may occur, for example, in our Examples 3.2 and 3.3, where the number of live colors remains small with increasing number of airplanes and crews if there are groups of hangars, each having distinct crews and airplanes. Again, however, in the way Höhn et al. [HKML11] model their steel manufacturing problem using 2-UNION INDEPENDENT SET, all colors in the resulting COLORFUL INDEPENDENT SET WITH LISTS instance are "live."

3.8 Conclusion

We charted the complexity landscape of 2-UNION INDEPENDENT SET and JOB INTERVAL SELECTION, which model various scheduling tasks. Our focus was on finding exact solutions, whereas, so far, approximations have been much better researched in the literature [Bar+06; BNR96; COR06; Spi99].

Besides hardness results from our complexity dichotomy, we provided first results on problem kernelization in this context. In the form of fixed-parameter linear-time algorithms, we found encouraging algorithmic results. Herein,

working with INDEPENDENT SET problems on vertex-colored interval graphs was beneficial and we believe that it may be also beneficial in future to work on vertex-colored interval graphs instead of edge-wise unions of interval graphs.

Experiments showed that instances of a scale that can model complex scheduling tasks are solvable within a few minutes.

For future work, it would be interesting to determine whether 2-UNION INDEPENDENT SET is fixed-parameter tractable or even fixed-parameter linear with respect to the "M-compositeness" parameter that is small in the steel manufacturing application considered by Höhn et al. [HKML11]. Moreover, it seems worthwhile trying to speed up our randomized algorithm for JOB INTERVAL SELECTION (Theorem 3.4) using the algebraic techniques described by Koutis and Williams [KW09].

4 DAG Partitioning

Finding the origin of short phrases propagating through the web has been formalized by Leskovec, Backstrom, and Kleinberg [LBK09] as DAG PARTITIONING: given an arc-weighted directed acyclic graph on n vertices and m arcs, delete arcs with total weight at most k so that each resulting weakly connected component contains exactly one sink—a vertex without outgoing arcs. DAG PARTITIONING is NP-hard even on directed acyclic graphs with uniform arc weights and two sinks.

We show an algorithm to solve DAG PARTITIONING in $O(2^k \cdot (n+m))$ time, that is, in linear time for fixed k. We complement it with linear-time data reduction rules. Experiments show that, in combination, they can solve DAG PARTITIONING on simulated citation networks within five minutes for $k \leq 190$ and m being 10^7 and larger. We compare our algorithm to Leskovec, Backstrom, and Kleinberg's [LBK09] heuristic, observing that the heuristic mostly finds optimal solutions in those graphs where our algorithm runs fast, but performs worse by a factor of more than two on other instances.

We show a simple data-reduction based linear-time algorithm to solve DAG PARTITIONING on unweighted directed acyclic graphs whose underlying undirected graphs are trees. The result generalizes to graphs of constant treewidth. We complement our algorithms with lower bounds on the running time of fixed-parameter algorithms and on the size of problem kernels with respect to the parameter k.

4.1 Introduction

The DAG PARTITIONING problem was introduced by Leskovec, Backstrom, and Kleinberg [LBK09] in order to analyze how short, distinctive phrases (typically, parts or mutations of quotations, also called *memes*) spread to various news sites and blogs. To demonstrate their approach, Leskovec, Backstrom, and Kleinberg [LBK09] collected phrases from 90 million articles from the time around the United States presidential elections of 2008 and analyzed their way through the internet; the results were featured in the New York Times [Loh09].

Meanwhile, the ideas of Leskovec, Backstrom, and Kleinberg [LBK09] have grown into NIFTY, a system that allows for near real-time observation of the rise and fall of trends, ideas, and topics on the internet [Sue+13].

The role of DAG PARTITIONING in the approach of Leskovec, Backstrom, and Kleinberg [LBK09] is clustering short phrases, which may undergo modifications while propagating through the web, with respect to their origins. To this end, Leskovec, Backstrom, and Kleinberg [LBK09] create a directed graph with phrases as vertices and draw an arc from a phrase p to a phrase q if p presumably originates from q, where the weight of an arc represents the support for this hypothesis. Herein, they draw arcs only from shorter phrases to longer phrases and, thus, obtain a directed acyclic graph. A vertex without outgoing arcs in this graph is called a *sink* and can be interpreted as the origin of a phrase. If a phrase has directed paths to more than one sink, its ultimate origin is ambiguous; some of the introduced "p originates from q" hypotheses must be wrong. Leskovec, Backstrom, and Kleinberg [LBK09] use a heuristic for DAG PARTITIONING in order to resolve these inconsistencies by removing a set of arcs (hypotheses) with low support:

DAG PARTITIONING
Input: A directed acyclic graph G with positive integer arc weights $\omega\colon A(G) \to \mathbb{N}$ and a positive integer $k \in \mathbb{N}$.
Question: Is there a *partitioning set* $S \subseteq A(G)$ with $\sum_{a \in S} \omega(a) \leq k$ such that each weakly connected component in $G \setminus S$ has exactly one sink?

Herein, the model of Leskovec, Backstrom, and Kleinberg [LBK09] exploits the fact that a weakly connected component of a directed acyclic graph contains exactly one sink if and only if all its vertices have directed paths to exactly one sink; we will see this in the next section. Note that, in the following, we will always use the term *connected* to mean *weakly connected*.

In contrast to Leskovec, Backstrom, and Kleinberg [LBK09] and Suen et al. [Sue+13], we aim for solving DAG PARTITIONING optimally. To this end, we employ fixed-parameter linear-time algorithms. Herein, a natural parameter to consider is the weight k of the sought partitioning set, since one would expect that wrong hypotheses have low support.

4.1.1 Known results

To date, there are only few studies on DAG PARTITIONING. Leskovec, Backstrom, and Kleinberg [LBK09] showed that DAG PARTITIONING is NP-hard and present heuristic solution algorithms. Alamdari and Mehrabian [AM12] showed that,

if P $\neq$ NP, then, for any fixed $\varepsilon > 0$, there is no polynomial-time algorithm for computing a partitioning set whose weight is within a factor of $O(n^{1-\varepsilon})$ of the minimum weight. This even holds if the input graph has unit-weight arcs, maximum outdegree three, and only two sinks. Moreover, Alamdari and Mehrabian [AM12] showed that DAG PARTITIONING can be solved in $2^{O(t^2)} \cdot n$ time given a width-t path decomposition of the input graph.

DAG PARTITIONING is very similar to the well-known NP-hard MULTIWAY CUT problem [Dah+94]: given an undirected edge-weighted graph and a subset of the vertices called *terminals*, delete edges of total weight at most k such that each terminal is separated from all others. DAG PARTITIONING can be considered as MULTIWAY CUT with the sinks being terminals and the additional constraint that not all arcs outgoing from a vertex may be deleted, since this would create a new sink. Xiao [Xia10] gives an algorithm to solve MULTIWAY CUT in $O(2^k \cdot \min(n^{2/3}, m^{1/2}) \cdot nm)$ time. Interestingly, in contrast to DAG PARTITIONING, MULTIWAY CUT is constant-factor approximable [Kar+04].

4.1.2 Our results

We provide algorithmic as well as intractability results. On the algorithmic side, we show an $O(2^k \cdot (n+m))$ time algorithm for DAG PARTITIONING and complement it with linear-time data reduction rules. We experimentally evaluate both and observe that, in combination, the algorithm and the data reduction solve instances with $k \leq 190$ optimally within five minutes, the number of input arcs being 10^7 and larger. We use the optimal solutions found by our algorithm to evaluate the quality of Leskovec, Backstrom, and Kleinberg's [LBK09] heuristic, observing that it mostly finds optimal solutions in those graphs where our algorithm runs fast, but performs worse by a factor of more than two on other instances.

We show a simple data-reduction based algorithm to solve DAG PARTITIONING in linear time on unweighted directed acyclic graphs whose underlying undirected graphs are trees. The result can be generalized to graphs of constant treewidth in order to improve Alamdari and Mehrabian's [AM12] algorithm for graphs of constant pathwidth.

On the side of intractability results, we strengthen the NP-hardness results of Leskovec, Backstrom, and Kleinberg [LBK09] and Alamdari and Mehrabian [AM12] to graphs of diameter two and to graphs of maximum degree three. Moreover, we show lower bounds on the running time of fixed-parameter algorithms as well as on the size of problem kernels for DAG PARTITIONING parameterized by k.

4.1.3 Chapter outline

First, in Section 4.2, we make two structural observations about partitioning sets that we will exploit in our proofs.

Section 4.3 considers DAG PARTITIONING parameterized by the weight k of the sought partitioning set. Here, we present our $O(2^k \cdot (n+m))$ time algorithm and its experimental evaluation. Moreover, we show lower bounds on the running time of fixed-parameter algorithms as well as on the size of problem kernels.

Section 4.4 presents a data-reduction based linear-time algorithm for DAG PARTITIONING on unweighted directed acyclic graphs whose underlying undirected graphs are trees. The result generalizes to a fixed-parameter linear-time algorithm for DAG PARTITIONING parameterized by the treewidth t of the input graph.

Section 4.5 shows that parameters other than k or the treewidth t are not as helpful in solving DAG PARTITIONING: the problem remains NP-hard even when parameters like the diameter or maximum degree are constants.

4.2 The structure of minimal partitioning sets

This section shows two simple structural observations about the structure of minimal partitioning sets that we will frequently exploit in our proofs.

The first observation does away with a seeming inaccuracy in modeling the task of clustering phrases with respect to their origins using DAG PARTITIONING: recall that our aim was to modify the input graph so that every vertex, which represents a phrase, has a *directed path* to exactly one sink, which is interpreted as the phrase's origin. Yet, Leskovec, Backstrom, and Kleinberg [LBK09] modeled this task as DAG PARTITIONING, where the task is finding a *partitioning set*—a set whose removal from the input graph results in a graph such that every vertex has an *undirected path* to exactly one sink. Herein, Leskovec, Backstrom, and Kleinberg [LBK09] exploited that these tasks are indeed equivalent:

Observation 4.1. Let G be a directed acyclic graph. An arc set S is a partitioning set for G if and only if each vertex in G can reach exactly one sink in $G \setminus S$.

Proof. If S is a partitioning set for G, then, by definition, each connected component of $G \setminus S$ contains exactly one sink. Therefore, each vertex in $G \setminus S$ can reach *at most* one sink. Moreover, since $G \setminus S$ is a directed acyclic graph and each vertex in a directed acyclic graph can reach *at least* one sink, it follows that each vertex in $G \setminus S$ can reach *exactly* one sink.

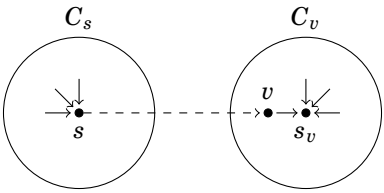


Figure 4.1: For a partitioning set S containing the dashed arc, two connected components C_s and C_v of a graph $G \setminus S$ are shown, where s_v is the only sink in C_v and s is the only sink in C_s.

Now, assume that each vertex in $G \setminus S$ can reach exactly one sink. We show that each connected component of $G \setminus S$ contains exactly one sink. For the sake of contradiction, assume that a connected component C of $G \setminus S$ contains multiple sinks $s_1, \dots, s_t$. For $i \in [t]$, let A_i be the set of vertices that reach s_i. These vertex sets are pairwise disjoint and, since every vertex in a directed acyclic graph reaches some sink, they partition the vertex set of C.

Since C is a connected component, there are $i, j \in [t]$ with $i \neq j$ and some arc (v, w) in $G \setminus S$ from some vertex $v \in A_i$ to some vertex $w \in A_j$. This is a contradiction, since v can reach s_i as well as s_j. □

The second observation is that a minimal partitioning set does not introduce new sinks. It is probably the most frequently used property of partitioning sets in this chapter and has implicitly already been used by Alamdari and Mehrabian [AM12], Leskovec, Backstrom, and Kleinberg [LBK09], and Suen et al. [Sue+13].

Observation 4.2. Let G be a directed acyclic graph and S be a minimal partitioning set for G. Then, a vertex is a sink in $G \setminus S$ if and only if it is a sink in G.

Proof. Clearly, deleting arcs from a directed acyclic graph cannot turn a sink into a non-sink. Therefore, it remains to show that every sink in $G \setminus S$ is a sink in G. The proof is illustrated in Figure 4.1.

Towards a contradiction, assume that there is a vertex s that is a sink in $G \setminus S$ but not in G. Then, there is an arc (s, v) in S for some vertex v of G. Let C_v and C_s be the connected components in $G \setminus S$ containing v and s, respectively, and let s_v be the sink in C_v. Then, for $S' := S \setminus \{(s, v)\}$, the connected component $C_v \cup C_s$ in $G \setminus S'$ has only one sink, namely s_v. Thus, S' is a partitioning set for G with $S' \subsetneq S$, contradicting the minimality of S. □

4.3 Finding constant-weight partitioning sets in linear time

This section considers DAG Partitioning parameterized by the weight k of the sought partitioning set. First, in Section 4.3.1, we present a fixed-parameter linear-time algorithm for DAG Partitioning. The algorithm is based on the search tree paradigm, for which we have already seen a simple example in Section 3.5.1 to solve Job Interval Selection.

In Section 4.3.2, we design linear-time data reduction rules for DAG Partitioning, since experiments will show that our search tree algorithm alone cannot solve large instances.

In Section 4.3.3, we investigate the question whether the provided data reduction rules might have a provable shrinking effect on the input instance in form of a polynomial-size problem kernel. Unfortunately, we will see that polynomial-size problem kernels presumably do not exist. Moreover, Section 4.3.3 also shows lower bounds on the running time of fixed-parameter algorithms for DAG Partitioning.

In Section 4.3.4, we experimentally evaluate our fixed-parameter linear-time algorithm and see that, although not yielding polynomial-size problem kernels, the data reduction significantly speeds up our algorithm.

4.3.1 A search tree algorithm

We now present an algorithm to compute partitioning sets of constant weight k in linear time. Interestingly, although both problems are NP-hard, it will turn out that DAG Partitioning is substantially easier to solve than the closely related Multiway Cut problem, for which a sophisticated algorithm running in $O(2^k \min(n^{2/3}, m^{1/2})nm)$ time was given by Xiao [Xia10]. This is in contrast to Multiway Cut being constant-factor approximable [Kar+04], while DAG Partitioning is inapproximable unless P = NP [AM12].

The main structural advantage of DAG Partitioning over Multiway Cut is the alternative characterization of partitioning sets given in Observation 4.1: we only have to decide which sink each vertex v will reach in the optimally partitioned graph. To this end, we first decide the question for all out-neighbors of v. This allows us to solve DAG Partitioning by the simple search tree algorithm shown in Algorithm 4.1, which we explain in the proof of the following theorem.

Theorem 4.1. DAG Partitioning *is solvable in* $O(2^k \cdot (n+m))$ *time.*

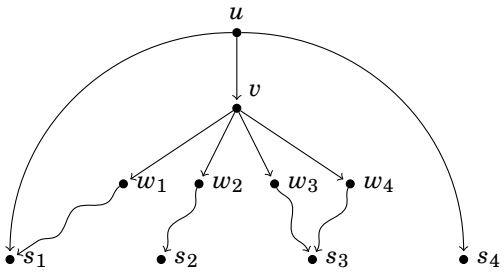


Figure 4.2: For each out-neighbor $w_1,\dots,w_4$ of v, we already know which of the sinks $s_1,\dots,s_4$ it can reach in $G \setminus S$, where S is a minimal partitioning set. This is indicated by the wavy lines. The *feasible* sinks for v in this example are s_1, s_2, and s_3.

Proof. Before explaining Algorithm 4.1, we first analyze the structure of any minimal partitioning set S for G. It is crucial to note that a vertex v is *connected* to a sink s in $G \setminus S$, that is, via an *undirected path*, if and only if v *can reach* that sink s in $G \setminus S$, that is, via a *directed path*. This is because each connected component of $G \setminus S$ is a directed acyclic graph and each vertex in a directed acyclic graph can reach at least one sink.

Now, consider a vertex v of G and assume that we know, for each out-neighbor w of v, the sink s that w can reach in $G \setminus S$. We call a sink s of G *feasible* for v if an out-neighbor of v can reach s in $G \setminus S$. This is illustrated in Figure 4.2. Let D be the set of feasible sinks for v. Since v can be connected to only one sink in $G \setminus S$, at least $|D|-1$ arcs outgoing from v are deleted by S. However, S does not disconnect v from all sinks in D, since then S would delete all arcs outgoing from v, contradicting Observation 4.2. Hence, exactly one sink $s \in D$ must be reachable by v in $G \setminus S$. For each such sink $s \in D$, the partitioning set S has to delete at least $|D|-1$ arcs outgoing from v. We simply try out all these possibilities, which gives rise to the search tree algorithm presented in Algorithm 4.1.

Algorithm 4.1 starts with $S = \emptyset$ and processes the vertices of G in reverse topological order, that is, each vertex is processed by procedure "searchtree" after its out-neighbors. The procedure exploits the invariant that, when called with a vertex v and a set S, each out-neighbor w of v has already been *associated* with the sink s that it reaches in $G \setminus S$ (in terms of Algorithm 4.1, $L[v] = s$). It then tries all possibilities of associating v with a feasible sink and augmenting S

Algorithm 4.1: Compute a partitioning set of weight at most k.

Input: A directed acyclic graph $G = (V, A)$ with arc weights ω.
Output: A partitioning set S of weight at most k, if it exists.

```
1  (v_1, v_2, ..., v_n) ← reverse topological order of the vertices of G
2  L ← array with n entries, each initialized to ⊥
3  searchtree(1, ∅)                          // start with vertex v_1 and S = ∅

4  Procedure searchtree(i, S)
5      while v_i is a sink or ∃s ∈ V : ∀w ∈ N^out(v_i) : L[w] = s do
6          L[v_i] ← s                        // associate v_i with sink s
7          i ← i + 1                         // continue with next vertex
8      if i > n then                         // all vertices have been handled, S is a partitioning set
9          if ω(S) ≤ k then output S
10     else
11         D ← {L[w] | w ∈ N^out(v_i)}       // set of at least two feasible sinks for v_i
12         if |D| − 1 ≤ k − ω(S) then        // check whether we may delete |D| − 1 arcs
13             foreach s ∈ D do              // try to associate v_i to each feasible sink s
14                 L[v_i] ← s
15                 S' ← S ∪ {(v_i, w) | w ∈ N^out(v_i) ∧ L[w] ≠ s}
16                 searchtree(i + 1, S')
```

accordingly, so that the invariant also holds for the successor of v in the reverse topological order.

Specifically, if v is a sink, then line 6 associates v with itself. If all out-neighbors of v are associated with the same sink s, then line 6 associates v with s. Otherwise, line 11 computes the set D of feasible sinks for v. In line 13, the algorithm branches into all possibilities of associating v with one of the $|D|$ feasible sinks $s \in D$ (by way of setting $L[v] \leftarrow s$ in line 14) and augmenting S so that v only reaches s in $G \setminus S$. That is, in each of the $|D|$ branches, line 15 adds to S the at least $|D| - 1$ arcs outgoing from v to the out-neighbors that are associated with sinks different from s. Hence, in each branch, the weight of S increases by at least $|D| - 1$. Then, line 16 continues with the next vertex in the reverse topological order. After processing the last vertex, each vertex of $G \setminus S$ can reach exactly one sink. By Observation 4.1, S is a partitioning set. If a branch finds a partitioning set of weight at most k, line 9 outputs it.

We analyze the running time of this algorithm. To this end, we first bound the total number of times that procedure "searchtree" is called. To this end, we analyze the number of *terminal calls*, that is, calls that do not recursively call the procedure. Let $T(\alpha)$ denote the maximum possible number of terminal calls caused by procedure "searchtree" when called with a set S satisfying $\omega(S) \geq \alpha$. Note that procedure "searchtree" calls itself only in line 16, that is, for each sink s of some set D of feasible sinks with $1 \leq |D| - 1 \leq k - \alpha$, it calls itself with a set S' of weight at least $\alpha + |D| - 1$. Thus, we have

$$T(\alpha) \leq |D| \cdot T(\alpha + |D| - 1).$$

We now inductively show that $T(\alpha) \leq 2^{k-\alpha}$ for $0 \leq \alpha \leq k$. Then, it follows that there is a total number of $T(0) \leq 2^k$ terminal calls. For the induction base case, observe that $T(k) = 0$, since, if procedure "searchtree" is called with a set S of weight at least k, then any recursive call is prevented by the check in line 12. Now, assume that $T(\alpha') \leq 2^{k-\alpha'}$ holds for all α' with $\alpha \leq \alpha' \leq k$. We show $T(\alpha - 1) \leq 2^{k-(\alpha-1)}$ by exploiting $2 \leq |D| \leq 2^{|D|-1}$ as follows:

$$\begin{aligned} T(\alpha - 1) &\leq |D| \cdot T(\alpha - 1 + |D| - 1) \leq |D| \cdot 2^{k-(\alpha-1)-(|D|-1)} \\ &\leq |D| \cdot \frac{2^{k-(\alpha-1)}}{2^{|D|-1}} \leq 2^{k-(\alpha-1)}. \end{aligned}$$

It follows that there are at most $T(0) = 2^k$ terminal calls to procedure "searchtree." Since every non-terminal call makes at least two new calls, it follows that there are $O(2^k)$ total calls to procedure "searchtree." For each such call, we iterate, in the worst case, over all out-neighbors of all vertices in the graph in lines 5–11, which works in $O(n+m)$ time. Moreover, for each call of procedure "searchtree," we compute a set S' in line 15 in $O(n+m)$ time. Hence, a total amount of $O(2^k \cdot (n+m))$ time is spent in procedure "searchtree." Initially, Algorithm 4.1 uses $O(n+m)$ time to compute a reverse topological ordering [CLRS01, Section 22.4]. □

Algorithm 4.1 has as special case the DAG PARTITIONING heuristic developed by Leskovec, Backstrom, and Kleinberg [LBK09], which is in more detail explained in the follow-up work by Suen et al. [Sue+13]: instead of trying out all possibilities of associating a vertex with a feasible sink, the heuristic just associates each vertex with the sink it would be most expensive to disconnect from.

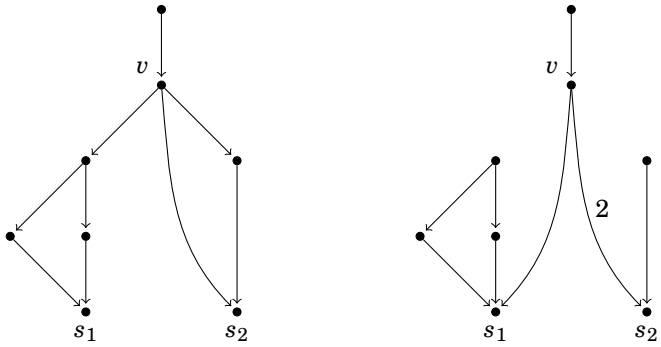


Figure 4.3: On the left hand side, a directed acyclic graph with unit weights is shown. The right hand side shows the same graph to which Reduction Rule 4.1 has been applied exhaustively. Since v can reach multiple sinks and each of its out-neighbors can reach only one sink, its arcs got redirected. Arc labels represent arc weights, where unlabeled arcs have weight one.

4.3.2 Linear-time data reduction

The experiments in Section 4.3.4 will show that Algorithm 4.1 alone cannot even solve moderately large instances. Therefore, we complement it with linear-time data reduction rules that will show a significant speedup. However, we will also see in Section 4.3.3 that the provided data reduction rules presumably cannot yield polynomial-size problem kernels. The first of two data reduction rules reads as follows and is illustrated in Figure 4.3.

Reduction Rule 4.1. If there is an arc (v,w) such that w can reach exactly one sink $s \neq w$ and v can reach multiple sinks, then

- if there is no arc (v,s), then add it with weight $\omega(v,w)$,
- otherwise, increase $\omega(v,s)$ by $\omega(v,w)$.

In both cases, delete the arc (v,w).

Note that, in the formulation of the data reduction rule, both v and w may be *connected* to an arbitrary number of sinks, that is, by an undirected path. However, we require that w *can reach* exactly one sink and that v *can reach* multiple sinks, that is, using a directed path.

Lemma 4.1. *Let (G,ω,k) be a* DAG PARTITIONING *instance and consider the graph G' with weights ω' output by Reduction Rule 4.1 applied to an arc (v,w) of G. Then (G,ω,k) is a yes-instance if and only if (G',ω',k) is.*

Proof. First, assume that (G,ω,k) is a yes-instance and that S is a minimal partitioning set of weight at most k for G. We show how to transform S into a partitioning set of equal weight for G'. We distinguish two cases: either S disconnects v from s or not.

Case 1) Assume that S disconnects v from s. Note that every subgraph of a directed acyclic graph is again a directed acyclic graph and that every vertex in a directed acyclic graph is not only connected to, but also can reach some sink. Hence, by Observation 4.2, S cannot disconnect w from s, since w can only reach s in G and would have to reach some other, that is, new sink in $G \setminus S$. It follows that S contains the arc (v,w). Now, however, $S' := (S \setminus \{v,w\}) \cup \{(v,s)\}$ is a partitioning set for G', since $G \setminus S = G' \setminus S'$. Moreover, since $\omega'(v,s) = \omega(v,s) + \omega(v,w)$, we have $\omega'(S') = \omega(S)$, where we assume that $\omega(v,s) = 0$ if there is no arc (v,s) in G.

Case 2) Assume that S does not disconnect v from s and, for the sake of a contradiction, that S is not a partitioning set for G'. Observe that S contains neither (v,w) nor (v,s), because it is a minimal partitioning set and does not disconnect v from s. Therefore, $G' \setminus S$ differs from $G \setminus S$ only in the fact that $G' \setminus S$ does not have the arc (v,w) but an arc (v,s) that was possibly not present in $G \setminus S$. Hence, since S is a partitioning set for G but not for G', two sinks are connected to each other in $G' \setminus S$ via an undirected path using the arc (v,s). Thus, one of the two sinks is s and the undirected path consists of (v,s) and a subpath p between v and some sink s'. Then, however, s is connected to s' also in $G \setminus S$ via an undirected path between s and w (S cannot disconnect s from w by Observation 4.2), the arc (v,w) and the undirected path p from v to s'. This contradicts S being a partitioning set for G. We conclude that S is a partitioning set for G'.

Now, assume that (G',ω',k) is a yes-instance and that S is a minimal partitioning set of weight at most k for G'. We show how to transform S into a partitioning set of equal weight for G. Again, we distinguish between two cases: either S disconnects v from s or not.

Case 1) Assume that S disconnects v from s. Then, $(v,s) \in S$. Now, $S' := S \cup \{(v,w)\}$ is a partitioning set for G, since $G \setminus S' = G' \setminus S$. Moreover, since $\omega'(v,s) = \omega(v,s) + \omega(v,w)$, we have $\omega(S') = \omega'(S)$, where we assume that $\omega(v,s) = 0$ if there is no arc (v,s) in G.

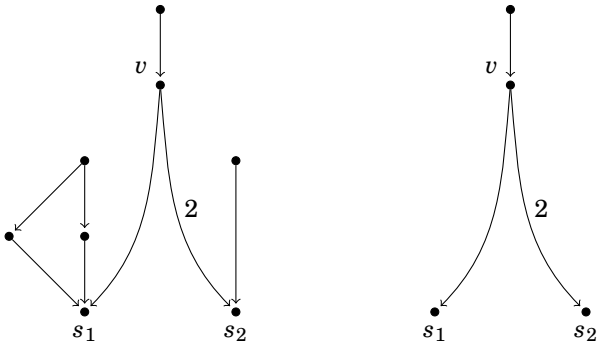


Figure 4.4: On the left hand side, a directed acyclic graph that cannot be reduced by Reduction Rule 4.1 is shown. The right hand side shows the same graph to which Reduction Rule 4.2 has been applied exhaustively. Arc labels represent arc weights, where unlabeled arcs have weight one.

Case 2) Assume that S does not disconnect v from s and, for the sake of a contradiction, assume that S is not a partitioning set for G. Then, since S is minimal, S does not contain (v,s). Now, observe that $G \setminus S$ and $G' \setminus S$ differ only in the fact that $G \setminus S$ has an additional arc (v,w) and that, possibly, (v,s) is missing. Hence, since S is a partitioning set for G' but not for G, there is an undirected path between two sinks in $G \setminus S$ through (v,w). Because S by Observation 4.2 cannot disconnect w from s, one of these sinks is s and the undirected path consists of a subpath between s and w, the arc (v,w), and a subpath p between v and a sink s'. Then, however, s and s' are also connected in $G' \setminus S$ via the arc (v,s) and the subpath p between v and s'. This contradicts S being a partitioning set for G'. □

After applying Reduction Rule 4.1 exhaustively, that is, as often as it is applicable, we apply a second data reduction rule, which is illustrated in Figure 4.4.

Reduction Rule 4.2. Let L be the maximal set of non-sink vertices that can reach exactly one sink s. Then, delete all vertices in L.

Lemma 4.2. *Let G be a graph that is exhaustively reduced with respect to Reduction Rule 4.1 and $G' := G - L$ be the graph output by Reduction Rule 4.2 when applied to G.*

Then, any partitioning set for G is a partitioning set of equal weight for G' and vice versa.

Proof. In order to prove the lemma, we first make three structural observations about the set L.

i) There is no arc (v,w) from a vertex $v \notin L$ to a vertex $w \in L$: for the sake of a contradiction, assume that such an arc exists. Then, since $v \notin L$ and v is obviously not a sink, v can reach a sink $s' \neq s$. It follows that v can reach two sinks: s, via w, and s'. This contradicts the assumption that Reduction Rule 4.1 is not applicable.

ii) There is no arc (v,w) from a vertex $v \in L$ to a vertex $w \notin L$ with $w \neq s$: for the sake of a contradiction, assume that such an arc exists. Then, since $w \notin L$ and $w \neq s$, it follows that w can reach a sink $s' \neq s$. It follows that v can reach two sinks: s', via w, and s. This contradicts $v \in L$.

iii) A minimal partitioning set S does not contain any arc between vertices in $L \cup \{s\}$: this is because, by Observation 4.2, no minimal partitioning set S can disconnect any vertex $v \in L$ from s, since otherwise v would reach another, that is, new sink in $G \setminus S$.

From (i) and (ii), it follows that the vertices in L cannot reach vertices outside of L except for the sink s and that all vertices in L are unreachable from vertices not in L.

Now, let S be a minimal partitioning set for G. Then, by (iii), S is also a partitioning set for G', since $G' \setminus S = (G \setminus S) - L$ and deleting L from $G \setminus S$ cannot create new sinks.

In the opposite direction, let S be a partitioning set for G'. Then, S is also a partitioning set for G, since $G \setminus S$ is just $G' \setminus S$ with the vertices in L and their arcs added. These vertices, however, reach only the sink s and are unreachable by vertices outside of L. □

We now show how to exhaustively apply both data reduction rules in linear time. To this end, we apply Algorithm 4.2: in lines 1–4, it computes an array L such that, for each vertex $v \in V$, we have $L[v] = \{s\}$ if v reaches exactly one sink s and $L[v] = \emptyset$ otherwise. It uses this information to apply Reduction Rule 4.1 in lines 6–10 and Reduction Rule 4.2 in lines 11 and 11.

Lemma 4.3. *Given a directed acyclic graph G with weights ω, in $O(n+m)$ time Algorithm 4.2 produces a directed acyclic graph G' with weights ω' such that G' is reduced with respect to Reduction Rules 4.1 and 4.2 and such that (G,ω,k) is a yes-instance if and only if (G',ω',k) is a yes-instance.*

Algorithm 4.2: Apply Reduction Rules 4.1 and 4.2 exhaustively.

Input: A directed acyclic graph $G = (V, A)$ with arc weights ω.
Output: The result of exhaustively applying Reduction Rules 4.1 and 4.2 to G.

1 $\mathscr{S} \leftarrow$ sinks of G
2 $L \leftarrow$ array with n entries
3 **foreach** $v \in \mathscr{S}$ **do** $L[v] \leftarrow \{v\}$ // $L[v] = \{s\} \iff v$ only reaches the sink s
4 **foreach** $v \in V \setminus \mathscr{S}$ *in reverse topological order* **do**
5 $\quad L[v] \leftarrow \bigcap_{u \in N^{\text{out}}(v)} L[u]$
// Application of Reduction Rule 4.1
6 **foreach** $v \in V$ *with* $L[v] = \emptyset$ **do**
7 $\quad$ **foreach** $w \in N^{out}(v)$ *with* $L[w] = \{s\}$ *for some* $s \in \mathscr{S}$ *and* $w \notin \mathscr{S}$ **do**
8 $\quad\quad$ **if** $(v, s) \notin A$ **then** add (v, s) with $\omega(v, s) := 0$ to A
9 $\quad\quad \omega(v, s) \leftarrow \omega(v, s) + \omega(v, w)$
10 $\quad\quad$ delete arc (v, w)
// Application of Reduction Rule 4.2
11 **foreach** $v \in V \setminus \mathscr{S}$ *such that* $L[v] = \{s\}$ *for some* $s \in \mathscr{S}$ **do** delete vertex v
12 **return** (G, ω)

Proof. We first discuss the semantics of Algorithm 4.2, then its running time. First, observe that $L[v] = \{s\}$ for some vertex v if v can reach exactly one sink s and $L[v] = \emptyset$ otherwise; this is, by definition, true for all $L[v]$ with $v \in \mathscr{S}$. For $v \in V \setminus \mathscr{S}$ it also holds, since v can reach exactly one sink s if and only if all of its out-neighbors $u \in N^{\text{out}}(v)$ can reach s and no other sinks, that is, if and only if $L[u] = \{s\}$ for all out-neighbors $u \in N^{\text{out}}(v)$ of v. Hence, the loop in lines 6–10 applies Reduction Rule 4.1 to all arcs to which Reduction Rule 4.1 is applicable. Moreover, Reduction Rule 4.1 does not change which sinks are reachable from any vertex and, hence, cannot create new arcs to which Reduction Rule 4.1 is applicable. Hence, when reaching line 11, the graph is reduced with respect to Reduction Rule 4.1 and we do not have to update the array L.

The loop in lines 11 and 11 applies Reduction Rule 4.2, which is correct by Lemma 4.2, since the graph is reduced with respect to Reduction Rule 4.1. Moreover, an application of Reduction Rule 4.2 cannot create new vertices to which Reduction Rule 4.2 is applicable or arcs to which Reduction Rule 4.1 is applicable. Hence, line 12 indeed returns a graph reduced with respect to both data reduction rules.

It remains to analyze the running time. Obviously, lines 1–3 of Algorithm 4.2 work in $O(n)$ time. To execute line 4 in $O(n + m)$ time, we iterate over the vertices

in $V \setminus \mathcal{S}$ in reverse topological order. A reverse topological order can be computed in $O(n+m)$ time [CLRS01, Section 22.4]. Due to the reverse topological order, when computing $L[v]$ for some vertex in line 4, we already know the values $L[u]$ for all $u \in N^{\text{out}}(v)$. Moreover, $L[v]$ is the intersection of sets with at most one element and, therefore, also contains at most one element. It follows that we can compute $L[v]$ in $O(|N^{\text{out}}(v)|)$ time for each vertex $v \in V \setminus \mathcal{S}$ and, therefore, in $O(n+m)$ total time for all vertices. The rest of the algorithm only iterates once over all arcs and vertices. Hence, to show that it works in $O(n+m)$ time, it remains to show how to execute lines 8 and 9 in constant time.

Herein, the main difficulty is that an adjacency list cannot answer queries of the form "$(v,s) \in A$?" in constant time. We use the following trick: assume that, when considering a vertex $v \in V$ in line 6, we have an n-element array A_v such that $A_v[s]$ holds a pointer to the value $\omega(v,s)$ if $(v,s) \in A$ and $A_v[s] = \perp$ otherwise. Then, we could in constant time check in line 8 whether $A_v[s] \neq \perp$ to find out whether $(v,s) \in A$ and, if this is the case, get a pointer to (and increment) the weight $\omega(v,s)$ in constant time in line 9. However, we cannot afford initializing all n cells of A_v, which would take $\Theta(n)$ time. Moreover, we cannot make assumptions on the value of uninitialized cells. Instead, we exploit that we access $A_v[s]$ for some s if and only if there is a vertex $w \in N^{\text{out}}(v)$ with $L[w] = \{s\}$. Hence, we can initialize all to be accessed cells of A_v in $O(|N^{\text{out}}(v)|)$ time as follows: when considering a vertex $v \in V$ in line 6, for each $w \in N^{\text{out}}(v)$, set $A_v[s] := \perp$ if $L[w] = \{s\}$. Then, for each $s \in N^{\text{out}}(v)$ with $s \in \mathcal{S}$, let $A_v[s]$ point to $\omega(v,w)$. □

Having the two Reduction Rules 4.1 and 4.2 and, with Algorithm 4.2, a way to exhaustively apply them in linear time, we will see in the experiments in Section 4.3.4 that Algorithm 4.1 is speeded up by applying it to a graph that has first been reduced using Algorithm 4.2. Moreover, in a way that Niedermeier and Rossmanith [NR00] observed to generally speed up search tree algorithms, we achieve an even stronger speedup of Algorithm 4.1 by interleaving the data reduction of Algorithm 4.2 with the branching of Algorithm 4.1.

4.3.3 Lower bounds

In Section 4.3.1, we have seen linear-time data reduction rules for DAG PARTITIONING. Moreover, the experiments in Section 4.3.4 will show that, due the data reduction executed by Algorithm 4.2, the running time of our $O(2^k \cdot (n+m))$ time Algorithm 4.1 is virtually independent from the input graph size when k is constant because Algorithm 4.2 shrinks all input instances to roughly the same size.

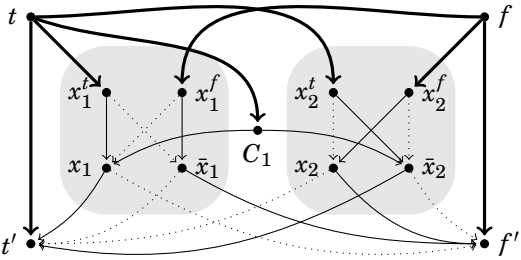


Figure 4.5: DAG PARTITIONING instance constructed from the formula consisting only of the clause $C_1 := (x_1 \vee \bar{x}_2)$. The thick arcs are heavy arcs. Dotted arcs are a partitioning set that corresponds to the satisfying assignment setting x_1 to true and x_2 to false. The variable gadgets are drawn on a gray background.

Therefore naturally arises the question whether we can provide data reduction rules that provably always shrink the size of input instances to some fixed polynomial in k, that is, whether there is a polynomial-size problem kernel for DAG PARTITIONING. Unfortunately, we negatively answer this question in this section: we prove that DAG PARTITIONING does not admit problem kernels with size polynomial in k unless the polynomial-time hierarchy collapses.

Moreover, we show lower bounds on the running time of fixed-parameter algorithms for DAG PARTITIONING parameterized by k. Towards proving these results, we exploit a polynomial-time many-to-one reduction from 3-SAT to DAG PARTITIONING given by Alamdari and Mehrabian [AM12].

3-SAT
Input: A Boolean formula φ in conjunctive normal form with at most three literals per clause.
Question: Does φ have a satisfying assignment?

We recall Alamdari and Mehrabian's [AM12] reduction from 3-SAT to DAG PARTITIONING in the following.

Construction 4.1 (Alamdari and Mehrabian [AM12])**.** Let φ be an instance of 3-SAT with the variables $x_1,\ldots,x_n$ and the clauses $C_1,\ldots,C_m$. We construct a DAG PARTITIONING instance (G,ω,k) with $k := 4n+2m$ that is a yes-instance if and only if φ is satisfiable. The weight function ω will assign only two different

weights to the arcs: a *normal arc* has weight one and a *heavy arc* has weight $k+1$ and, thus, cannot be contained in any partitioning set of weight k. The remainder of this construction is illustrated in Figure 4.5.

We start constructing the directed acyclic graph G by adding the special vertices f, f', t, and t' together with the heavy arcs (f, f') and (t, t'). The vertices f' and t' will be the only sinks in G. For each variable x_i, introduce the vertices x_i^t, x_i^f, x_i and $\bar{x}_i$ together with the heavy arcs (t, x_i^t) and (f, x_i^f) and the normal arcs (x_i^t, x_i), $(x_i^t, \bar{x}_i)$, (x_i^f, x_i), $(x_i^f, \bar{x}_i)$, (x_i, f'), $(\bar{x}_i, f')$, (x_i, t'), and $(\bar{x}_i, t')$. For each clause C_j, add a vertex C_j together with the heavy arc (t, C_j). Finally, if some clause C_j contains the literal x_i, then add the arc (C_j, x_i); if some clause C_j contains the literal $\bar{x}_i$, then add the arc $(C_j, \bar{x}_i)$.

Using observations made by Alamdari and Mehrabian [AM12], we can easily show the following lemma.

Lemma 4.4. *Given a formula φ in* 3-CNF *with n variables and m clauses, Construction 4.1 outputs a graph G with arc weights ω such that the following three statements are equivalent:*

i) The formula φ is satisfiable.

ii) There is a partitioning set for G that does not contain heavy arcs.

iii) There is a partitioning set of weight $4n+2m$ for G. It partitions G into one connected component containing the constructed vertices t and t' and the other containing f and f'.

Proof. The equivalence of (i) and (ii) has been shown by Alamdari and Mehrabian [AM12]. Trivially, (iii) implies (ii), since a partitioning set of weight $4n+2m$ cannot contain any heavy arc, which has a weight of more than $4n+2m$.

It remains to show that (ii) implies (iii). To this end, let S be a minimal partitioning set for G that does not contain heavy arcs. Since G has only the two sinks t' and f', the partitioning set S, by Observation 4.2, has to partition G into two connected components, one connected component containing the heavy arc (t, t') and the other containing (f, f'). Moreover, S has weight at most $4n+2m$: for each x_i of the n variables of φ, it deletes at most one of two arcs outgoing from each of the vertices x_i^t, x_i^f, x_i, and $\bar{x}_i$, and for each C_j of the m clauses, S deletes at most two out of the at most three arcs outgoing from the clause vertex C_j. □

We exploit Lemma 4.4 to show lower bounds on the running time of fixed-parameter algorithms and on the size of problem kernels for DAG PARTITIONING parameterized by the weight k of the sought partitioning set.

Limits of fixed-parameter algorithms

To show lower bounds on the running time for fixed-parameter algorithms for DAG Partitioning parameterized by the weight k of the sought partitioning set, we exploit the Exponential Time Hypothesis stated by Impagliazzo, Paturi, and Zane [IPZ01]. We already used the Exponential Time Hypothesis to prove lower bounds on the running time of algorithms for Job Interval Selection in Section 3.3. Recall that the hypothesis implies that n-variable 3-Sat cannot be solved in $2^{o(n)} \cdot n^{O(1)}$ time and is justified by the fact that a subexponential-time algorithm for 3-Sat would lead to subexponential-time algorithms for a large class of other NP-complete problems (also see the survey of Lokshtanov, Marx, and Saurabh [LMS11]).

Using the reduction from 3-Sat to DAG Partitioning given by Alamdari and Mehrabian [AM12] (Construction 4.1), we can easily show the following theorem, which essentially means that DAG Partitioning cannot be solved in time subexponential in k, even if we sacrifice our aim for a linear running time dependence on the input size.

Theorem 4.2. *Unless the Exponential Time Hypothesis fails,* DAG Partitioning *cannot be solved in* $2^{o(k)}n^{O(1)}$ *time.*

Proof. Construction 4.1 reduces an instance of 3-Sat consisting of a formula with n variables and m clauses to an equivalent instance (G, ω, k) of DAG Partitioning with $k = 4n + 2m$. Thus, a $2^{o(k)}n^{O(1)}$-time algorithm for DAG Partitioning would yield a $2^{o(m)}n^{O(1)}$-time algorithm for 3-Sat. This, in turn, by the so-called *Sparsification Lemma* of Impagliazzo, Paturi, and Zane [IPZ01, Corollary 2], would imply a $2^{o(n)}n^{O(1)}$ time algorithm for 3-Sat, which contradicts the Exponential Time Hypothesis. □

Although the proof of Theorem 4.2 exploits Construction 4.1, which requires arc weights, we will see in Section 4.3.3 that Theorem 4.2 even holds on graphs with unit weights.

Limits of problem kernelization

We now show that there is presumably no polynomial-size problem kernel for DAG Partitioning parameterized by the weight k of the sought partitioning set. It follows that, despite the effectiveness of data reduction we will observe in the experiments in Section 4.3.4, we presumably cannot generally shrink a DAG Partitioning instance to a size polynomial in k in polynomial time.

To show that DAG PARTITIONING does not allow for polynomial-size problem kernels, we use the cross-composition framework presented in Section 2.4.4: we show how s instances of 3-SAT can be transformed into one instance of DAG PARTITIONING that is a yes-instance if and only if one of the input formulas is satisfiable. In the reduction of s instances of 3-SAT to one instance of DAG PARTITIONING, we ensure that the parameter k of the output DAG PARTITIONING instance is bounded polynomially in the size of the largest input 3-SAT instance and logarithmically in the number s of input instances. This will yield the following theorem.

Theorem 4.3. DAG PARTITIONING *does not have a polynomial-size problem kernel with respect to the weight k of the sought partitioning set unless the polynomial-time hierarchy collapses.*

Although the proof of Theorem 4.3 is based on the following construction, which requires arc weights, we will see in Section 4.3.3 that Theorem 4.3 even holds on graphs with unit weights.

Construction 4.2. Let $\varphi_1, \ldots, \varphi_s$ be instances of 3-SAT. Since we may assume $\varphi_1, \ldots, \varphi_s$ to be from the same equivalence class of a polynomial equivalence relation, we may assume that each of the formulas $\varphi_1, \ldots, \varphi_s$ has the same number n of variables and the same number m of clauses. Moreover, we may assume that s is a power of two; otherwise, we simply add unsatisfiable formulas to the list of input instances. We now construct a DAG PARTITIONING instance (G, ω, k) with $k := 4n + 2m + 4\log s$ that is a yes-instance if and only if φ_i is satisfiable for at least one $1 \leq i \leq s$. Similarly as in Construction 4.1, the weight function ω will only assign two possible weight values: a *heavy arc* has weight $k+1$ and thus cannot be contained in any partitioning set. A *normal arc* has weight one. The remainder of the construction is illustrated in Figure 4.6.

For each instance φ_i, let G_i be the graph produced by Construction 4.1. By Lemma 4.4, G_i can be partitioned with $4n + 2m$ arc deletions if and only if φ_i is a yes-instance. We now build a gadget that, by means of $4\log s$ additional arc deletions, chooses exactly one graph G_i that has to be partitioned using the $4n + 2m$ remaining arc deletions.

To distinguish between multiple instances, we denote the special vertices f, f', t, and t' of G_i by f_i, f_i', t_i, and t_i'. For all $1 \leq i \leq s$, we add G_i to the output graph G and merge the vertices $f_1, f_2, \ldots, f_t$ into a vertex f and the vertices $f_1', f_2', \ldots, f_t'$ into a vertex f'. Furthermore, we add the vertices t, t', and t'' and the heavy arcs (t, t') and (t, t'') to G.

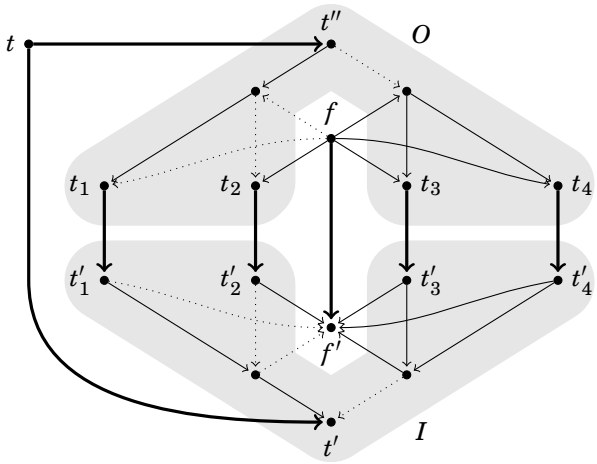


Figure 4.6: A cross-composition of four formulas $\varphi_1, \ldots, \varphi_4$ into a DAG PARTITIONING instance. Of each subgraph G_i corresponding to a formula φ_i, only the vertices t_i, t'_i, and the heavy arc joining them are shown. The introduced binary trees O and I are highlighted in gray. Deleting the $4 \log 4 = 8$ dotted arcs requires the graph G_1 to be partitioned, since its vertices t_1 and t'_1 are in a different connected component than its vertices $f_1 = f$ and $f'_1 = f'$. The graphs G_i for $i > 1$ do not have to be partitioned; they are completely contained in one connected component with f and f'.

We add a balanced binary tree O rooted in t'' that is formed by normal arcs directed from the root to its leaves $t_1, \ldots, t_s$. That is, O is an *out-tree*. For each vertex $v \neq t''$ in O, add a normal arc (f, v). Moreover, add a balanced binary tree I rooted in t' that is formed by normal arcs directed from its leaves $t'_1, \ldots, t'_s$ to its root. That is, I is an *in-tree*. For each vertex $v \neq t'$ in I, add a normal arc (v, f').

Using this construction, we can now prove Theorem 4.3.

Proof of Theorem 4.3. To prove the theorem, as described in our introduction to the cross-composition framework in Section 2.4.4, it remains to show that the instance (G, ω, k) constructed by Construction 4.2 is a yes-instance if and only if one of the input formulas φ_i is satisfiable.

First, assume that a formula φ_i is satisfiable for some $1 \leq i \leq s$. By Lemma 4.4 it follows that (G_i, k') can be partitioned by $k' := 4n + 2m$ arc deletions into two connected components P_t and P_f such that P_t contains t_i and t'_i and such that P_f contains $f_i = f$ and $f'_i = f'$. We apply these arc deletions to G and delete $4 \log s$ additional arcs from G as follows. Let L be the unique directed path in O from t'' to t_i. Analogously, let L' be the unique directed path in I from t'_i to t'. Observe that each of these directed paths has $\log s$ arcs. We partition G into the connected component $P'_t = P_t \cup \{t, t', t''\} \cup V(L) \cup V(L')$ with sink t' and into the connected component $P'_f = V(G) \setminus P'_t$ with sink f'. To this end, for each vertex $v \neq t''$ of L, we remove the incoming arc from f. For each vertex $v \neq t_i$ of L, we remove the outgoing arc that does not belong to L. Hence, exactly $2 \log s$ arcs incident to vertices of L are removed. Similarly, for each vertex $v \neq t_i$ of L', we remove the incoming arc not belonging to L'. For each vertex $v \neq t'$ of L, we remove the arc to f'. Hence, also exactly $2 \log s$ arcs incident to vertices of L' are removed. Thus, in total, at most $k = 4n + 2m + 4 \log s$ normal arcs are removed to partition G into P'_t and P'_f.

Second, let S be a minimal partitioning set for G with $\omega(S) \leq k$. Then, by Observation 4.2, $G \setminus S$ has two connected components, namely P'_t with sink t' and P'_f with sink f'. Since S cannot contain heavy arcs, t and t'' are in P'_t. Hence, t'' can reach t' in $G \setminus S$. Since every directed path from t'' to t' goes through some vertices t_i and t'_i, it follows that there is an $i \in \{1, \dots s\}$ such that t_i and t'_i are in P'_t. Since $f = f_i$ and $f' = f'_i$ are in P'_f, the partitioning set $S \cap A(G_i)$ partitions G_i into two connected components: one containing t_i and t'_i and the other containing $f = f_i$ and $f' = f'_i$. Since S does not contain heavy arcs, from Lemma 4.4 follows that φ_i is satisfiable. □

Strengthening hardness results to unit-weight graphs

The proofs of the lower bounds on the running time for solving DAG Partitioning in Theorem 4.2 and on problem kernel sizes in Theorem 4.3 heavily rely on Construction 4.1, which forbids the deletion of certain arcs by giving them high weights.

We now conclude Section 4.3 by showing that both theorems also hold on graphs with unit weights. This is implied by the following lemma, since the weights used in the instances created in the proofs of Theorems 4.2 and 4.3 are polynomial in the number of created vertices and edges.

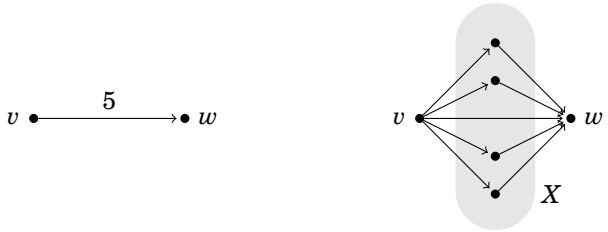


Figure 4.7: Replacing an arc of weight five on the left by the gadget of unit-weight arcs on the right. The vertex set X introduced by the proof of Lemma 4.5 is highlighted in gray.

Lemma 4.5. *Let (G,ω,k) be a* DAG PARTITIONING *instance such that, for each arc (v,w) of G, the weight $\omega(v,w)$ is bounded from above by a polynomial in $n+m$.*

In polynomial time, we can compute an equivalent instance (G',ω',k) such that ω' assigns uniform weights to all arcs.

Proof. Let (G,ω,k) be an instance of DAG PARTITIONING. We show how to obtain an instance (G',ω',k) by replacing a single arc of weight more than one by arcs of weight one such that (G,ω,k) is a yes-instance if and only if (G',ω',k) is a yes-instance. The replacement is illustrated in Figure 4.7. The claim then follows by repeating this procedure for every arc of weight more than one.

Consider an arc $a=(v,w)$ in G with $\omega(a)>1$. We obtain G' and ω' from G and ω by setting the weight $\omega'(a):=1$, adding a set X of $\omega(a)-1$ vertices to G', and inserting for each $u\in X$ a weight-one arc (v,u) and a weight-one arc (u,w). We show that (G,ω,k) is a yes-instance if and only if (G',ω',k) is a yes-instance.

First, assume that (G,ω,k) is a yes-instance and that S is a minimal partitioning set of weight k for G. We show how to obtain a partitioning set of weight k for G'. Clearly, if $a\notin S$, then S is a partitioning set of equal weight for (G',ω',k). If $a\in S$, then we get a partitioning set of equal weight for (G',ω',k) by adding the arcs between v and X to S.

Second, assume that (G',ω',k) is a yes-instance and that S is a minimal partitioning set of weight k for G'. We show how to obtain a partitioning set of weight k for G. To this end, we consider two cases: v and w are in a common or in separate connected components of $G'\setminus S$.

Case 1) If v and w are in one connected component of $G' \setminus S$, then, by minimality, S does not contain a or any arc incident to vertices in X. Hence, S is a partitioning set of equal weight for G.

Case 2) If v and w are in separate connected components of $G' \setminus S$, then $a \in S$. Moreover, each vertex in X has only one outgoing arc. Hence, by Observation 4.2, S does not contain arcs from X to w but, therefore, contains all arcs from v to X. Removing these arcs from S results in a partitioning set of equal weight for (G, ω, k). □

4.3.4 Experimental evaluation

In this section, we demonstrate to which extend our $O(2^k \cdot (n+m))$ time search tree algorithm presented in Section 4.3.1 can solve instances of DAG PARTITIONING. Herein, like previously in our experiments for COLORFUL INDEPENDENT SET WITH LISTS in Chapter 3, we demonstrate to which extent instances of DAG PARTITIONING are solvable within five minutes.

Moreover, using the optimal solutions found by our algorithm, we evaluate the quality of a heuristic presented by Leskovec, Backstrom, and Kleinberg [LBK09]. Unfortunately, the optimal partitioning sets for the data set used by Leskovec, Backstrom, and Kleinberg [LBK09] turned out to have too high weights to be found by our algorithm. Therefore, we evaluate our algorithm and the heuristic of Leskovec, Backstrom, and Kleinberg [LBK09] on preferential attachment graphs—a random graph model commonly used to model citations between articles [BA99; JNB03; Pri76].

Implementation details. We implemented Leskovec, Backstrom, and Kleinberg's [LBK09] heuristic as a special case of our search tree algorithm given in Algorithm 4.1 in Section 4.3.1. The difference is that, while our search tree algorithm branches into all possibilities of putting a vertex into a connected component with some sink, the heuristic just puts each vertex into the connected component with the sink it would be most expensive to disconnect from. Out of several strategies that Leskovec, Backstrom, and Kleinberg [LBK09] tried out, they described this strategy as working best. The heuristic is described in more detail by Suen et al. [Sue+13]. We implemented the search tree algorithm as well as the heuristic in three variants:

1. without data reduction,

2. with initially applying the data reduction algorithm presented in Algorithm 4.2, and

3. with interleaving the data reduction of Algorithm 4.2 with the branching of Algorithm 4.1.

The source code uses about 1000 lines of C++ and is freely available.[1] The experiments were run on a computer with a 3.6 GHz Intel Xeon processor and 64 GB RAM under Linux 3.2.0, where the source code has been compiled using the GNU C++ compiler in version 4.7.2 and using the highest optimization level (-O3).

Data. We applied our algorithm to the data set obtained as described by Leskovec, Backstrom, and Kleinberg [LBK09]; unfortunately, its optimal partitioning sets have too large weights to be found by our algorithm, that is, our algorithm could not find an optimal solution even after several hours. In order to prove the feasibility of solving large instances with *small* minimum partitioning sets, we generated artificial instances. In generating these instances, we stick to the clustering motivation of DAG PARTITIONING and test our algorithm on simulated citation networks: vertices in a graph represent articles and if an article v cites an article w, there is an arc (v,w). Such a citation network naturally is a directed acyclic graph, since an article usually only cites older articles. A partitioning of such a network into connected components of which each contains only one sink can be interpreted as a clustering into different topics of which we want to identify the origins or seminal works.

To simulate citation networks, we model the natural growth of citation networks, in which new articles are published over time and with high probability cite the already highly-cited articles. Indeed, Jeong, Néda, and Barabási [JNB03] empirically verified that, in this model, which is known as *preferential attachment model*, the probability of an article being cited is linear in the number of times the article has been cited in the past.

To create a preferential attachment graph, we first choose two parameters: the number c of sinks to create and the maximum outdegree d of each vertex. After creating c sinks, n new vertices are introduced one after another. After introducing a vertex, we add to it d outgoing arcs, where each of the previously introduced vertices is chosen as the target of that arc with probability proportional to its indegree.

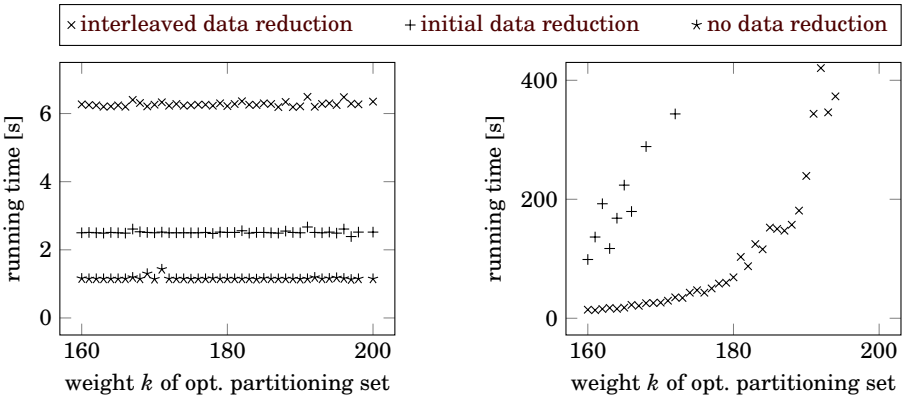
[1] http://fpt.akt.tu-berlin.de/dagpart/

We compared our algorithm to the heuristic of Leskovec, Backstrom, and Kleinberg [LBK09] on these graphs, but could solve only instances with up to 300 arcs optimally, since the weight of an optimal solution grows too quickly in the size of the generated graphs. To show the running time behavior of our algorithm on larger graphs with small solution sizes, we employ an additional approach: we generate multiple connected components, each being a preferential attachment graph with one sink, and randomly add k additional arcs between these connected components in a way so that the graph remains acyclic. Then, obviously, an optimal partitioning set cannot be larger than k. The k randomly added arcs can be viewed as noise in data that clusters well.

Experimental results. In all plots to be shown, each point has been obtained from a single run of our algorithm; the running times and memory usage are not averaged in any way. In comparison to our experiments for COLORFUL INDEPENDENT SET WITH LISTS in Chapter 3, it can be observed that the running time fluctuations of our Algorithm 4.1 for DAG PARTITIONING are more severe. This is because the search tree algorithm is highly sensitive to the structure of the input graph, which influences the number of edges deleted and, hence, the progress made in each recursive call.

Figure 4.8 compares the running time of the heuristic of Leskovec, Backstrom, and Kleinberg [LBK09] to the running time of our Algorithm 4.1 with increasing optimal partitioning set size k. One can observe that the data reduction using Algorithm 4.2 slows down the heuristic. This is not surprising, since the heuristic itself is implemented to run in linear time and, hence, instead of first shrinking the input instance by Algorithm 4.2 in linear time, one might right away solve the instance heuristically. Moreover, one can observe that, as expected, the running time of Algorithm 4.1 increases exponentially in k. We can solve instances with $k \leq 190$ optimally within five minutes. This allowed us to verify that the heuristic solved all 40 generated instances optimally, regardless of the type of data reduction applied.

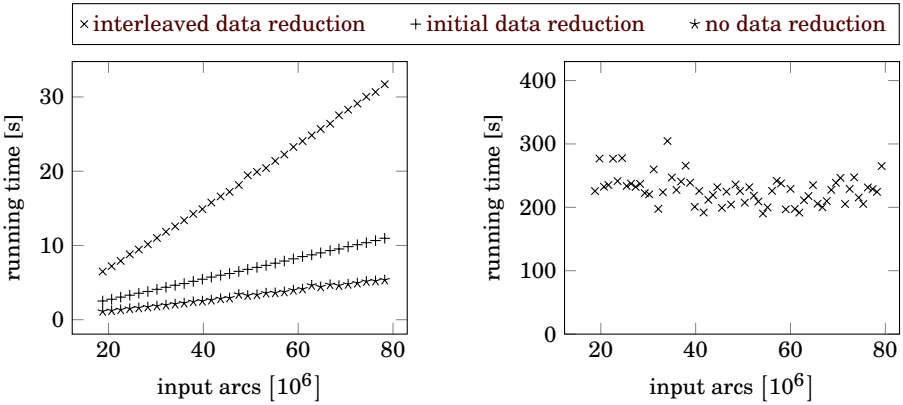
Figure 4.9 compares the running time of the heuristic of Leskovec, Backstrom, and Kleinberg [LBK09] to the running time of our Algorithm 4.1 with increasing graph size. While the heuristic shows a linear increase of running time with the graph size, such a behavior cannot be observed for the search tree algorithm. The reason for this can be seen in Figure 4.10: the data reduction applied by Algorithm 4.2 initially shrinks most input instances to about 2000 arcs in less than ten seconds. Thus, the influence of the graph size on the running time is



(a) Heuristic of Leskovec, Backstrom, and Kleinberg [LBK09]

(b) Algorithm 4.1

Figure 4.8: Comparison of the running time of Leskovec, Backstrom, and Kleinberg's [LBK09] heuristic with the running time of our search tree algorithm. Without data reduction, the search tree algorithm solved no instance in less than an hour. All graphs have 10^6 vertices and roughly $18 \cdot 10^6$ arcs and were generated by adding k random arcs between ten connected components, each being a preferential attachment graph on 10^5 vertices with outdegree twenty and one sink.



(a) Heuristic of Leskovec, Backstrom, and Kleinberg [LBK09]

(b) Algorithm 4.1

Figure 4.9: Comparison of the running time of Leskovec, Backstrom, and Kleinberg's [LBK09] heuristic with the running time of our search tree algorithm. Without interleaved data reduction, the search tree algorithm solved no instance in less than an hour. The graphs were generated by adding $k = 190$ random arcs between ten connected components, each being a preferential attachment graph with outdegree twenty and one sink.

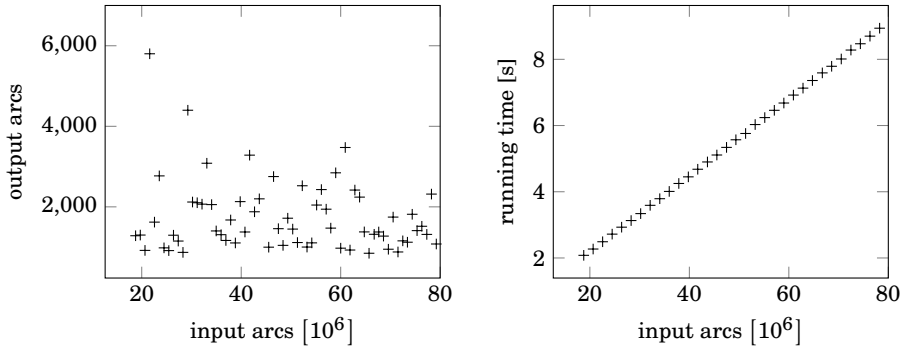
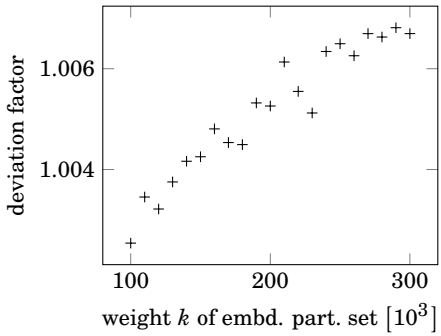


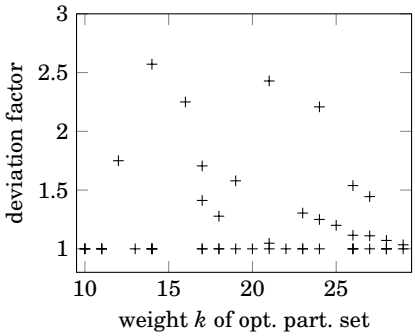
Figure 4.10: Effect and running time of initially running Algorithm 4.2 for data reduction. The graphs were generated by adding $k = 190$ random arcs between ten connected components, each being a preferential attachment graph with outdegree twenty and one sink.

too small to be observable. Again, our search tree algorithm allowed us to verify that the heuristic by Leskovec, Backstrom, and Kleinberg [LBK09] solved all 60 generated instances optimally, regardless of the type of data reduction applied.

Finally, Figure 4.11 presents instances that could not be optimally solved by Leskovec, Backstrom, and Kleinberg's [LBK09] heuristic. One can observe that, in instances with large embedded partitioning sets, the heuristic of Leskovec, Backstrom, and Kleinberg [LBK09] does not find the embedded partitioning set but an about 5‰ larger one. In all cases, the heuristic found the same partitioning sets regardless of the type of data reduction applied. Note that the comparison with the size of an embedded partitioning set only gives a lower bound on the deviation factor, since there might be even better partitioning sets in the instances than the embedded one; we were unable to compute the optimal partitioning sets in these instances. Figure 4.11 also compares the partitioning sets found by the heuristic of Leskovec, Backstrom, and Kleinberg [LBK09] with optimal solutions on small preferential attachment graphs without embedded partitioning sets. We see that Leskovec, Backstrom, and Kleinberg's [LBK09] heuristic can be more than by a factor of two off the optimal partitioning set. Data reduction had no effect on the quality of the found partitioning sets.



(a) Comparison with large embedded partitioning sets in graphs with 10^6 vertices and roughly $18 \cdot 10^6$ arcs generated by adding k random arcs between ten connected components, each being a preferential attachment graph on 10^5 vertices with outdegree two and one sink.



(b) Comparison with optimal partitioning sets in preferential attachment graphs with a number of arcs varying between 150 and 350, with two sinks and outdegree three.

Figure 4.11: Comparison of partitioning sets found by Leskovec, Backstrom, and Kleinberg's [LBK09] heuristic with large embedded partitioning sets and with optimal partitioning sets in small preferential attachment graphs without embedded solutions.

Summary. We have seen that solving large instances with partitioning sets of small weight is realistic using our algorithm. In particular, instances with more than 10^7 arcs and $k \leq 190$ could be solved in less than five minutes. A crucial ingredient in this success is the data reduction executed by Algorithm 4.2; without its help, we could not solve any of our instances in less than one hour.

However, we also observed that our algorithm works best on those instances that can already be solved mostly optimally by Leskovec, Backstrom, and Kleinberg's [LBK09] heuristic and that the data reduction executed by Algorithm 4.2 slows down the heuristic.

Having seen that the heuristic by Leskovec, Backstrom, and Kleinberg [LBK09] can be more than a factor of two off the optimum on random preferential attachment graphs diminishes the hope that, in spite of the non-approximability results of DAG PARTITIONING by Alamdari and Mehrabian [AM12], the heuristic of Leskovec, Backstrom, and Kleinberg [LBK09] might find good approximations on naturally occurring instances. As we see, we do not have to construct adversarial instances to make the heuristic find far from optimal solutions.

4.4 Linear-time partitioning tree-like graphs

In Section 4.3, we have seen that DAG PARTITIONING is linear-time solvable when the weight k of the sought partitioning set is constant. Alamdari and Mehrabian [AM12] asked whether DAG PARTITIONING is fixed-parameter tractable with respect to the parameter treewidth, which is, as introduced in Section 2.5, a measure of the "tree-likeness" of a graph.

Alamdari and Mehrabian [AM12] showed a dynamic programming based algorithm to solve DAG PARTITIONING in $2^{O(t^2)} \cdot n$ time when a path decomposition of width t of the input graph is given as input. One can indeed improve their algorithm by showing the following theorem:

Theorem 4.4 ([Bev+13])**.** *Given a width-t tree decomposition of the input graph,* DAG PARTITIONING *can be solved in $2^{O(t^2)} \cdot n$ time.*

The theorem can be proven using a dynamic programming based algorithm. The algorithm and its correctness proof are very technical, its practical applicability is rather limited already for small values of t (nevertheless, recently Abseher et al. [Abs+14] have made efforts of implementing it), and the involved techniques do not give further insights into DAG PARTITIONING. For this reason, we skip the presentation of the algorithm. Instead, we present a simple and easy to

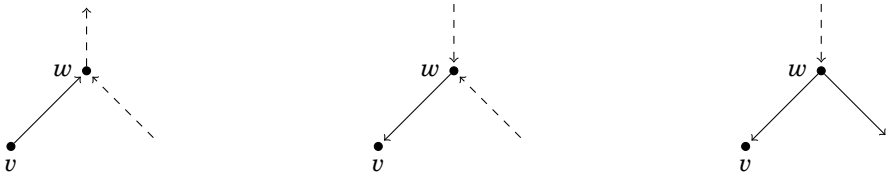


Figure 4.12: Illustration of the cases (i), (ii), and (iii) of Reduction Rule 4.3 from left to right. Dashed lines represent arc types that might be present. Solid arcs have to be present.

implement linear-time algorithm that solves DAG PARTITIONING on unweighted directed acyclic graphs whose underlying undirected graphs are trees.

Theorem 4.5. DAG PARTITIONING *is solvable in linear time by a simple data reduction rule on unweighted directed acyclic graphs whose underlying undirected graphs are trees.*

To prove Theorem 4.5, we apply data reduction rules to the leaves of the underlying undirected graph until the graph becomes empty. The three cases of the data reduction rule are illustrated in Figure 4.12.

Reduction Rule 4.3. Let (G, ω, k) be a DAG PARTITIONING instance with unit weights, where G is a directed acyclic graph whose underlying undirected graph is a tree. Exhaustively apply the following rules to G.

i) If there is a leaf v with outdegree one, then delete v.

ii) If there is a leaf v with indegree one such that v is the only out-neighbor of its parent, then delete v.

iii) If (i) is not applicable to any leaf and there is a leaf v with indegree one such that v is not the only out-neighbor of its parent, then delete v and decrement k by one.

Lemma 4.6. *For the instance (G', ω', k') obtained from an instance (G, ω, k) by applying Reduction Rule 4.3 to a vertex v of a directed acyclic graph G whose underlying undirected graph is a tree, it holds that (G, ω, k) is a yes-instance if and only if (G', ω', k') is a yes-instance.*

Proof. We show the lemma for each of the cases of Reduction Rule 4.3 independently.

i) If (G,ω,k) is a yes-instance, then so is (G',ω',k'), since any minimal partitioning set of size k for G is a partitioning set of size $k' = k$ for G': by Observation 4.2, it cannot contain the only arc outgoing from v. Now, let (G',ω',k') be a yes-instance and S be a partitioning set of size $k = k'$ for G'. Let w be the out-neighbor of v in G. The vertex w is connected to exactly one sink in $G' \setminus S$. Since v is not a sink in G, it follows that w is connected to exactly one sink in $G \setminus S$. Finally, since w is the only out-neighbor of v in $G \setminus S$, the vertex v is connected to the same sink as w in $G \setminus S$. It follows that S is a partitioning set of size $k = k'$ for G and, therefore, (G,ω,k) is a yes-instance.

ii) Observe that the in-neighbor w of v has outdegree one. Hence, by Observation 4.2, no minimal partitioning set contains the only arc outgoing from w. Thus, any minimal partitioning set for G is a partitioning set for $G' = G - \{v\}$. Moreover, any partitioning set S for G' is a partitioning set also for G: $G \setminus S$ and $G' \setminus S$ differ only in the out-neighbors of w. Since w is a sink in $G' \setminus S$, all vertices that are connected to w in $G' \setminus S$ are only connected to the sink v in $G \setminus S$. Thus, S is a partitioning set for G.

iii) It is clear that, if (G',ω',k') is a yes-instance, then so is (G,ω,k): if S is a partitioning set for G' of size $k' = k-1$, then $S \cup \{(w,v)\}$ is a partitioning set of size $k'+1 = k$ for G, where w is the in-neighbor of v.

We now show that, if (G,ω,k) is a yes-instance, then so is $(G',\omega',k-1)$. To this end, let S with $|S| \leq k$ be a minimal partitioning set for G. If $(w,v) \in S$, where w is the in-neighbor of v, then $S \setminus \{(w,v)\}$ is a partitioning set of size $k-1$ for G' and we have shown that $(G',\omega',k-1)$ is a yes-instance.

It remains to show that if $(w,v) \notin S$, then we can transform S into a partitioning set S' of equal size that contains (w,v). The remainder of this proof is illustrated in Figure 4.13. Observe that every tree has at least two leaves and, since (i) is not applicable to any leaf, all leaves in G are sinks. Hence, $S \neq \emptyset$. Let $a \in S$ be an arc outgoing from w and, if S contains no arcs outgoing from w, let $a \in S$ be an arc with least distance to v. We finish the proof by showing that $S' := (S \setminus \{a\}) \cup \{(w,v)\}$ is a partitioning set for G.

Assume, for the sake of a contradiction, that S but not S' is a partitioning set for G. Then, $G \setminus S'$ contains two sinks s_1, s_2 that are connected by an undirected path p' containing the arc a. If s_1 and s_2 are sinks also in $G \setminus S$, we obtain a contradiction as follows: we show that s_1 or s_2 are connected to the sink v by an undirected path in $G \setminus S$. Since the undirected path p' between s_1 and s_2 in G contains a, it also contains the vertex v' of a that has least distance to v.

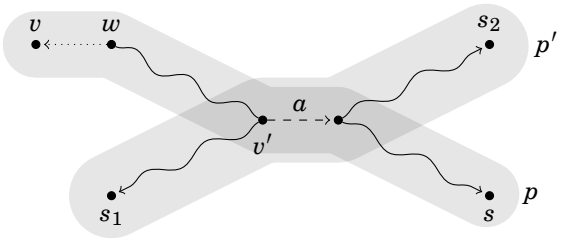


Figure 4.13: Illustration for the proof of the correctness of Reduction Rule 4.3(iii). Shown are the undirected path p between v and s and the undirected path p' between s_1 and s_2. The dashed arc a is contained in S, while the dotted arc (w,v) is contained in S'. Wavy lines represent undirected paths of arbitrary length.

Moreover, since each leaf in G is a sink, there is an undirected path p between v and some sink s in G that goes through a. By choice of a, S contains no arcs of p between v and v'; moreover, since p' exists in $G \setminus S'$, a is the only arc of S on p', and, therefore, S contains no arc on the subpath of p' between v' and one of s_1 or s_2. It follows that v is connected to s_1 or s_2 in $G \setminus S$, which contradicts S being a partitioning set for G.

We have seen that, if both sinks s_1 and s_2 of $G \setminus S'$ are also sinks in $G \setminus S$, we obtain a contradiction to S being a partitioning set for G. Hence, without loss of generality, assume that s_1 is not a sink in $G \setminus S$. Since s_1 is a sink in $G \setminus S'$, the construction of S' from S implies $s_1 = w$. That is, S contains all arcs outgoing from w except for (w,v) and, by choice of a, a is one such arc. We now have a contradiction: by construction of S' from S, the graph $G \setminus S'$ contains the arc a outgoing from w and, hence, $w = s_1$ is not a sink in $G \setminus S'$. □

Having shown the correctness of Reduction Rule 4.3, we show how to exhaustively apply it in linear time to prove Theorem 4.5.

Proof of Theorem 4.5. First, observe that Reduction Rule 4.3 indeed solves DAG PARTITIONING on unweighted directed acyclic graphs whose underlying undirected graphs are trees: as long as the tree has leaves, one of the cases of Reduction Rule 4.3 applies. It remains to show how to exhaustively apply Reduction Rule 4.3 in linear time.

For a given leaf $v \in V$, it is checkable in O(1) time whether it has outdegree or indegree one and whether it is the only out-neighbor of its parent (just check

whether the outdegree of the parent is one or two). Hence, applicability of Reduction Rule 4.3(i) and (ii) can be checked in O(1) time for each leaf. We cannot check the applicability of (iii) in O(1) time, since we would have to check whether (i) applies to any vertex. However, when any case of Reduction Rule 4.3 is applicable, Reduction Rule 4.3 deletes only a constant-degree vertex v, which works in constant time.

We proceed as follows: we build three lists L_1, L_2, and L_3 of leaves. To L_1, we add all leaves to which Reduction Rule 4.3(i) is applicable. To L_2, we add all leaves to which (ii) is applicable. All remaining leaves we add to L_3. We will maintain the invariant that L_1 contains all leaves to which (i) applies, that L_2 contains all leaves to which (ii) applies, and that all leaves are contained in at least one of L_1, L_2, or L_3.

We now exhaustively apply the following steps, where a step with higher number is only executed when no step with a lower number applies.

1. If the first vertex in L_1, L_2 or L_3 has already been deleted from G, then delete it from L_1, L_2, or L_3, respectively. Whether a vertex has already been deleted can be recorded and looked up in constant time in a size-n array.

2. If L_1 is nonempty, then delete the first leaf v in L_1 from G (because Reduction Rule 4.3(i) is applicable to v). Moreover, if the neighbor w of v has become a leaf, add w to L_1 if Reduction Rule 4.3(i) applies to w, add it to L_2 if (ii) applies to w, and add it to L_3 otherwise.

3. If L_2 is nonempty, then delete the first leaf v in L_2 from G (because Reduction Rule 4.3(ii) is applicable to v). Moreover, if the neighbor w of v has become a leaf, add w to L_1 if Reduction Rule 4.3(i) applies to w, add it to L_2 if (ii) applies to w, and add it to L_3 otherwise.

4. If L_3 is nonempty, then delete the first leaf v in L_3 from G and decrement k by one (because Reduction Rule 4.3(iii) is applicable to v). The in-neighbor w of v cannot have become a leaf, but if w has only one out-neighbor v' after deletion of v and v' is a leaf, (ii) has become applicable to v'. If so, add v' to L_2.

Initially, the lists L_1, L_2, and L_3 have at most n entries. Since at most n vertices are deleted from G, in total at most n new entries are added to the lists. Since each entry in L_1, L_2, and L_3 is processed in constant time, it follows that the overall procedure runs in $O(n)$ time. □

4.5 Stronger NP-hardness results

In Sections 4.3 and 4.4, we have seen that DAG PARTITIONING is solvable in linear time when fixing the weight of the sought partitioning set or the treewidth of the input graph. Naturally arises the question whether fixed-parameter algorithms can be obtained for other parameters that are likely to be small in applications. One such parameter is, for example, the maximum vertex outdegree in the graph: a citation network of journal articles will, for example, have a small outdegree, since journal articles seldom contain more than fifty references.

However, Alamdari and Mehrabian [AM12] already showed that DAG PARTITIONING remains NP-hard even if the input graph has only two sinks and maximum outdegree three. We complement this negative result by showing that the problem remains NP-hard even if the diameter or the maximum vertex degree of the input graph are constant. In conclusion, parameters like the number of sinks, the graph diameter or maximum degree cannot lead to fixed-parameter algorithms unless P = NP.

Theorem 4.6. DAG PARTITIONING *is solvable in linear time when the underlying undirected graph has diameter one, but NP-complete if the underlying undirected graph has diameter two.*

Proof. If the underlying undirected graph has diameter one, it is a clique. Thus, the input graph is an acyclic *tournament*. As such, it already contains exactly one source and one sink. Hence, we just verify in linear time whether the input graph is an acyclic tournament and answer "yes" or "no" accordingly.

For the case where the underlying undirected graph has diameter two, we show NP-hardness by means of a polynomial-time many-one reduction from DAG PARTITIONING, which is NP-hard even when all arcs have weight one (Lemmas 4.4 and 4.5). Therefore, we agree on all arcs in this proof having weight one.

Given an instance (G, ω, k) of DAG PARTITIONING, we add a gadget to G to obtain in polynomial time an instance (G', ω', k') such that the underlying undirected graph of G' has diameter two and such that (G, ω, k) is a yes-instance if and only if (G', ω', k') is a yes-instance. We obtain G' from G by adding an acyclic tournament consisting of $k+n+2$ vertices and by adding outgoing arcs from the source s of the tournament to all vertices in $V(G)$ of G'. We set $k' := k+n$. Since every vertex in G' is adjacent to s, the underlying undirected graph of G' has diameter two.

By choosing the added acyclic tournament to have $k+n+2$ vertices, we ensure that a partitioning set of the sought size $k' = k+n$ for G' cannot disconnect the tournament's source from its sink. Now, since every vertex in G is connected to at

least one sink in G, we have to separate the source s of the tournament from all vertices in $V(G)$ of G', since, otherwise, s would be connected to at least two sinks in G': to one sink of $V(G)$ and to the sink of the tournament. Since a partitioning set S of size k' cannot separate the source s of the tournament from the sink of the tournament, S has to remove from G' the n arcs connecting s to the vertices in $V(G)$. Thereafter, the tournament is a connected component with exactly one sink and it remains to find a partitioning set of size $k' - n = k$ in the remaining graph. This, however, is precisely the original graph G. Thus, (G', ω', k') is a yes-instance if and only if (G, ω, k) is a yes-instance. □

In the following, we show that, also on graphs with maximum degree three, DAG PARTITIONING remains NP-hard.

Theorem 4.7. DAG PARTITIONING *is solvable in linear time on graphs of maximum degree two, but NP-complete on graphs of maximum degree three.*

Proof. The underlying undirected graph of any directed acyclic graph of maximum degree two consists of cycles or paths and, thus, has treewidth at most two. We have seen in Section 4.4 that DAG PARTITIONING is linear-time solvable when the treewidth of the input graph is bounded by a constant.

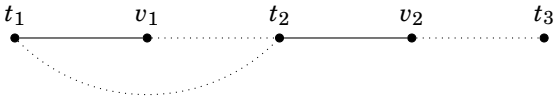
We now prove NP-hardness on graphs of maximum degree three. To this end, we adapt a polynomial-time many-one reduction from MULTIWAY CUT to DAG PARTITIONING presented by Leskovec, Backstrom, and Kleinberg [LBK09]. In their reduction, we replace vertices of degree greater than three by equivalent structures of degree at most three.

MULTIWAY CUT

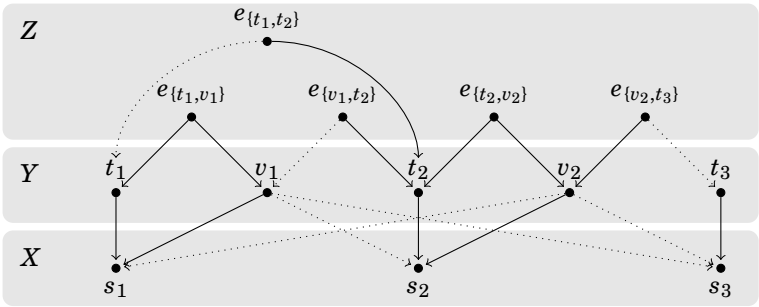
Input: An undirected graph $G = (V, E)$, a weight function $\omega : E \to \mathbb{N}$, a set $T \subseteq V$ of *terminals*, and an integer k.

Question: Is there a *multiway cut* $S \subseteq E$ with $\sum_{e \in S} \omega(e) \leq k$ such that the removal of S from G disconnects each terminal from all the others?

We first recall the reduction from MULTIWAY CUT to DAG PARTITIONING. From a MULTIWAY CUT instance $I_1 := (G_1, \omega_1, T, k_1)$, we construct in polynomial time a DAG PARTITIONING instance $I_2 := (G_2, \omega_2, k_2)$ such that I_1 is a yes-instance if and only if I_2 is. From I_2, we then obtain an instance I_3 with maximum degree three. Since MULTIWAY CUT remains NP-hard even for three terminals and unit weights [Dah+94], we assume $|T| = 3$ and, similarly as in the proof of Theorem 4.6, we agree on all arcs in this proof having weight one. We now construct the DAG PARTITIONING instance $I_2 = (G_2, \omega_2, k_2)$ from $I_1 = (G_1, \omega_1, T, k_1)$ as follows. The construction is illustrated in Figure 4.14.



(a) An instance I_1 of MULTIWAY CUT, where the dotted edges are a multiway cut of size $k_1 = 3$.



(b) The corresponding instance I_2 of DAG PARTITIONING and a corresponding partitioning set of size $k_2 = k_1 + 2(n-3) = 7$ symbolized by the dotted arcs, where n is the number of vertices in the MULTIWAY CUT instance.

Figure 4.14: Reduction from a MULTIWAY CUT instance with the terminals t_1, t_2, and t_3 to DAG PARTITIONING. The constructed vertex sets X, Y, and Z are highlighted using a gray background.

1. Add three vertices s_1, s_2, s_3 to G_2, forming the vertex set X,
2. add each vertex of G_1 to G_2, forming the vertex set Y,
3. for each edge $\{u, v\}$ of G_1, add a vertex $e_{\{u,v\}}$ to G_2, forming the vertex set Z,
4. for each terminal $t_i \in T$, add the arc (t_i, s_i) to G_2,
5. for each vertex $v \in Y \setminus T$, add the arcs (v, s_i) for $i = 1,2,3$ to G_2, and
6. for each edge $\{u, v\}$ of G_1, add the arcs $(e_{\{u,v\}}, u)$ and $(e_{\{u,v\}}, v)$ to G_2.

Set $k_2 := k_1 + 2(n-3)$, where n is the number of vertices of G_1. We claim that I_1 is a yes-instance if and only if I_2 is a yes-instance.

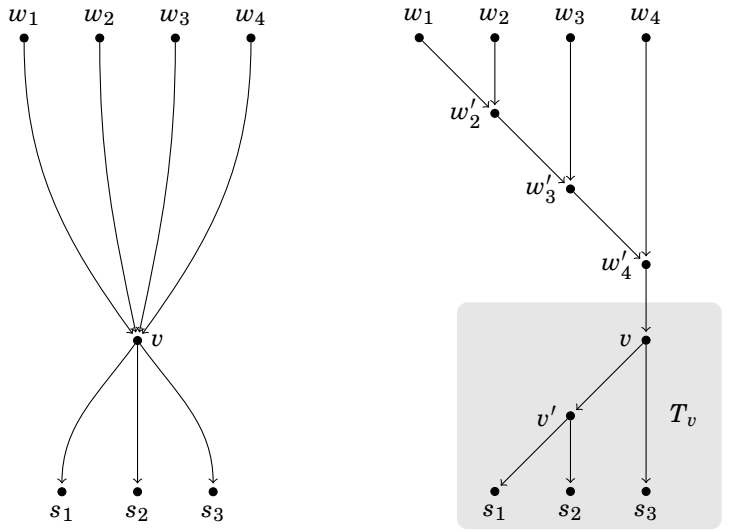
First, suppose that there is a multiway cut S of size at most k_1 for G_1. Then, we obtain a partitioning set of size at most k_2 for G_2 as follows: if a vertex v belongs to the same connected component of $G_1 \setminus S$ as the terminal t_i, then remove every arc (v, s_j) with $j \neq i$ from G_2. Furthermore, for each edge $\{u, v\} \in S$, remove *either* the arc $(e_{\{u,v\}}, u)$ or the arc $(e_{\{u,v\}}, v)$ from G_2. One can easily check that we end up with a valid partitioning set of size $k_2 = k + 2(n-3)$ for G_2: we delete at most k arcs from Z to Y and, for each of the $n-3$ vertices in $Y \setminus T$, we delete two arcs from Y to X. There are no arcs from X to Z.

Conversely, suppose that we are given a minimal partitioning set S of size at most k_2 for G_2. Note that it has to remove at least two of the three outgoing arcs of each vertex $v \in Y \setminus T$ but cannot remove all three of them: contrary to Observation 4.2, this would create a new sink. Thus, S deletes $2(n-3)$ arcs from Y to X and the remaining $k_2 - 2(n-3) = k_1$ arcs from Z to Y. Therefore, we can define the following multiway cut of size k_1 for G_1: remove an edge $\{u, v\}$ from G_1 if and only if one of the arcs $(e_{\{u,v\}}, u)$ and $(e_{\{u,v\}}, v)$ is removed from G_2 by S. Again, one can easily check that we end up with a valid multiway cut.

It remains to modify the instance $I_2 = (G_2, \omega_2, k_2)$ to get an instance $I_3 := (G_3, \omega_3, k_3)$ of maximum degree three. To this end, first we show how to reduce the outdegree of each vertex of G_2 to at most two. Thereafter, we show how to reduce the indegree of each vertex of G_2 to at most one. To this end, we will introduce gadget vertices, each having degree at most three. The construction is illustrated in Figure 4.15.

Note that all vertices of G_2 with outdegree larger than two are in Y. In order to decrease the degree of these vertices, we obtain a graph G_3' from G_2 by carrying out the following modifications (see Figure 4.15) to G_2: for each vertex $v \in Y$, with $N^{\text{out}}(v) = \{s_1, s_2, s_3\}$, remove (v, s_1) and (v, s_2), add a new vertex v', and insert the three arcs (v, v'), (v', s_1), and (v', s_2).

We show that (G_3', ω_3', k_2) is a yes-instance if and only if (G_2, ω_2, k_2) is. To this end, for $v \in Y$, let T_v be the induced subgraph $G_3[\{v, v', s_1, s_2, s_3\}]$. In G_2, a minimal partitioning set removes exactly two of the outgoing arcs of v, since s_1, s_2, and s_3 are sinks. It is enough to show that a minimal partitioning set S removes exactly two arcs in T_v from G_3 in such a way that there remains exactly one directed path from v to exactly one of s_1, s_2, or s_3. This remaining directed path then one-to-one corresponds to the arc that a partitioning set would leave in G_2 between v and s_1, s_2, or s_3. Since s_1, s_2, and s_3 are sinks, S indeed has to remove at least two arcs from T_v: otherwise, two sinks will belong to the same connected component. However, due to Observation 4.2, S cannot remove more than two



(a) The neighborhood of a vertex v with degree more than three.

(b) The neighborhood of v after the transformation.

Figure 4.15: Reduction of the degree of a vertex v to three. The tree structure T_v constructed in the proof is highlighted using a gray background.

arcs from T_v. Moreover, again exploiting Observation 4.2, the two arcs removed by S leave a single directed path from v to exactly one of s_1, s_2, or s_3.

We have seen that (G'_3, ω'_3, k_2) is a yes-instance for DAG PARTITIONING if and only if (G_2, ω_2, k_2) is and that all vertices of G'_3 have outdegree at most two. To shrink the overall maximum degree to three, it remains to reduce the indegree of all vertices to at most one. Note that all vertices in $V(G'_3) \setminus V(G_2)$, that is, the vertices introduced in the previous step, already have degree three. We obtain the graph G_3 of maximum degree three from G'_3 as follows. For each vertex v of G'_3 with $d := |N^{\text{in}}(v)| = |\{w_1, \ldots, w_d\}| \geq 2$, do the following (see Figure 4.15): remove all arcs incoming to v. Instead, for $j \in \{2, \ldots, d\}$, add a vertex w'_j together with the arc (w_j, w'_j). Moreover, add the arcs (w_1, w'_2), (w'_d, v), and (w'_j, w'_{j+1}) for each $j \in \{2, \ldots, d-1\}$. Now, every vertex of $V(G'_3)$ in G_3 has indegree at most one and outdegree at most two. Every vertex in G_3 that is not in $V(G'_3)$, that is, every

newly introduced vertex, has indegree two and outdegree one. It follows that all vertices in G_3 have degree at most three.

It remains to show that (G_3, ω_3, k_2) is a yes-instance if and only if (G'_3, ω'_3, k_2) is. It then follows that (G_3, ω_3, k_2) is a yes-instance if and only if (G_2, ω_2, k_2) is. To this end, note that, by Observation 4.2, among the arcs introduced for a vertex v with the in-neighbors $w_1, \ldots, w_d$, only the arcs (w_j, w'_j) and (w_1, w'_2) can be removed by a minimal partitioning set, since the vertices w'_j have outdegree one. From this, we get a one-to-one correspondence between deleting the arc (w_j, v) in the graph G'_3 and deleting the arc (w_j, w'_j) (or the arc (w_j, w'_2) for $j = 1$) in the graph G_3. □

4.6 Conclusion

We have presented a fixed-parameter linear-time algorithm for DAG Partitioning parameterized by the weight k of the sought partitioning set. We demonstrated the feasibility of applying the algorithm to large input instances with optimal partitioning sets of small weight. However, we were unable to solve the instances from the clustering task of Leskovec, Backstrom, and Kleinberg [LBK09], since the weights of optimal partitioning sets in their setting are too large. We found out that the heuristic presented by Leskovec, Backstrom, and Kleinberg [LBK09] finds mostly optimal partitioning sets on the instances that our algorithm works on best. However, we have also seen that one does not have to specially craft adversarial instances to make the heuristic perform badly.

Since the main reason for not being able to solve DAG Partitioning on instances arising in the clustering task of Leskovec, Backstrom, and Kleinberg [LBK09] is the size of the parameter k, it would be interesting to find smaller parameters with respect to which DAG Partitioning is fixed-parameter tractable. A canonical parameter, which is at most k, would be the number of vertices for which incident arcs are allowed to be modified.

Note that, in finding fixed-parameter algorithms for smaller parameters, especially fixed-parameter linear-time algorithms are of interest, since, as expressed by Suen et al. [Sue+13], the data set is too large to be processed by algorithms with a quadratic running time.

5 Hitting Set

In Chapter 4, we have seen that linear-time data reduction for DAG Partitioning had a vast impact on the solvability of our instances. In this chapter, we show a linear-time computable problem kernel for d-Hitting Set—the NP-hard problem of selecting at most k vertices of a hypergraph so that each hyperedge, all of which have cardinality at most d, contains at least one selected vertex. The applications of d-Hitting Set are, for example, fault diagnosis, automatic program verification, and the noise-minimizing assignment of frequencies to radio transmitters [KW87; OC03; Rei87; SMNW14].

We show a linear-time computable problem kernel comprising $O(k^d)$ hyperedges and vertices for d-Hitting Set. Our kernelization algorithm is based on speeding up the well-known approach of finding and shrinking *sunflowers* in hypergraphs [FG06; FSV13; Kra12], which yields problem kernels with structural properties that we will condense into the concept of *expressive kernelization*.

We conduct experiments to show that our kernelization algorithm can kernelize instances with more than 10^7 hyperedges in less than five minutes.

Finally, we show that the number of vertices in the problem kernel can be further reduced to $O(k^{d-1})$ with additional $O(k^{1.5d})$ processing time by nontrivially combining the sunflower technique with d-Hitting Set problem kernels due to Abu-Khzam [Abu10] and Moser [Mos10].

5.1 Introduction

Many problems, like the examples given below, can be modeled as the NP-hard d-Hitting Set problem:

d-Hitting Set
Input: A hypergraph $H = (V, E)$ with hyperedges whose cardinality is bounded from above by a constant d, and a natural number k.
Question: Is there a *hitting set* $S \subseteq V$ with $|S| \leq k$ and $\forall e \in E : e \cap S \neq \emptyset$?

Problems that can be modeled as d-Hitting Set arise, among others, in the following fields.

Construction of Golomb rulers. A *Golomb ruler* of length n is a subset of *marks* $R \subseteq [n]$ such that no pair of marks in R has the same distance as another pair. The task of finding shortest Golomb rulers with a fixed number of marks or Golomb rulers of fixed length with a maximum number of marks arises, among others, in radio frequency allocation [Dra09]. Sorge et al. [SMNW14] showed how to construct Golomb rulers using 4-Hitting Set. That is, $d = 4$.

Fault diagnosis. The task is to detect faulty components of a malfunctioning system. To this end, those sets of components are mapped to hyperedges of a hypergraph that are known to contain at least one broken component [KW87; Rei87]. By the principle of Occam's razor, a small hitting set is then a likely explanation of the malfunction. In this application, d is the maximum number of components that a wrong observation depends on.

Program verification. O'Callahan and Choi [OC03] used d-Hitting Set in order to automatically detect bugs in parallel Java programs while aiming for a small slowdown of the program monitored at execution time. In their experiments, $d \leq 10$ was sufficient to debug complex software suites. In most cases, even $d \leq 5$ sufficed. Remarkably, in this application, one is interested in the question whether a hypergraph allows for a hitting set of size at most $k = d$, that is, both k and d are small.

The described problems have in common that a large number of "conflicts" (the possibly $O(n^d)$ hyperedges in a d-Hitting Set instance) is caused by a small number of elements (the hitting set S), whose removal or repair could fix a broken system or establish a useful property.

We show how to compute a problem kernel with $O(k^d)$ hyperedges for d-Hitting Set in linear time. We experimentally evaluate our kernelization algorithm on 4-Hitting Set instances arising in the construction of Golomb rulers with a maximum number of marks and see that instances with more than 10^7 hyperedges are kernelizable in less than five minutes.

5.1.1 Known results

Hitting Set is W[2]-complete with respect to the parameter k when the cardinality of the hyperedges is unbounded [FG06, Theorem 7.14]. Hence, unless FPT = W[2], it has no problem kernel. Dell and van Melkebeek [DM14] showed that the existence of a problem kernel with $O(k^{d-\varepsilon})$ hyperedges for any $\varepsilon > 0$ for

d-Hitting Set implies a collapse of the polynomial-time hierarchy. Therefore, d-Hitting Set is assumed not to admit problem kernels with $O(k^{d-\varepsilon})$ hyperedges. For the same reason, d-Hitting Set presumably has no polynomial-size problem kernels if d is *not* constant.

Various problem kernels for d-Hitting Set have been developed [Abu10; Dam06; FG06; FK14; Kra12; Mos10; NR03; NRT04]. Niedermeier and Rossmanith [NR03] showed a problem kernel for 3-Hitting Set of size $O(k^3)$. They implicitly claimed that a polynomial-size problem kernel for d-Hitting Set is computable in linear time, without giving a proof for the running time. Nishimura, Ragde, and Thilikos [NRT04] claimed that a problem kernel with $O(k^{d-1})$ vertices is computable in $O(k(n+m)+k^d)$ time, which, however, does not always yield correct problem kernels [Abu10]. Damaschke [Dam06] focused on developing small problem kernels for d-Hitting Set and other problems with the focus on preserving *all* minimal solutions of size at most k (so-called *full kernels*). Fafianie and Kratsch [FK14] presented a so-called *streaming kernelization* for d-Hitting Set, which reads every hyperedge in the input hypergraph at most once and has logarithmic memory usage for fixed k. Abu-Khzam [Abu10] showed a problem kernel with $O(k^{d-1})$ vertices for d-Hitting Set, thus proving the previously claimed result of Nishimura, Ragde, and Thilikos [NRT04] on the number of vertices in the problem kernel. Moser [Mos10, Section 7.3] built upon the work of Abu-Khzam [Abu10] to show a problem kernel for d-Hitting Set that also comprises $O(k^{d-1})$ vertices but, in contrast to the problem kernel of Abu-Khzam [Abu10], yields a subgraph of the input hypergraph. The problem kernels of Abu-Khzam [Abu10] and Moser [Mos10] comprise $\Omega(k^{2d-2})$ hyperedges in the worst case.[1]

Several exponential-time algorithms for Hitting Set exist and aim to decrease the exponential dependence of the running time on the number of input vertices [SC10], on the number of input hyperedges [Fer06], and on the size of the sought hitting set [Fer10]. Also exponential-time approximation stepped into the field of interest [BF12], since, in polynomial time, d-Hitting Set appears to be hard to approximate within a factor of better than d [KR08].

[1] Although not directly analyzed in the works of Abu-Khzam [Abu10] and Moser [Mos10], this can be seen as follows: the kernel comprises vertices of a set W of "weakly related" hyperedges and an independent set I. In the worst case, $|W| = k^{d-1}$, $|I| = dk^{d-1}$, and each hyperedge in W has d subsets of size $d-1$. Each such subset can constitute a hyperedge with each vertex in I and the kernel has $\Omega(k^{2d-2})$ hyperedges.

5.1.2 Our results

We show a linear-time computable problem kernel for d-Hitting Set with $O(k^d)$ hyperedges and vertices. Thereby, we prove the previously claimed result by Niedermeier and Rossmanith [NR03] and complement recent results in improving the efficiency of kernelization algorithms [Bev+12; FK14; Hag12; Kam13; PDS09].

Our problem kernel has useful structural properties that ensure the interpretability of the problem kernel. We will condense these properties into the concept of *expressive kernelization*. Moreover, in the sense that a problem kernel with $O(k^{d-\varepsilon})$ hyperedges for some $\varepsilon > 0$ would lead to a collapse of the polynomial-time hierarchy, the size of our problem kernel is optimal.

We implement our kernelization algorithm and evaluate its applicability to the construction of Golomb rulers with a maximum number of marks. We find instances with more than 10^7 hyperedges to be kernelizable in five minutes.

Finally, using ideas of Abu-Khzam [Abu10] and Moser [Mos10], we show that the number of vertices can be further reduced to $O(k^{d-1})$ with an additional amount of $O(k^{1.5d})$ time. We can thus compute in $O(n+m+k^{1.5d})$ time a problem kernel comprising $O(k^d)$ hyperedges and $O(k^{d-1})$ vertices.

5.1.3 Chapter outline

We start by giving a concept of expressive kernelization in Section 5.2.

Then, we present an expressive linear-time kernelization algorithm for d-Hitting Set in Section 5.3, which we evaluate experimentally on hypergraphs occurring in the computation of optimal Golomb rulers in Section 5.4.

Finally, we show how the number of vertices can be reduced to $O(k^d)$ in additional $O(k^{1.5k})$ time in Section 5.5. Since the resulting problem kernel is not expressive, we have not implemented it.

5.2 Expressive kernelization

The core component of our linear-time kernelization algorithm for d-Hitting Set is an algorithm to find and shrink *sunflowers* in linear time. Sunflowers are special constellations of hyperedges that Erdős and Rado [ER60] discovered to appear in any sufficiently large hypergraph and their use in kernelization algorithms for d-Hitting Set is a standard technique [FG06; FSV13; Kra12]. They are defined as follows and illustrated in Figure 5.1.

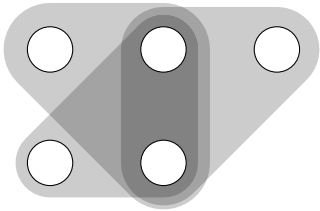


Figure 5.1: A sunflower with three petals and two core elements.

Definition 5.1. A *sunflower* in a hypergraph $H = (V, E)$ is a set of *petals* $P \subseteq E$ such that each pair of sets in P intersects in exactly the same set $C \subseteq V$, which is called the *core* (possibly, $C = \emptyset$). The *size* of the sunflower is $|P|$.

The approach of finding and shrinking sunflowers yields problem kernels that contain more structural information than the formal definition of problem kernels requires. Specifically, sunflowers help computing problem kernels that have the following three properties, which we henceforth require to be guaranteed by *expressive* problem kernels for d-Hitting Set and that we will describe in more detail in the following.

Definition 5.2. A kernelization algorithm for d-Hitting Set is *expressive* if, given an instance (H, k), it outputs an instance (H', k') such that

i) H' is a subgraph of H,

ii) any vertex set of size at most k is a minimal hitting set for H if and only if it is a minimal hitting set for H', and

iii) it outputs a certificate for (H', k') being a yes-instance if and only if (H, k) is.

Interpretability of the problem kernel. The kernelization algorithm should output a subgraph of the input hypergraph. Kernelization algorithms for d-Hitting Set with this explicit goal have been developed by Moser [Mos10] and Kratsch [Kra12], since newly introduced hyperedges or vertices in the problem kernel might not be interpretable in the context of the original problem modeled as d-Hitting Set. Kratsch [Kra12] exploited this property to show polynomial-size problem kernels for a large class of problems formalizable as d-Hitting Set. In some scenarios, as pointed out by Abu-Khzam and Fernau [AF06], it is

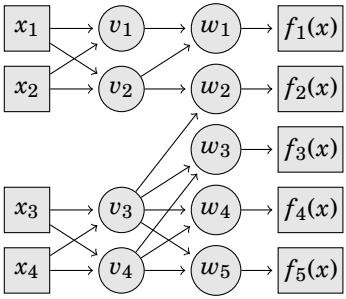
even desirable that the kernelization algorithm outputs an *induced* subgraph of the input hypergraph. However, our problem kernel for d-Hitting Set will not satisfy this requirement.

Interpretability of solutions. Any vertex set of size at most k should be a minimal hitting set for the resulting problem kernel if and only if it is a minimal hitting set for the original instance. If the input instance and the problem kernel allow for exactly the same minimal hitting sets of size at most k, the problem kernel retains enough information for interpreting solutions and finding alternative solutions without having to consider the input hypergraph. This property has been exploited by Fomin, Saurabh, and Villanger [FSV13] as an important building block in a polynomial-size problem kernel for a problem that cannot easily be modeled as d-Hitting Set for *constant* d. As pointed out by Fomin, Saurabh, and Villanger [FSV13], this property is stronger than those guaranteed by the *full kernels* introduced by Damaschke [Dam06]: full kernels contain all minimal hitting sets of size at most k for the input hypergraph, but not necessarily the information whether a hitting set is minimal.

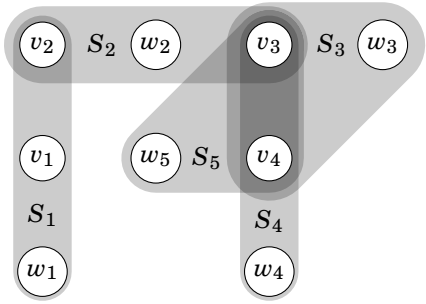
Certifying data reduction. Similarly to how certifying algorithms provide a certificate for the correctness of their output [MMNS11], an expressive kernelization algorithm should provide a certificate for the correctness of the executed data reduction. Ideally, the proof that a certificate indeed certifies the correctness should be easily understandable, so that a human can easily verify the executed data reduction to be correct without having to trust on the correctness of algorithms and their implementations. A sunflower P with $k+1$ petals in a d-Hitting Set instance fulfills this requirement: every hitting set S of size at most k contains an element of the core C of P, since otherwise S cannot contain an element of each of the $k+1$ petals. Thus, any additional hyperedge in the hypergraph that contains C already contains an element of S; it is redundant and may be removed. The sunflower P is a certificate for this being correct.

Example 5.1. Sunflowers not only certify the correctness of data reduction, but also lead the way to alternative solutions. We illustrate this using an example of d-Hitting Set in a fault diagnosis context.

Figure 5.2(a) represents a Boolean circuit. It gets as input a 4-bit string $x = x_1 \dots x_4$ and outputs a 5-bit string $f(x) = f_1(x) \dots f_5(x)$. The nodes drawn as circles represent Boolean gates, which output some bit depending on their two input bits. They might, for example, represent the logical operators $\wedge$ or $\vee$. Assume



(a) A Boolean circuit with circle nodes representing gates and square nodes representing input and output nodes.



(b) Sets containing at least one faulty gate, found by the analysis of the circuit.

Figure 5.2: Illustrations for Example 5.1.

that all output bits of $f(x)$ are observed to be the opposite of what would have been expected by the designer of the circuit. We want to identify broken gates. For each wrong output bit $f_i(x)$, we obtain a set S_i of gates for which we know that at least one is broken because $f_i(x)$ is wrong. That is, S_i contains precisely those gates that have a directed path to $f_i(x)$ in the graph shown in Figure 5.2(a). We obtain the sets $S_1,\ldots,S_5$ illustrated in Figure 5.2(b).

The sets S_1 and S_4 are disjoint. Hence, the wrong output is not explainable by only one broken gate. We thus assume *two* broken gates and search for a hitting set of size $k=2$ in the hypergraph with the vertices $v_1,\ldots,v_4,w_1,\ldots,w_5$ and hyperedges $S_1,\ldots,S_5$. The set $\{S_3,S_4,S_5\}$ is a sunflower of size $k+1=3$ with core $\{v_3,v_4\}$. Therefore, the functionality of gate v_3 and v_4 must be checked. If, in contrast to our expectations, both gates v_3 and v_4 turn out to be working correctly, the sunflower shows not only that at least three gates are broken, but also shows which gates have to be checked for malfunctions next: w_3, w_4, and w_5.

There are few expressive kernelization algorithms for d-Hitting Set in the literature. For example, the algorithm of Abu-Khzam [Abu10] does not yield a subgraph of the input hypergraph. Thus, it does not satisfy Definition 5.2(i), which has been "fixed" by Moser [Mos10]. However, both problem kernels may discard minimal solutions of size at most k and, thus, do not satisfy Definition 5.2(ii). Damaschke [Dam06] designed a problem kernel that retains all

minimal solutions of size at most k. However, it is not certifying and thus does not satisfy Definition 5.2(iii).

We are aware of only one expressive kernelization algorithm for d-Hitting Set: this is the problem kernel shown by Kratsch [Kra12], which is also used by Fomin, Saurabh, and Villanger [FSV13]. This is precisely the algorithm we will improve to run in linear time.

5.3 A linear-time kernelization algorithm

This section shows a linear-time computable problem kernel for d-Hitting Set comprising $O(k^d)$ hyperedges. That is, we show that a hypergraph H can be transformed in linear time into a hypergraph G such that G has $O(k^d)$ hyperedges and allows for a hitting set of size k if and only H does. In Section 5.5, we show how to shrink the number of vertices to $O(k^{d-1})$.

Theorem 5.1. *d-Hitting Set allows for an expressive problem kernel with $d! \cdot d^{d+1} \cdot (k+1)^d$ hyperedges and d times as many vertices that is computable in $O(d \cdot n + 2^d d \cdot m)$ time.*

We prove Theorem 5.1 with the help of the sunflower lemma of Erdős and Rado [ER60], who showed that every sufficiently large hypergraph contains a sunflower with $k+2$ petals: if we shrink all of these sunflowers, it follows that the resulting hypergraph will be small. Kernelization algorithms based on this strategy, like those of Flum and Grohe [FG06] and Kratsch [Kra12] usually proceed along the lines of repeatedly

- finding a sunflower of size $k+2$ in the input hypergraph and
- deleting redundant petals until no more sunflowers of size $k+2$ exist.

This approach has the drawback of finding only one sunflower at a time and restarting the process from the beginning.

In contrast, to prove Theorem 5.1, we construct a subgraph G of a given hypergraph H not by hyperedge deletion, but by a bottom-up approach that allows us to "grow" many sunflowers in G simultaneously, stopping "growing sunflowers" when they become too large.

Algorithm 5.1 repeatedly (after some initialization work in lines 1–6) in line 9 copies a hyperedge e from H to the initially empty G unless we find in line 8 that e contains the core C of a sunflower of size $k+1$ in G. We maintain the number of

petals found for a core C in petals[C]. If we find that a hyperedge e can be added to a sunflower with core C in line 11, then we increment petals[C] in line 12 and mark the vertices in $e \setminus C$ as "used" for the core C in line 13. This information is maintained by setting "used[C][v] $\leftarrow$ true." In this way, vertices in $e \setminus C$ are not considered again for finding petals for the core C in line 11, therefore ensuring that petals later found for the core C intersect e only in C.

Algorithm 5.1: Linear-Time kernelization for d-Hitting Set

Input: A hypergraph $H = (V, E)$ and a natural number k.
Output: A hypergraph $G = (V', E')$ with $|E'| \in O(k^d)$.

```
1  E' ← ∅
2  foreach e ∈ E do                                   // Initialization for each hyperedge
3      foreach C ⊆ e do                               // Initialization for all possible cores of sunflowers
4          petals[C] ← 0                              // No petals found for sunflower with core C yet
5          foreach v ∈ e do
6              used[C][v] ← false                     // No vertex v is in a petal of a
                                                      // sunflower with core C yet
7  foreach e ∈ E do
8      if ∀C ⊆ e: petals[C] ≤ k then
9          E' ← E' ∪ {e}
10         foreach C ⊆ e do                           // Consider all possible cores for the petal e
11             if ∀v ∈ e \ C: used[C][v] = false then
12                 petals[C] ← petals[C] + 1
13                 foreach v ∈ e \ C do used[C][v] ← true
14 V' := ⋃_{e∈E'} e
15 return (V', E')
```

By storing in petals[C] a list of found petals, the algorithm can output the discovered sunflowers without any increase in running time. Thus, it gives certificates for the correctness of the executed data reduction.

It is important to note that, as illustrated in Figure 5.3, the value in petals[C] is not necessarily the size of the largest possible sunflower with core C, but depends on the order in that Algorithm 5.1 processes the hyperedges of the input hypergraph. Computing the size of a largest sunflower with core C is, for $C = \emptyset$, the problem of computing a maximum matching in a hypergraph, which is NP-hard [Kar72].

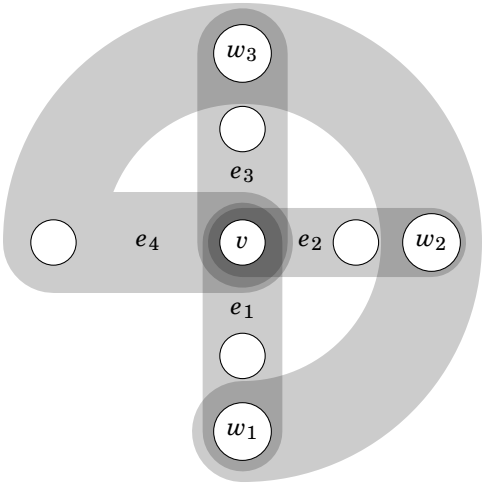


Figure 5.3: The result of applying Algorithm 5.1 depends on the order in that it processes the hyperedges of the input hypergraph H. If applied for $k = 2$, then Algorithm 5.1 will not add the hyperedge e_4 to the output hypergraph G if it before added e_1, e_2, and e_3, since it discovers e_4 to contain the core $\{v\}$ of the sunflower $\{e_1, e_2, e_3\}$ with $k + 1 = 3$ petals. However, if it first adds e_4 to G, then it marks the vertices w_1, w_2, and w_3 as used for the sunflower with core $\{v\}$. Thus, none of the hyperedges e_1, e_2, and e_3 is recognized as a petal for a sunflower with core $\{v\}$ and all shown hyperedges are added to the output hypergraph.

Towards proving Theorem 5.1, we now proceed as follows. Section 5.3.1 shows that Algorithm 5.1 is correct and expressive. Section 5.3.2 shows that the hypergraph output by Algorithm 5.1 contains $O(k^d)$ hyperedges. Finally, Section 5.3.3 shows that Algorithm 5.1 runs in linear time.

5.3.1 Correctness

On our way to proving that d-Hitting Set has an expressive linear-time computable problem kernel with $O(k^d)$ hyperedges and thus proving Theorem 5.1, we now prove the correctness and expressiveness of Algorithm 5.1. This, together with a proof for the size of the output hypergraph and a proof of the running time of Algorithm 5.1, will provide a proof of Theorem 5.1.

Proposition 5.1. *Let G be the hypergraph returned by Algorithm 5.1 when given a hypergraph H and an integer k. Then,*

i) any hitting set S of size at most k for G is a hitting set for H and

ii) any minimal hitting set S of size at most k for H is a hitting set for G.

Moreover, G is a subgraph of H and any subset S of at most k vertices of H is a minimal hitting set for H if and only if it is a minimal hitting set for G.

Proof. By construction of G from H in Algorithm 5.1, it is clear that G is a subgraph of H. We now show that it is sufficient to prove (i) and (ii) to also conclude the last statement of Proposition 5.1: let S be a minimal hitting set of size at most k for H. By (ii), it is a hitting set for G. Assume, towards a contradiction, that S is not a *minimal* hitting set for G. Then, there is a hitting set $S' \subsetneq S$ for G. However, by (i), $S' \subsetneq S$ is also a hitting set for H. This contradicts S being a minimal hitting set for H. Symmetrically, let S be a minimal hitting set S of size at most k for G. By (i), S is a hitting set for H. Assume, towards a contradiction, that S is not a *minimal* hitting set for H. Then, there is a minimal hitting set $S' \subsetneq S$ for H. However, by (ii), $S' \subsetneq S$ is also a hitting set for G. This contradicts S being a minimal hitting set for G. It remains to prove (i) and (ii).

(i) Let S be a hitting set of size at most k for G. Obviously, all hyperedges that H and G have in common are hit in H by S. We show that every hyperedge e in H that is not in G is also hit. If e is in H but not in G, then adding e to G in line 9 of Algorithm 5.1 has been skipped because the condition in line 8 is false. That is, petals$[C] \geq k+1$ for some $C \subseteq e$. Consequently, for this particular C, a sunflower P with $k+1$ petals and core C exists in G, since we only increment petals$[C]$ in line 12 if we find a suitable additional petal for the sunflower with core C in line 11. Note that $C \neq \emptyset$ because, otherwise, $k+1$ pairwise disjoint hyperedges would exist in G, contradicting our assumption that S is a hitting set of size k for G. Since $|S| \leq k$, we have $S \cap C \neq \emptyset$. Therefore, since $C \subseteq e$, the hyperedge e is hit by S also in H.

(ii) Let S be a minimal hitting set of size at most k for $H = (V, E)$. The set $S' := S \cap V'$ is a hitting set for $G = (V', E')$ with $S' \subseteq S$: the set S contains an element of every hyperedge in E and, since $E' \subseteq E$ and $V' = \bigcup_{e \in E'} e$, the set S' contains an element of every hyperedge in E'. By (i), S' is also a hitting set for H. Since $S' \subseteq S$ and we required S to be a *minimal* hitting set of size at most k for H, we have that $S' = S$ and, thus, that S is a hitting set for G. □

5.3.2 Problem kernel size

Having shown that Algorithm 5.1 is correct, we now show that the hypergraph output by Algorithm 5.1 contains $O(k^d)$ hyperedges. To prove Theorem 5.1, it then remains to prove that Algorithm 5.1 runs in linear time.

In order to show an upper bound on the size of the hypergraph output by Algorithm 5.1, we exploit an upper bound on the size of the sunflowers in the output hypergraph:

Lemma 5.1. *Let G be the hypergraph output by Algorithm 5.1 applied to a hypergraph H and a natural number k. Every sunflower P in G with core $C \notin P$ has size at most $d(k+1)$.*

Proof. Let P be a sunflower in G with core $C \notin P$. Then, $|P| \leq d(k+1)$ follows from the following two observations:

(i) Every petal $e \in P$ present in G is copied from H in line 9 of Algorithm 5.1. Consequently, every petal $e \in P$ contains a vertex v satisfying used$[C][v]$ = true: if this condition is violated in line 11, then line 13 applies "used$[C][v] \leftarrow$ true" to all vertices $v \in e \setminus C$.

(ii) Whenever petals$[C]$ is incremented by one in line 12, then, in line 13, "used$[C][v] \leftarrow$ true" is applied to the at most d vertices $v \in e$. Thus, since petals$[C]$ never exceeds $k+1$, at most $d(k+1)$ vertices v satisfy used$[C][v]$ = true. Moreover, since, by line 13, no $v \in C$ satisfies used$[C][v]$ = true and the petals in P pairwise intersect only in C, it follows that at most $d(k+1)$ petals in P contain vertices satisfying used$[C][v]$ = true. □

Having shown an upper bound on the size of the sunflowers in the hypergraph output by Algorithm 5.1, we now show that the output hypergraph contains $O(k^d)$ hyperedges. To this end, in a way similar to Flum and Grohe [FG06, Lemma 9.7], we show the following refined version of Erdős and Rado's [ER60] sunflower lemma. Herein, recall that a hypergraph is ℓ-uniform if and only if every hyperedge has cardinality exactly ℓ.

Lemma 5.2. *Let H be an ℓ-uniform hypergraph and $b, c \in \mathbb{N}$ with $b \leq \ell$ such that every pair of hyperedges in H intersects in at most $\ell - b$ vertices.*

If H contains more than $\ell! c^{\ell+1-b}$ hyperedges, then H contains a sunflower with more than c petals.

For $b = 1$, we obtain the sunflower lemma stated by Flum and Grohe [FG06]. For $b = 2$, we will exploit it in Section 5.5 to reduce the number of vertices in the output hypergraph.

Proof. We prove the lemma by induction on ℓ. As base case, consider $\ell = b$. For $\ell = b$, all hyperedges in H are pairwise disjoint. Hence, if H has more than $\ell! c^{\ell+1-b}$ hyperedges, then these form a sunflower with empty core and more than $\ell! c^{\ell+1-b} = \ell! c \geq c$ petals. That is, the lemma holds for $\ell = b$.

Now, assume that the lemma holds for some $\ell \geq b$. It remains to prove that it holds for $\ell + 1$. Let M be a maximal set of pairwise disjoint hyperedges of the $(\ell+1)$-uniform hypergraph $H := (V, E)$. If $|M| > c$, then the lemma holds because M is a sunflower with empty core. Otherwise, for $N := \bigcup_{e \in M} e$, it holds that $|N| \leq (\ell+1)c$ and some vertex $w \in N$ is contained in a set E_w of more than

$$\frac{|E|}{|N|} \geq \frac{(\ell+1)! c^{\ell+2-b}}{(\ell+1)c} = \ell! c^{\ell+1-b} \text{ hyperedges.}$$

The hypergraph H_w that contains for each hyperedge $e \in E_w$ the hyperedge $e \setminus \{w\}$ is an ℓ-uniform hypergraph and, by induction hypothesis, contains a sunflower P with more than c petals. Adding w to each of the petals of P yields a sunflower P' with more than c petals in H. □

By combining Lemma 5.1 with Lemma 5.2, we can easily show that the hypergraph output by Algorithm 5.1 contains $O(k^d)$ hyperedges. Since we have already shown in Proposition 5.1 that the algorithm is correct, it thereafter only remains to show that Algorithm 5.1 runs in linear time in order to complete the proof of Theorem 5.1.

Proposition 5.2. *The hypergraph G returned by Algorithm 5.1 on input H and k contains at most $d! \cdot d^{d+1} \cdot (k+1)^d$ hyperedges and d times as many vertices.*

Proof. Obviously, G has at most d times as many vertices as hyperedges, since the vertex set of G is constructed as the union of its hyperedges in line 14 of Algorithm 5.1.

To bound the number of hyperedges, consider, for $1 \leq \ell \leq d$, the ℓ-uniform hypergraph $G_\ell = (V_\ell, E_\ell)$ comprising only the hyperedges of size ℓ of G. If G had more than $d! \cdot d^{d+1} \cdot (k+1)^d$ hyperedges, then, for some $\ell \leq d$, G_ℓ would have more than $d! \cdot d^d \cdot (k+1)^d$ hyperedges. Lemma 5.2 with $b = 1$ and $c = d(k+1)$ states that, if G_ℓ had more than $\ell! \cdot d^\ell \cdot (k+1)^\ell$ hyperedges, then G_ℓ would contain a sunflower P with core C and more than $d(k+1)$ petals. Obviously, $C \notin P$, since all petals have cardinality ℓ. Moreover, this sunflower would also exist in the supergraph G of G_ℓ, contradicting Lemma 5.1. □

5.3.3 Running time

Since Proposition 5.1 has shown that Algorithm 5.1 is correct and Proposition 5.2 has shown that the output hypergraph contains $O(k^d)$ hyperedges, to prove Theorem 5.1, it remains to show that Algorithm 5.1 runs in linear time. In order to implement the algorithm efficiently, we need data structures that allow us to quickly look up the values petals[C] and used[C] for some vertex set C of size at most d.

Earlier, in Chapter 3, for our dynamic programming algorithm for COLORFUL INDEPENDENT SET WITH LISTS, we realized table look-ups for subsets of some universe of size γ in $O(\gamma)$ time by representing the subsets as bitstrings of length γ and looking up these in a trie [AHU83, Section 5.3]. However, now, our universe is the set of vertices of the input hypergraph and, thus, has size n. Hence, this method would yield table look-ups in $\Theta(n)$ time, which is too slow to prove that Algorithm 5.1 runs in linear time. For this reason, we will not represent vertex subsets of size at most d as bitstrings, but uniquely represent them as sorted sequences of at most d integers. Then, we will exploit the following lemma.

Lemma 5.3. *Let L be a list of sequences of length at most d of integers in $[n]$.*

In $O(d \cdot n + d \cdot |L|)$ time, we can compute an associative array $A[]$ such that, for each sequence s in L, accessing the value $A[s]$ and storing a value to $A[s]$ works in $O(d)$ time.

Proof. We use a trie to associate values with sequences in L. However, the trie will be too large to initialize it fully in linear time. We have to show that we can create the trie so that, for a look-up of a value for any sequence s in L, no uninitialized memory cells are read.

We define a *trie* as a size-n array, of which each cell contains a pointer to a structure consisting of two more pointers: one of them points to data, the other one to another trie. This is illustrated in Figure 5.4. A look-up of the value associated with a sequence $s = (s_1, \ldots, s_d)$ in a trie T_1 then works in $O(d)$ time as follows: for $i \in [d-1]$, we get the trie T_{i+1} pointed to by $T_i[s_i]$. Then, from $T_d[s_d]$, we get a pointer to the data associated with s.

In the creation of the trie to associate values with the sequences in L, we now face a bigger version of a problem that we already had to solve in our linear-time data reduction algorithm for DAG PARTITIONING in Section 4.3.2: we do not have enough time to initialize all cells of all arrays that implement the inner nodes of the trie: this would take $\Theta(n)$ time per node and, as seen in Figure 5.4, the number of nodes required in the trie can be more than $|L|$. This is a problem since, when creating the trie, we do not know whether an array cell already

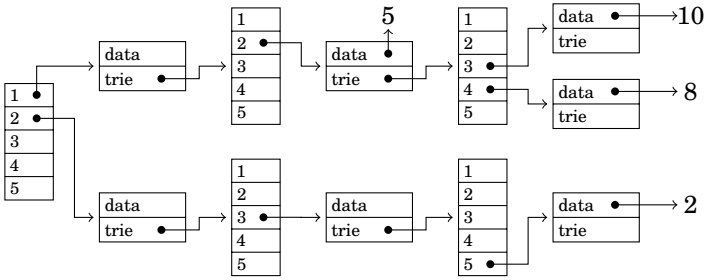


Figure 5.4: A trie that associates integer values with sequences of integers in $\{1,\ldots,5\}$. That is, each node of the trie is an array of size five. With $(1,2)$ the trie associates 5, with $(1,2,3)$ it associates 10, with $(1,2,4)$ it associates 8, and, finally, with $(2,3,5)$ it associates 2.

contains a pointer to a trie node of the next level or whether we have to create such a pointer with a corresponding new node. We have to make sure that we only follow pointers of initialized cells and that we do not overwrite previously correctly set up pointers since, otherwise, information in subtries will be lost. We achieve this as follows:

The input list L contains sequences of length at most d of integers in $[n]$. Hence, we can sort L lexicographically in $O(d \cdot (n + |L|)) = O(d \cdot n + d \cdot |L|)$ time using radix sort [CLRS01, Section 8.3]. We construct the trie by iterating over L once. For each sequence p in L, we find in $O(d)$ time the first position i in which the sequence p differs from its predecessor sequence in the lexicographically sorted list L. This tells us that we already created all nodes on the path from the trie's root node to the leaf corresponding to s up to a depth of i. Pointers up to this depth i are valid and may not be overwritten. Nodes and pointers beyond this depth have to be newly created. □

Using Lemma 5.3, we can finally prove that Algorithm 5.1 runs in linear time. Note that, together with Propositions 5.1 and 5.2, Proposition 5.3 completes the proof of Theorem 5.1.

Proposition 5.3. *Algorithm 5.1 can be implemented to run in* $O(d \cdot n + 2^d d \cdot m)$ *time.*

Proof. We first describe how Lemma 5.3 helps us efficiently implementing the associative arrays petals[] and used[] required by Algorithm 5.1. To this end,

we assume that every vertex is represented as an integer in $[n]$ and that every hyperedge is represented as a sequence sorted by increasing vertex numbers, which we call *sorted hyperedge*. We can initially sort each hyperedge of H in $O(m \cdot d \log d)$ total time. Note that, on hyperedges represented as sorted sequences, the set subtraction operation needed in line 11 can be executed in $O(d)$ time such that the resulting set is again sorted [AHU83, Section 4.4]. Moreover, we can generate all subsets of a sorted set such that the resulting subsets are sorted. Hence, we may assume to always deal with sorted hyperedges as a unique representation of hyperedges.

We now apply Lemma 5.3. Observe that Algorithm 5.1 looks up petals[C] and used[C] only for sets $C \subseteq e$ for some hyperedge e. Thus, from the set of sorted hyperedges, in $O(2^d d \cdot m)$ time, we compute a length-$(2^d \cdot m)$ list L of all possible sets $C \subseteq e$ for all hyperedges e and use this list in Lemma 5.3 to create the associative arrays petals[] and used[] in $O(d \cdot n + d \cdot |L|) = O(d \cdot n + 2^d d \cdot m)$ time.

Now, we can implement lines 1–6 of Algorithm 5.1 to run in $O(d \cdot n + 2^d d \cdot m)$ time, observing that the loop in line 5 can be implemented to run in $O(d)$-time, since only one look-up to used[C] is needed to obtain a pointer to an array in which, then, $O(d)$ values are set.

The for-loop in line 7 iterates m times. Its body works in $O(2^d d)$ time: obviously, this time bound holds for lines 8 and 9; it remains to show that the body of the for-loop in line 10 works in $O(d)$ time. This is easy to see if one considers that, in lines 11 and 13, one only has to do one look-up to used[C] to find a pointer to an array that holds the values for the at most d vertices of a hyperedge. Also line 14 works in linear time by first initializing all entries of an array vertices[] of size n to "false" and then, for each output hyperedge e and each vertex $v \in e$, setting "vertices[v] $\leftarrow$ true" in $O(d)$ time. Afterward, we can build the vertex set V' of the output hypergraph G using the vertices v for which vertices[v] = true. This takes $O(n + d \cdot m)$ time. □

5.4 Experimental evaluation

This section experimentally evaluates the linear-time kernelization algorithm from Section 5.3. Like previously in our experiments for Colorful Independent Set with Lists in Chapter 3 and for DAG Partitioning in Chapter 4, we demonstrate to which size our algorithm can process instances within five minutes.

Implementation details. Our implementation of Algorithm 5.1 comprises about 700 lines of C++ and is freely available.[2] The experiments were run on a computer with a 3.6 GHz Intel Xeon processor and 64 GB RAM under Linux 3.2.0, where the C++ source code has been compiled using the GNU C++ compiler in version 4.7.2 and using the highest optimization level (-O3).

Given a hypergraph $H = (V, E)$, our implementation of Algorithm 5.1 checks for each hyperedge $e \in E$ with $\ell := |e|$ independently whether it is a *large hyperedge* ($2^\ell > m$) or a *small hyperedge* ($2^\ell \leq m$). For a small hyperedge e, Algorithm 5.1 chooses to consider all subsets $C \in e$ as possible cores of sunflowers in line 8. For a large hyperedge e, all subsets $e \cap e'$ for any $e' \in E$ are considered as possible cores instead. Hence, our implementation chooses the variant which promises the lower running time for each hyperedge independently.

Additionally to discarding hyperedges that contain some core of a sunflower of size $k+1$, our implementation of Algorithm 5.1 also makes sure that the output hypergraph contains no pair of hyperedges such that one is a superset of the other. To this end, the implementation initially sorts all hyperedges by increasing cardinality in $O(d+m)$ time using counting sort [CLRS01, Section 8.2]. Moreover, after adding a hyperedge e to the output hypergraph, the implementation sets petals[e] to $k+1$: in this way, the algorithm will not add hyperedges to the output hypergraph that are supersets of already added hyperedges.

As data structures to hold the values used[C] and petals[C] used by Algorithm 5.1 to associate values with sets C of size at most d, we implemented the following variants.

By *malloc trie*, we refer to the associative array created in Lemma 5.3. It is implemented as a trie whose nodes are allocated as uninitialized arrays in constant time using the C-routine `malloc`. It guarantees $O(d)$ look-up time and, as described in the proof of Proposition 5.3, $O(d \cdot n + 2^d d \cdot m)$ creation time. However, $\Omega(n \cdot m)$ random access memory cells may need to be reserved by the program in the worst case, although at most $O(d \cdot n + 2^d d \cdot m)$ memory cells are actually accessed.

By *calloc trie*, we refer to a trie whose nodes are allocated as arrays pre-initialized by zero using the C-routine `calloc`. This makes the intricate initialization by Lemma 5.3 unnecessary. However, the running time of acquiring a zero-initialized array and its actual memory usage may vary depending on

[2] http://fpt.akt.tu-berlin.de/hslinkern/

the implementation of the routine by the used C library. For naïve implementations of `calloc`, creation time and memory usage of the calloc tree could be $\Omega(n \cdot m)$ in the worst case. However, we can still guarantee $O(d)$ look-up time.

By *hash table*, we refer to an associative array implemented using the data structure `unordered_map` provided in the C++11 Standard Template Library. At most $O(2^d \cdot m)$ values are stored in the hash table. According to the C++11 reference, storage and look-up work in $O(2^d \cdot m)$ time in the worst case, but in $O(d)$ time in the average case, where $O(d)$ time accounts for computing the hash value of a hyperedge of cardinality d.

By *balanced tree*, we refer to an associative array implemented using the `map` data structure provided by the C++11 Standard Template Library. According to the C++11 reference, it is usually implemented as a balanced binary tree. Since, in the worst case, $O(2^d \cdot m)$ values are stored in the tree, the C++11 reference guarantees $O(d + \log m)$ time for storage and look-up. Its memory requirements are $O(2^d m)$.

Data. We execute our experiments on hypergraphs generated from the GOLOMB SUBRULER problem: one gets as input a set $R \subseteq \mathbb{N}$ and wants to remove at most k numbers ("marks") from R such that the result is a Golomb ruler, that is, no pair of remaining marks has the same distance as another pair. As we briefly discussed in Chapter 1, the applications of Golomb rulers lie, among others, in radio frequency allocation [Dra09]. Optimum solutions for GOLOMB SUBRULER are only known for $R = [n]$ with $n \leq 553$ at the current time.[3]

From a GOLOMB SUBRULER instance, we obtain a *conflict hypergraph* as follows: the vertex set is R, and for each $a, b, c, d \in R$, create a hyperedge $\{a, b, c, d\}$ if $|a-b| = |c-d|$. Asking for a hitting set of size k in this conflict hypergraph is equivalent to GOLOMB SUBRULER [SMNW14]. As shown by Sorge et al. [SMNW14], the class of conflict hypergraphs for $R = [n]$ has n vertices and $\Theta(n^3)$ hyperedges, their cardinality being three or four. Our data set consists of the conflict hypergraphs for GOLOMB SUBRULER instances $R = [n]$ with $100 \leq n \leq 600$, which yields conflict hypergraphs with 10^5 to $2 \cdot 10^7$ hyperedges. Since, in this way, we obtain a whole family of growing hypergraphs, this data set is well-suited to show the running time and memory scalability of Algorithm 5.1.

[3] http://blogs.distributed.net/2014/02/

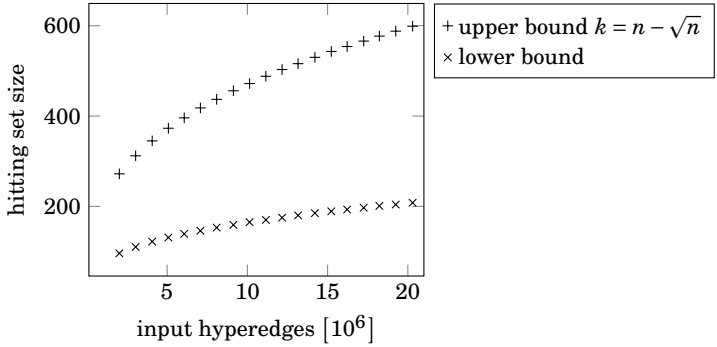


Figure 5.5: Upper and lower bounds for the minimum hitting set sizes for the data set obtained from the GOLOMB SUBRULER problem. The number of vertices n in the input instances is omitted, as it almost coincides with the upper bound $k = n - \sqrt{n}$ of the hitting set size. The lower bound was obtained from a maximal set of pairwise disjoint hyperedges.

Experimental setup. Algorithm 5.1 requires as input not only a hypergraph H but also an upper bound k on the size of a sought hitting set. We choose as k an upper bound on the size of a *minimum* hitting set, so that the kernelization algorithm will not output small trivial no-instances and so that the computed problem kernel will retain all minimum hitting sets.

To obtain this upper bound for the 4-HITTING SET instances that we obtain from GOLOMB SUBRULER, we exploit that, for all $n \leq 4.2 \cdot 10^9$, Dimitromanolakis [Dim02, Theorem 6.1] verified that there is a Golomb ruler $R \subseteq [n]$ with strictly more than $\sqrt{n}$ marks. Hence, in our experiments with $n \leq 600$, a conflict hypergraph of a GOLOMB SUBRULER instance $R = [n]$ has a hitting set of size at most $k := \lfloor n - \sqrt{n} \rfloor$. We use this k to compute problem kernels for 4-HITTING SET. Figure 5.5 shows this upper bound together with a lower bound.

Experimental results. In all plots to be shown, each point has been obtained from a single run of our algorithm; the running times and memory usage are not averaged in any way.

Figure 5.6 shows the performance of our kernelization algorithm on conflict hypergraphs of the GOLOMB SUBRULER problem of size up to $2 \cdot 10^6$ hyperedges. On larger instances, the implementations based on malloc tries, calloc tries, and

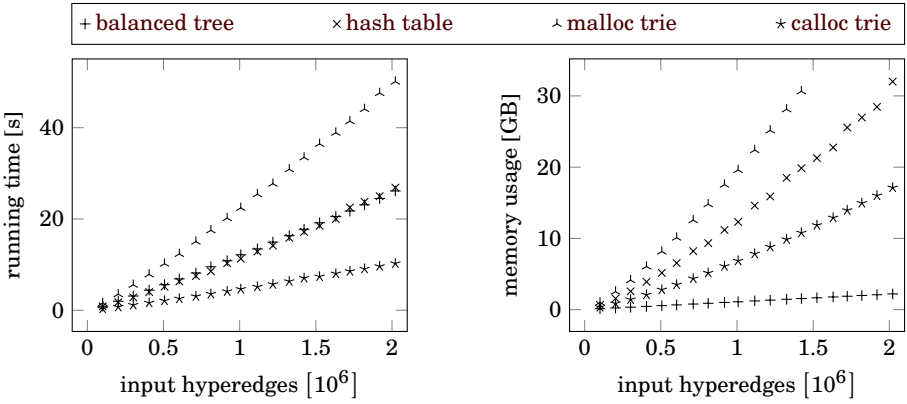


Figure 5.6: Performance of Algorithm 5.1 on conflict hypergraphs of the GOLOMB SUBRULER problem with at most $2 \cdot 10^6$ hyperedges.

hash tables hit the 32 GB memory limit of the `valgrind` memory measuring tool. One can observe that the implementation using the malloc trie is the slowest. This is due to the complicated initialization procedure required by Lemma 5.3. The fastest implementation is the variant using the calloc trie, which is the same as the malloc trie implementation except that we skip the intricate initialization of the trie using Lemma 5.3. Unsurprisingly, the memory usage of the balanced tree implementation is the lowest, as it grows linearly with the number of stored elements. Surprisingly, the hash table implementation of the GNU C++ compiler consumes even more memory than our calloc trie.

Since the malloc trie, calloc trie, and hash table reach the 32 GB memory limit of the `valgrind` memory measurement tool between $2 \cdot 10^6$ and $5 \cdot 10^6$ input hyperedges, we made ongoing experiments only with the balanced tree implementation. Thus, unfortunately, we were unable to see how the running time of our fastest implementation—using calloc tries—scales to larger instances. Figure 5.7 shows that the implementation using the balanced tree solves 4-HITTING SET instances on conflict hypergraphs of GOLOMB SUBRULER with $20 \cdot 10^6$ hyperedges in less than five minutes and does not even hit the 32GB memory limit of the `valgrind` memory measurement tool.

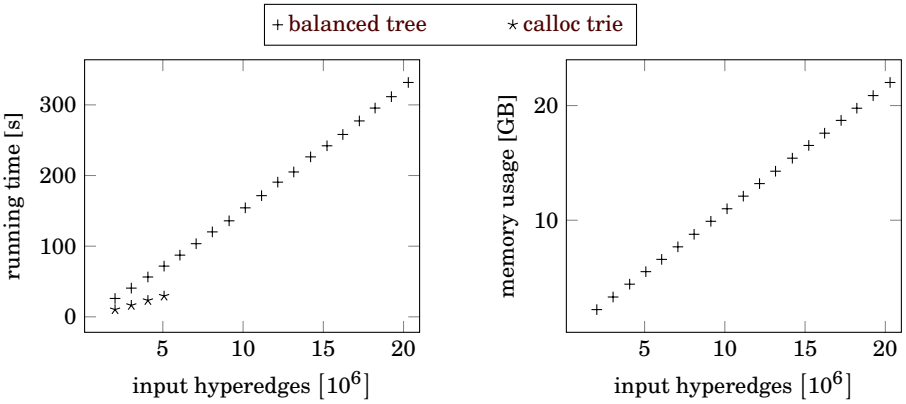


Figure 5.7: Performance of Algorithm 5.1 on conflict hypergraphs of the GOLOMB SUBRULER problem with at most $20 \cdot 10^6$ hyperedges.

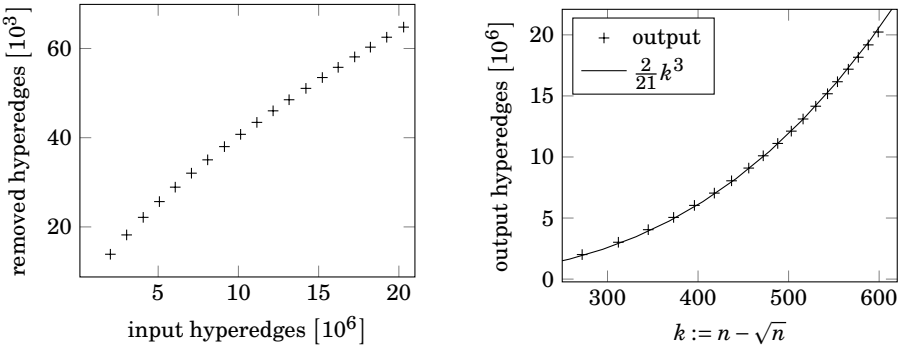


Figure 5.8: Size of the resulting problem kernel when Algorithm 5.1 is applied to conflict hypergraph of the GOLOMB SUBRULER problem.

Effect of data reduction. As shown in Figure 5.8, the kernelization algorithm removed between $20 \cdot 10^3$ and $60 \cdot 10^3$ hyperedges from the input instances. Thus, although we observed the algorithm to handle large input instances well, the observed data reduction effect is rather limited. This is partly due to the lack of a better upper bound k for the size of the sought hitting set: we observed that the input 4-Hitting Set instances obtained from Golomb Subruler have roughly $1/12 \cdot n^3$ hyperedges. This, for any $n \geq 0$, is already below the upper bound of $4! \cdot 4^5 \cdot (k+1)^4$ with $k = \lfloor n - \sqrt{n} \rfloor$ given by Theorem 5.1 on the problem kernel size for 4-Hitting Set.

The limited effect of the data reduction on 4-Hitting Set instances obtained from Golomb Subruler is also due to the fact that they are nearly 4-uniform: this prevents hyperedges from getting deleted for being supersets of smaller hyperedges. The presence of smaller hyperedges can significantly influence the output instance size. As an extreme example, consider a hypergraph containing $\binom{n}{2}$ hyperedges of cardinality two. Then, the output problem kernel will contain $O(k^2)$ output hyperedges, regardless of how many more input hyperedges of cardinality 100 there are. Such phenomena are not captured in the theoretical upper bound on the problem kernel size given by Theorem 5.1, which is based on the analysis of uniform hypergraphs.

As shown in Figure 5.8, when measuring the size of the problem kernels in k, we observe that the resulting problem kernels contain about $2/21 \cdot k^3$ hyperedges. Thus, our empirically measured problem kernel size is lower than the upper bound of $3k^3 + 3k^2$ hyperedges that Sorge et al. [SMNW14] have proven using data reduction rules specifically designed for Golomb Subruler. Moreover, our problem kernel is computable in linear time, while the problem kernel of Sorge et al. [SMNW14] takes $O(k(n+m))$ time. Both problem kernels require the conflict hypergraph as input.

Summary. The calloc trie implementation of Algorithm 5.1 is superior when enough memory is available, since it is the fastest variant if the C++ environment at hand implements the allocation of zero-initialized memory using `calloc` efficiently. In all other cases, the balanced tree implementation of Algorithm 5.1 yields a good compromise between running time and memory usage.

One can observe that the data reduction effect on nearly uniform hypergraphs, like those from Golomb Subruler, is rather limited. On the other hand, problem kernels for d-Hitting Set can be small even for high values of d if the input hypergraph is less uniform.

5.5 Reducing the number of vertices in $O(k^{1.5d})$ additional time

This section combines the linear-time computable problem kernel from Section 5.3 with techniques of Abu-Khzam [Abu10] and Moser [Mos10, Section 7.3]. This will yield a problem kernel for d-Hitting Set with $O(k^d)$ hyperedges and $O(k^{d-1})$ vertices in $O(n+m+k^{1.5d})$ time. Towards this problem kernel, Section 5.5.1 first briefly sketches the running-time bottleneck of the kernelization idea of Abu-Khzam [Abu10], which is also a bottleneck in the algorithm of Moser [Mos10]. Then, Section 5.5.2 describes our improvements.

5.5.1 The approaches of Abu-Khzam and Moser

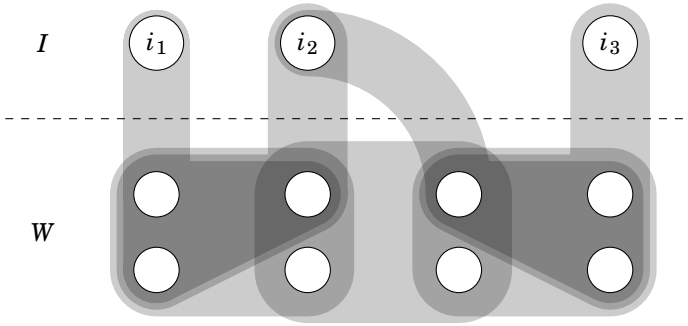
Abu-Khzam [Abu10] has shown a problem kernel for d-Hitting Set that comprises $O(k^{d-1})$ vertices. Moser [Mos10, Section 7.3] built upon the work of Abu-Khzam [Abu10] to show a problem kernel for d-Hitting Set that also comprises $O(k^{d-1})$ vertices but that, in contrast to the kernelization algorithm of Abu-Khzam [Abu10], yields a subgraph of the input hypergraph.

The approach of Abu-Khzam [Abu10] and Moser [Mos10] is as follows. Given a hypergraph H and a natural number k, Abu-Khzam [Abu10] first computes a maximal *weakly related* set W:

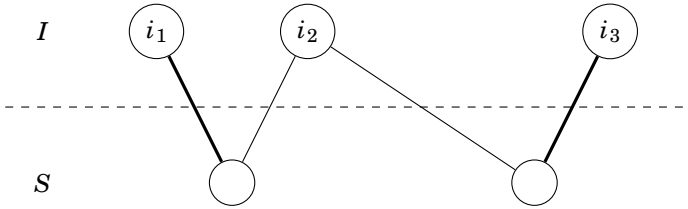
Definition 5.3 (Abu-Khzam [Abu10])**.** A set W of hyperedges is *weakly related* if every pair of hyperedges in W intersects in at most $d-2$ vertices.

Whether a given hyperedge e can be added to a weakly related set W, Abu-Khzam [Abu10] checks in $O(d|W|)$ time. After adding a hyperedge e to W, he applies data reduction to W in $O(2^d|W|\log|W|)$ time that ensures $|W| \leq k^{d-1}$. Hence, since $|W|$ never exceeds k^{d-1}, Abu-Khzam [Abu10] can compute the maximal weakly related set W in $O(2^d \cdot k^{d-1} \cdot (d-1)\log k \cdot m)$ time.

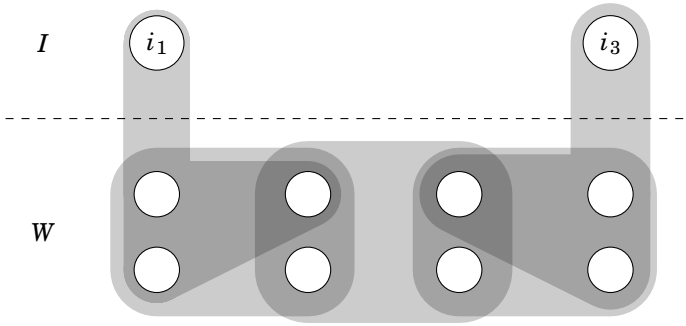
Since $|W| \leq k^{d-1}$, it remains to bound the size of the set I of vertices not contained in hyperedges of W. This is achieved by the following steps, which are illustrated in Figure 5.9. The set I is an *independent set*, that is, I contains no pair of vertices occurring in the same hyperedge [Abu10]. A bipartite graph $B = (I \uplus S, E')$ is constructed from the input hypergraph $H = (V, E)$, where $S := \{e \subseteq V \mid \exists v \in I : \exists w \in W : e \subseteq w, \{v\} \cup e \in E\}$ and $E' := \{\{v, e\} \mid v \in I, e \in S, \{v\} \cup e \in E\}$. Whereas Abu-Khzam [Abu10] shrinks the size of I using so-called *crown reductions*, Moser [Mos10, Lemma 7.16] shows that it is sufficient to compute a



(a) Input hypergraph. The hyperedges fully below the dashed line are a maximal weakly related set of hyperedges. The vertices above the dashed line are the independent set I.



(b) The resulting bipartite graph B with the thick edges being a maximum matching.



(c) The resulting hypergraph with the unmatched vertex i_2 and its incident hyperedges removed.

Figure 5.9: Illustration of the kernelization of Moser [Mos10, Lemma 7.16] using 4-Hitting Set.

maximum matching in B and to remove unmatched vertices in I together with the hyperedges containing them from the input hypergraph. The bound of the number of vertices in the problem kernel is thus $O(k^{d-1})$, since $|W| \leq k^{d-1}$, and, therefore, $|I| \leq |S| \leq d|W| \leq dk^{d-1}$.

5.5.2 Our improvements

We now discuss our running time improvements over the kernelization algorithms of Abu-Khzam [Abu10] and Moser [Mos10].

Given a hypergraph H and a natural number k, we first compute our problem kernel in $O(n+m)$ time, leaving $O(k^d)$ hyperedges in H. Afterward, we aim for applying the ideas of Abu-Khzam [Abu10] and Moser [Mos10] to reduce the number of vertices to $O(k^{d-1})$. However, as discussed in Section 5.5.1, the computation of a maximal weakly related set on our reduced instance already takes $O(2^d \cdot k^{d-1} \cdot (d-1)\log k \cdot m) = O(k^{2d-1}\log k)$ time. We improve the running time of this step in order to show the following theorem.

Theorem 5.2. *d-Hitting Set has a problem kernel with $d! \cdot d^{d+1} \cdot (k+1)^d$ hyperedges and $2 \cdot d! \cdot d^{d+1} \cdot (k+1)^{d-1}$ vertices computable in $O(d \cdot n + 2^d d \cdot m + (d! \cdot d^{d+1} \cdot (k+1)^d)^{1.5})$ time.*

Note that the problem kernel resulting from Theorem 5.2 will no longer be expressive in the sense of Section 5.2. For example, not every minimal hitting set of size at most k for the input hypergraph will be a minimal hitting set for the problem kernel. This can be observed in Figure 5.9, where the vertex i_2 might be contained in a minimal hitting set of the input hypergraph and is absent in the output hypergraph.

To prove Theorem 5.2, we compute a maximal weakly related set W in linear time and show that our problem kernel already ensures $|W| \in O(k^{d-1})$ and thus, that further data reduction on W is unnecessary. To compute a maximal weakly related set in linear time, we employ Algorithm 5.2.

After some initialization work in lines 1–5, Algorithm 5.2 in lines 6–11 adds a hyperedge e to the weakly related set W if none of the subsets $C \subseteq e$ with $|C| = d-1$ is a subset of a set previously added to W. The information whether C is some subset of a hyperedge previously added to W is saved in intersection[C]. Note that, in line 11, Algorithm 5.2 also sets "intersection[$e \setminus C$] $\leftarrow$ true" and thus saves which vertices are parts of hyperedges added to W. We will use this later to quickly reduce the number of vertices not contained in hyperedges in W.

Algorithm 5.2: Computation of a maximal weakly related set

Input: Hypergraph $H = (V, E)$, natural number k.
Output: Maximal weakly related set W.

```
1  W ← ∅
2  foreach e ∈ E do                                  // Initialization for each hyperedge
3      foreach C ⊆ e, |C| = d − 1 do
4          intersection[C] ← false                   // No hyperedges in W contain C yet.
5          intersection[e \ C] ← false               // The vertex in e \ C is not in W yet.
6  foreach e ∈ E do
7      if ∀C ⊆ e, |C| = d − 1 : intersection[C] = false then
8          W ← W ∪ {e}
9          foreach C ⊆ e, |C| = d − 1 do
10             intersection[C] ← true
11             intersection[e \ C] ← true
12 return W
```

Lemma 5.4. *Given a hypergraph H, a maximal weakly related set is computable in $O(d \cdot n + d^2 \cdot m)$ time.*

Proof. First, observe that the set W returned in line 12 of Algorithm 5.2 when applied to $H = (V, E)$ is indeed weakly related: let $w_1 \neq w_2 \in E$ intersect in more than $d-2$ vertices and assume that w_1 is added to W in line 8. Let $C := w_1 \cap w_2$. Obviously, $|C| = d-1$. Hence, when w_1 is added to W, we apply "intersection[C] ← true" in line 10. Therefore, when $e = w_2$ is considered in line 6, the condition in line 7 does not hold, which implies that w_2 is not added to W in line 8. In the same way it follows that each hyperedge is added to W if it does not intersect any hyperedge of W in more than $d-2$ vertices. Therefore, W is maximal.

Now, Algorithm 5.2 works as follows. We use Lemma 5.3 to look up values in intersection[] in $O(d)$ time. To this end, like in the proof of Proposition 5.3, we represent vertex subsets of size at most d as sorted sequences of length at most d. Thus, we first sort each hyperedge of H in $O(m \cdot d \log d)$ total time. To apply Lemma 5.3 to create the associative array intersection[], we need a list L of all values that we are going to store values for. As L, we use the list that, for each hyperedge e of H and each vertex $v \in e$, contains $e \setminus \{v\}$ and $\{v\}$. Of course, $e \setminus \{v\}$ can be computed in $O(d)$ time from e so that $e \setminus \{v\}$ is sorted. It follows that

L contains at most $2d \cdot m$ elements and is computable in $O(d^2 m)$ time. Hence, by Lemma 5.3, we can build the associative array intersection[] in $O(dn + d^2 \cdot m)$ time and looking up values in intersection[] works in $O(d)$ time for elements of L.

Now, the initialization in lines 1–5 works in $O(d^2 \cdot m)$ time. Finally, for every hyperedge, the body of the for-loop in line 6 can be executed in $O(d^2)$ time by doing $O(d)$-time look-ups for each of the $2d \cdot m$ sets. □

We can now prove Theorem 5.2 by showing how to compute a problem kernel with $O(k^{d-1})$ vertices in $O(n + m + k^{1.5d})$ time.

Proof of Theorem 5.2. It is shown in Theorem 5.1 that d-Hitting Set has a problem kernel with $d! \cdot d^{d+1} \cdot (k+1)^d$ hyperedges that is computable in $O(dn + 2^d d \cdot m)$ time. It remains to show that, in additional $O(d! \cdot d^{d+1} \cdot (k+1)^d)^{1.5}$ time, the number of vertices of a hypergraph H output by Algorithm 5.1 can be reduced to $2 \cdot d! \cdot d^{d+1} \cdot (k+1)^{d-1}$. To this end, we follow the approaches of Abu-Khzam [Abu10] and Moser [Mos10] as discussed in Section 5.5.1 and as illustrated in Figure 5.9.

First, we compute a maximal weakly related set W in H in $O(d \cdot n + d^2 \cdot m)$ time using Algorithm 5.2. We show that $|W| \leq d! \cdot d^d \cdot (k+1)^{d-1}$. To this end, consider the hypergraph $H_\ell := (V, W_\ell)$ for $1 \leq \ell \leq d$, where W_ℓ is the set of cardinality-ℓ hyperedges in W. Since H has been output by Algorithm 5.1, we know that, by Lemma 5.1, H_ℓ has no sunflowers with more than $d(k+1)$ petals. Moreover, since every pair of hyperedges in W intersects in at most $d-2$ vertices, also each pair of hyperedges of H_ℓ intersects in at most $d-2$ vertices. Hence, by Lemma 5.2 with $b = 2$ and $c = d(k+1)$, we know that H_ℓ for $\ell \geq 2$ has at most $\ell! d^{\ell-1}(k+1)^{\ell-1}$ hyperedges. Moreover, H_1 contains at most $d(k+1)$ hyperedges, as they form a sunflower with empty core. Therefore, $|W| \leq d! \cdot d^d \cdot (k+1)^{d-1}$.

Next, we construct a bipartite graph $B = (I \uplus S, E')$ from the input hypergraph $H = (V, E)$, where

i) I is the set of vertices in V not contained in any hyperedge in W, which is an independent set [Abu10],

ii) $S := \{e \subseteq V \mid \exists v \in I \colon \exists w \in W \colon e \subseteq w, \{v\} \cup e \in E\}$, and

iii) $E' := \{\{v, e\} \mid v \in I, e \in S, \{v\} \cup e \in E\}$.

This can be done in $O(d^2 \cdot m)$ time by exploiting the information stored in the associative array intersection[] computed by Algorithm 5.2: for each $e \in E$ with $|e| = d$ and each $v \in e$, add $\{v, e \setminus \{v\}\}$ to the graph B if and only if intersection[$e \setminus$

$\{v\}]$ = true and intersection$[\{v\}]$ = false. In this case, it follows that e can be partitioned into

i) a subset $e \setminus \{v\}$ of a hyperedge of W, since intersection$[e \setminus \{v\}]$ = true, and

ii) the vertex v, which is not contained in any hyperedge in W, since we have intersection$[\{v\}]$ = false, and, hence, is contained in I.

Thus, e clearly satisfies the definition of E'. Observe that the graph B constructed in this way contains at most $|E| = m$ edges. It remains to shrink I so that it contains at most $|S|$ vertices. Then, the number of vertices in the output hypergraph will be at most $d|W| + |I| \leq d|W| + |S| \leq 2d|W| = 2 \cdot d! \cdot d^{d+1} \cdot (k+1)^{d-1}$. This, as shown by Moser [Mos10, Section 7.3], is achieved by computing a maximum matching in B and deleting from H the unmatched vertices in I and the hyperedges containing them. To analyze the running time of computing the maximum matching, recall that the number of edges in B is at most $m \leq d! \cdot d^{d+1} \cdot (k+1)^d$ and that the number $|I| + |S|$ of vertices is at most twice as much. Hence, a maximum matching in B can be computed in $O(\sqrt{|I \uplus S|} \cdot |E'|) = O(d! \cdot d^{d+1} \cdot (k+1)^d)^{1.5}$ time using the algorithm of Hopcroft and Karp [Sch03, Theorem 16.4]. □

5.6 Conclusion

We have given an understanding of expressive kernelization for d-Hitting Set and have shown, as earlier claimed by Niedermeier and Rossmanith [NR03], that a problem kernel for d-Hitting Set with $O(k^d)$ hyperedges and vertices can be computed in linear time. Using the linear-time computable problem kernel for d-Hitting Set, we have improved the worst-case running times of the $O(k^{d-1})$-vertex problem kernels by Abu-Khzam [Abu10] and Moser [Mos10].

Our experiments have shown that the kernelization algorithm runs efficiently, yet the observed data reduction effect on the nearly uniform hypergraphs occurring in the construction of Golomb rulers was limited.

An interesting question is whether a problem kernel with $O(k^{d-1})$ vertices and $O(k^d)$ hyperedges for d-Hitting Set can be computed in linear time. Answering this question would merge the best known results for problem kernels for d-Hitting Set. However, to date, all $O(k^{d-1})$-vertex problem kernels for d-Hitting Set that we are aware of, that is, the problem kernels by Abu-Khzam [Abu10] and Moser [Mos10], involve the computation of maximum matchings. This seems to be difficult to avoid this bottleneck.

6 Hypergraph Cutwidth and a general method

In Chapters 3–5, we developed a variety of algorithms for *specific* graph and hypergraph problems. In contrast, in Section 2.5, we have seen that there are *general* tools for obtaining fixed-parameter linear-time algorithms for many graph problems parameterized by treewidth: this tool is monadic second-order logic. As sketched in Section 2.5.2, it is based on transforming constant-size monadic second-order logic formulas into tree automata that decide graph properties by "looking" at terms representing tree decompositions of graphs.

In this chapter, we provide a necessary and sufficient condition for the recognizability of a *hypergraph* property by tree automata "looking" at terms representing tree decompositions of incidence graphs.

Using this, we show an algorithm for testing whether a hypergraph has cutwidth at most k that runs in linear time for constant k. Moreover, using the provided tools, one can show that having constant hypertree width is not expressible in monadic second-order logic.

6.1 Introduction

In Section 2.5.1, we have seen a general approach to derive fixed-parameter linear-time algorithms for a large class of graph problems parameterized by treewidth: express a graph property in monadic second-order logic of graphs. The constant-size monadic second-order logic formula can be turned into a linear-time algorithm for deciding the graph property on graphs of constant treewidth.

In some cases, graph problems do not easily give in to this standard technique. In this case, another technique helps finding linear-time algorithms on graphs of constant treewidth or to prove the inapplicability of the monadic second-order logic approach [AF93; BFW92; GNNW13]: similarly to how regular languages can be recognized by finite automata, some graph problems on graphs of constant treewidth can be recognized in linear time by tree automata [DF13, Section 12.7].

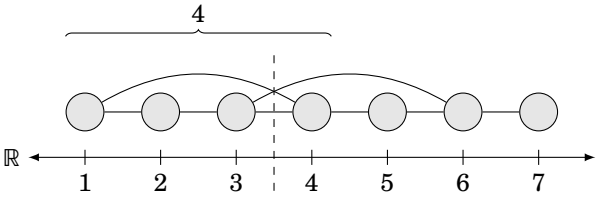


Figure 6.1: A graph with vertices at integer positions between one and seven. The graph's cutwidth is at most three: there are three edges with vertices placed to the left as well as to the right of the dashed line. Moreover, it has bandwidth at most four: one of the graph's "longest" edges joins the vertex at position one with the vertex at position four.

In fact, as explained in Section 2.5.2, the monadic second-order logic approach is based on transforming an expression in monadic second-order logic into a tree automaton. Disproving the existence of a tree automaton for a problem therefore shows that it not expressible in monadic second-order logic. Note that this statement is not tied to any complexity-theoretical assumptions like $\mathrm{P} \neq \mathrm{NP}$ or $\mathrm{FPT} \neq \mathrm{W}[1]$.

A sufficient and necessary condition for the decidability of a graph problem by tree automata has been given by an adaption of the Myhill-Nerode theorem from formal language theory to graphs [DF13, Section 12.7], which helped gaining insights into the following graph problems:

Cutwidth. A graph has *cutwidth* k if there is a layout of its vertices on the real line such that, for any $i \in \mathbb{R}$, there are at most k edges joining a vertex at position less than i with a vertex at position greater than i. This is illustrated in Figure 6.1.

It can be shown that, for every constant k, there *exists* a constant-size formula in monadic second-order logic that a graph satisfies if and only if it has cutwidth at most k [DF13, Example 19.1.1]. The proof, however, is non-constructive and we are not aware of any method for obtaining the actual formula for a given k. The difficulty is expressing the search for a vertex layout in terms of a search for vertex subsets and edge subsets, which are the only tools provided by monadic second-order logic.

Using the Myhill-Nerode theorem for graphs, Abrahamson and Fellows [AF93] have shown that testing a graph for constant cutwidth k can be done in linear time. Thilikos, Serna, and Bodlaender [TSB05] later gave a dynamic programming algorithm that is more technical, but has the advantage of constructing the sought vertex layout instead of only answering whether it exists.

Bandwidth. A graph has *bandwidth* k if each vertex v can be mapped to an integer position $p(v)$ such that each edge $\{v,w\}$ satisfies $|p(v)-p(w)| \leq k$. This is illustrated in Figure 6.1.

Again, the difficulty in solving this problem in linear time on graphs of constant treewidth using monadic second-order logic is expressing the search for a vertex layout in terms of the search for vertex and edge subsets.

Indeed, the problem is NP-hard even on trees [GGJK78]. Thus, unless P = NP, the problem is not linear-time solvable on graphs of constant treewidth and, consequently, the property of having constant bandwidth is not expressible in constant-size monadic second-order logic formulas unless P = NP.

Using the Myhill-Nerode theorem for graphs, Abrahamson and Fellows [AF93] have shown that that the property is not expressible using constant-size monadic second-order logic formulas, independently of whether P ≠ NP.

This chapter extends the Myhill-Nerode theorem for graphs to hypergraphs. In this way, we provide a method to derive fixed-parameter linear-time algorithms for hypergraph problems parameterized by *incidence treewidth* (the treewidth of the incidence graph) or to prove that hypergraph properties are not expressible in monadic second-order logic.

From the point of view of parameterized complexity, incidence treewidth is an interesting hypergraph parameter, since it is bounded from above by the commonly used treewidth generalizations to hypergraphs and can be arbitrarily smaller [KV00; SS10].

Applying our Myhill-Nerode theorem for hypergraphs, we obtain a fixed-parameter linear-time algorithm for HYPERGRAPH CUTWIDTH. Moreover, using the Myhill-Nerode theorem for hypergraphs, one can show that having constant generalized hypertree width is not expressible in monadic second-order logic. Herein, generalized hypertree width is another generalization of treewidth to hypergraphs.

6.1.1 Known results

This section summarizes other known generalizations of the Myhill-Nerode theorem and known results regarding the problems HYPERGRAPH CUTWIDTH and GENERALIZED HYPERTREE WIDTH.

Generalizations of the Myhill-Nerode theorem. The Myhill-Nerode theorem as a sufficient and necessary condition for the decidability of a formal language using finite automata is due to Myhill [Myh57] and Nerode [Ner58]. Since then, analogs of the Myhill-Nerode theorem have been provided for graphs of constant treewidth [AF93], matroids of constant branchwidth [Hli06], graphs of constant rankwidth [GH10], and edge- and vertex-colored graphs of constant treewidth and cliquewidth [CE12, Sections 4.2.2 and 4.4.2].

Hypergraph Cutwidth. HYPERGRAPH CUTWIDTH is the problem of positioning the vertices of a hypergraph on the real line so that, for any $i \in \mathbb{R}$, there are at most k hyperedges containing vertices with positions less than as well as greater than i. As a natural generalization of CUTWIDTH on graphs, HYPERGRAPH CUTWIDTH is NP-complete [Gav77]. In the context of VLSI design, HYPERGRAPH CUTWIDTH is known as BOARD PERMUTATION [MS91]. Moreover, HYPERGRAPH CUTWIDTH naturally arises in solving CNF-SAT in the context of automatically testing digital hardware [PCK99; WCZK01].

For the special case of CUTWIDTH on graphs, several fixed-parameter algorithms are known [AF93; BFT09; Cyg+14; FL92; FL94; TSB05]. Miller and Sudborough [MS91] designed an algorithm for HYPERGRAPH CUTWIDTH running in $O(n^{k^2+3k+3})$ time. Nagamochi [Nag12] presented a framework for solving cutwidth-related graph problems in $n^{O(k)}$ time.

Generalized Hypertree Width. The task in the GENERALIZED HYPERTREE WIDTH problem is checking whether a hypergraph has generalized hypertree width k. Generalized hypertree width is a generalization of treewidth to hypergraphs. It is known that GENERALIZED HYPERTREE WIDTH remains NP-hard even for $k = 3$ [GMS09]. Hence, unless P = NP, the computation of this width parameter is not fixed-parameter tractable parameterized by k.

6.1.2 Our results

We prove a Myhill-Nerode theorem for hypergraphs having constant incidence treewidth. We exploit it to show a fixed-parameter linear-time algorithm for HYPERGRAPH CUTWIDTH parameterized by k. Moreover, our Myhill-Nerode theorem can be used to show that the property of having constant generalized hypertree width is not expressible in monadic second-order logic.

6.1.3 Chapter outline

First, Section 6.2 presents our Myhill-Nerode theorem for hypergraphs of constant incidence treewidth.

Then, Section 6.3 discusses how the Myhill-Nerode theorem for hypergraphs yields linear-time algorithms and excludes the possibility for monadic second-order logic expressions for hypergraph problems.

Section 6.4 presents our proof for HYPERGRAPH CUTWIDTH being fixed-parameter linear parameterized by k.

Finally, Section 6.5 sketches how the Myhill-Nerode theorem for hypergraphs can be exploited to show that the property of having constant generalized hypertree width is not expressible in monadic second-order logic.

6.2 A Myhill-Nerode theorem for hypergraphs

The theorem of Myhill [Myh57] and Nerode [Ner58] is a necessary and sufficient condition for a formal language being recognizable by finite automata. The aim of this section is generalizing it to a necessary and sufficient condition for the recognizability of a hypergraph property by *tree* automata.

To this end, we first briefly recall the original Myhill-Nerode theorem for regular languages in Section 6.2.1.

Then, Section 6.2.2 states a Myhill-Nerode theorem for vertex-colored graphs. We will exploit vertex-colored graphs to uniquely represent hypergraphs as incidence graphs with two vertex colors: the vertices of one color in the incidence graph will represent hypergraph vertices whereas the vertices of the other color will represent hyperedges.

Based on the Myhill-Nerode theorem for colored graphs, Section 6.2.3 proves the Myhill-Nerode theorem for hypergraphs.

6.2.1 Regular languages

The Myhill-Nerode theorem is a tool for proving or disproving that a formal language is *regular*, that is, decidable by a finite automaton. The theorem states that a language is regular if and only if its so-called *canonical right congruence* has a finite number of equivalence classes.

Definition 6.1. Let $L \subseteq \Sigma^*$ be a language. The *canonical right congruence* $\sim_L$ is defined as follows: for $v, w \in \Sigma^*, v \sim_L w :\iff \forall x \in \Sigma^* : vx \in L \iff wx \in L$, where vx is the concatenation of v and x.

Example 6.1. Consider the language $L := \{a^i b^j \mid i, j \in \mathbb{N}\} \subseteq \{a, b\}^*$ consisting of words starting with an arbitrary number of a's and ending in an arbitrary number of b's. Then, $a \sim_L aa$. However, $a \not\sim_L ab$, since, for example, $aa \in L$ but $aba \notin L$.

Obviously, for a language $L \subseteq \Sigma^*$, the canonical right congruence $\sim_L$ is an equivalence relation.

Theorem 6.1 (Myhill [Myh57] and Nerode [Ner58])**.** *A language $L \subseteq \Sigma^*$ is recognizable by a finite automaton if and only if the canonical right congruence $\sim_L$ has finite index.*

Thus, the Myhill-Nerode theorem gives a necessary and sufficient condition for a language being recognizable by a finite automaton.

6.2.2 Colored graphs

Our proof of the Myhill-Nerode theorem for hypergraphs is based on two main ingredients. The first ingredient is a Myhill-Nerode theorem for colored graphs, that is, a necessary and sufficient condition for the recognizability of a property of colored graphs by tree automata. The second ingredient is the fact that every hypergraph can be uniquely represented by an incidence graph with two vertex colors: the vertices of one color in the incidence graph represent hypergraph vertices whereas the vertices of the other color represent hyperedges.

This section provides the first ingredient: Courcelle and Engelfriet [CE12, Section 4.2.2] have shown a Myhill-Nerode theorem for vertex- as well es edge-colored graphs. We will state here only the vertex-colored version for the purpose of introducing the concepts necessary towards our Myhill-Nerode theorem for hypergraphs.

In order to apply tree automata to decide a problem on colored graphs, we follow the approach illustrated in Figure 6.2: we first show how every graph of

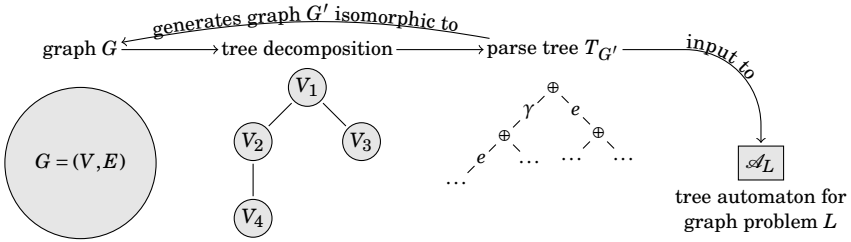


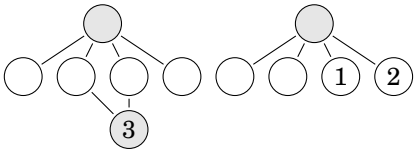
Figure 6.2: Solving a graph problem L using a tree automaton: from a graph G with constant treewidth, a minimum-width tree decomposition can be computed in linear time [Bod96]. The tree decomposition can be turned into a size-$O(n)$ expression over a set $\{\emptyset, e, u, \gamma, i, \oplus\}$ of operators in linear time such that the value of the expression is a graph G' isomorphic to G [DF13, Theorem 12.7.1]. The parse tree $T_{G'}$ of the expression is fed into a tree automaton $\mathscr{A}_L$ that accepts $T_{G'}$ in $O(n)$ time if and only if $G' \in L$. The existence of $\mathscr{A}_L$ for the problem L can be proven or disproved by the Myhill-Nerode theorem for graphs.

constant treewidth can be represented by an expression over a constant-size set of operators. Then, we feed the parse tree of this expression into a tree automaton similarly to how we used a tree automaton to recognize the set of all true Boolean expressions in Section 2.5.2.

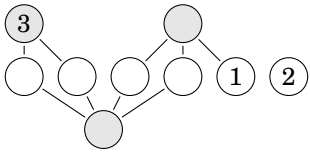
Our central operator for generating colored graphs corresponds to the operator for concatenating words in the language setting of the Myhill-Nerode theorem: every word with more than one letter is the concatenation of shorter words and we will see that every graph of treewidth $t-1$ with more than t vertices is isomorphic to the result of *gluing* smaller graphs together at a *boundary* consisting of t *labeled* vertices. This is formalized by the following definition.

Definition 6.2. The following concepts are illustrated in Figure 6.3.

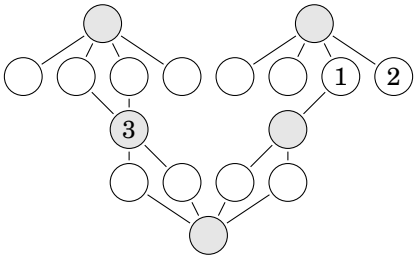
A *c-colored graph*	is a graph whose vertices have *colors* in $[c]$.
A *t-boundaried graph*	is a graph with t distinguished vertices that have pairwise distinct *labels* in $[t]$.
A *boundary vertex*	of a t-boundaried graph is a vertex with a label.
The *boundary* $\partial(G)$	of a t-boundaried graph G is the set of its t boundary vertices.



(a) A 2-colored 3-boundaried graph G.



(b) A 2-colored 3-boundaried graph H.



(c) The glued graph $G \oplus_c H$.

Figure 6.3: Two color-compatible 2-colored 3-boundaried graphs G and H and their glued graph. White vertices have color 1, gray vertices have color 2, and vertex labels are represented by numbers. For each of the shown graphs, the boundary is the set of vertices labeled 1, 2, and 3.

Let G_1 and G_2 be c-colored t-boundaried graphs. Then, G_1 and G_2 are

isomorphic, $G_1 \cong G_2$, if there is an isomorphism between the underlying (uncolored and unlabeled) graphs mapping each vertex to a vertex with the same color *but ignoring labels*, and

color-compatible, if the vertices with the same labels in $\partial(G_1)$ and $\partial(G_2)$ have the same color.

Finally, $G_1 \oplus_c G_2$ for two color-compatible c-colored t-boundaried graphs G_1 and G_2 is the c-colored graph obtained by *gluing* G_1 and G_2, that is, by taking the disjoint union of G_1 and G_2 and identifying the vertices of $\partial(G_1)$ and $\partial(G_2)$ having the same label; the vertex colors of $G_1 \oplus_c G_2$ are inherited from G_1 and G_2. For two color-incompatible graphs G_1 and G_2, we leave $\oplus_c$ undefined.

It is important to note the difference between *colors* and *labels*: all vertices may have *colors*. In contrast, only the t boundary vertices have *labels*, which are required to be pairwise distinct. *Colors* are just vertex annotations that we will use to represent hypergraphs uniquely using 2-colored incidence graphs: the vertices of one color in the incidence graph will represent hypergraph vertices, the vertices of the other color will represent hyperedges. In contrast, *labels* determine on which vertices of a graph our gluing operator $\oplus_c$ and the operators defined below will act.

Additionally to $\oplus_c$, we use a set of operators to create primitive graphs and to arbitrarily permutate the labels of the boundary vertices. Using these primitive graphs and the operator $\oplus_c$, we will be able to represent any c-colored graph of constant treewidth as the value of an expression over a constant number of operators and, hence, as a parse tree that can be fed into a tree automaton.

Definition 6.3. The *size-t parsing operators* for c-colored t-boundaried graphs are defined as follows and illustrated in Figure 6.4:

$\emptyset_{n_1,\dots,n_c}$ for $\sum_{i=1}^{c} n_i = t$ is a family of nullary operators, each of which creates a graph consisting of isolated boundary vertices, of which the first n_1 vertices get color 1, the next n_2 vertices get color 2, and so on.

e is a unary operator that adds an edge between the boundary vertices labeled 1 and 2.

u_ℓ for $1 \le \ell \le c$ is a family of unary operators, each of which adds a new boundary vertex of color ℓ and labels it 1, unlabeling the vertex previously labeled 1.

γ is a unary operator that cyclically shifts the boundary. That is, γ moves each label j to the vertex with label $j+1 \pmod t$.

i is a unary operator that assigns label 1 to the vertex currently labeled 2 and label 2 to the vertex currently labeled 1.

$\oplus_c$ is the gluing operator from Definition 6.2.

Note that the operators γ and i allow to arbitrarily permutate the labels of *all* t boundary vertices, since any permutation can be represented as a series of transpositions and cyclical shifts. Thus, the operator e effectively allows us to add edges between arbitrary boundary vertices and the operator u_ℓ can effectively add a new boundary vertex of any label.

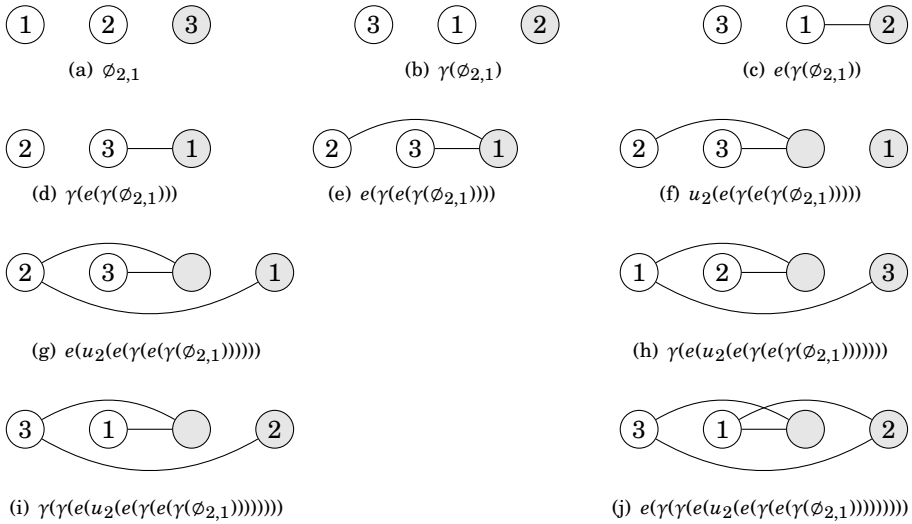


Figure 6.4: The value of the expression $e(\gamma(\gamma(e(u_2(e(\gamma(e(\gamma(\varnothing_{2,1}))))))))))$ over the size-3 parse operators for 2-colored graphs is a cycle on four vertices with alternating colors. Vertex labels are represented as numbers written into the vertices. White vertices have color 1, gray vertices have color 2.

Our aim is to represent each graph of constant treewidth as an expression over the operators in Definition 6.3, so that its parse tree can be fed into a tree automaton. In Figure 6.4, we have already seen that a cycle, which is a graph of treewidth two, can be generated using the size-3 parsing operators. This observation generalizes to all graphs of constant treewidth: Downey and Fellows [DF13, Theorem 12.7.1] have shown the following theorem for graphs with $c = 1$ colors, which can in full analogy be proven for any constant $c > 1$ [BFGR14].

Theorem 6.2. *Let G be a c-colored graph with constant treewidth $t-1$ and at least t vertices.*

Then, in linear time, G can be transformed into an expression over the size-t operators in Definition 6.3 whose value is a graph H that is isomorphic to G, that is, when ignoring the labels of H.

We are now at a point where we can transform each graph of constant treewidth into a representation that we can feed into a tree automaton: we feed it the parse tree of an expression over the operators in Definition 6.3. A central question remains: which graph properties can be recognized by tree automata operating on such parse trees? The Myhill-Nerode theorem for colored graphs will give a sufficient and necessary condition. To state the theorem, we first lift the concept of a canonical right congruence from the language setting (Definition 6.1) to graphs.

Definition 6.4. An example for the following concepts will be given in Example 6.2 below. We denote by

$\mathbb{U}_{t,c}^{\text{large}}$ the *large universe* of all c-colored t-boundaried graphs and by

$\mathbb{U}_{t,c}^{\text{small}} \subseteq \mathbb{U}_{t,c}^{\text{large}}$ the *small universe* of c-colored t-boundaried graphs that are the values of expressions over the size-t operators in Definition 6.3. Importantly, $\mathbb{U}_{t,c}^{\text{small}}$ is not necessarily closed under isomorphism.

$F \subseteq U$ for $U \in \{\mathbb{U}_{t,c}^{\text{small}}, \mathbb{U}_{t,c}^{\text{large}}\}$ is a *graph problem* if, for all $G \in F$ and $H \in U$ with $G \cong H$, we also have $H \in F$. That is, in particular, changing vertex labels does not influence membership in F.

Finally, for a graph problem $F \subseteq U$, where $U \in \{\mathbb{U}_{t,c}^{\text{small}}, \mathbb{U}_{t,c}^{\text{large}}\}$, we define the *canonical right congruence* $\sim_F$ *over* U for F as follows:

$G_1 \sim_F G_2$ for two graphs $G_1, G_2 \in U$ holds if and only if G_1 and G_2 are color-compatible and if, for all $H \in U$ color-compatible with G_1 and G_2, we have $G_1 \oplus_c H \in F \iff G_2 \oplus_c H \in F$.

Observe that, by Theorem 6.2, for each c-colored t-boundaried graph $G \in \mathbb{U}_{t,c}^{\text{large}}$ with at least t vertices and treewidth $t-1$, there is an isomorphic graph H in $\mathbb{U}_{t,c}^{\text{small}}$.

Example 6.2. We show an example for a graph problem $F \subseteq \mathbb{U}_{1,2}^{\text{small}}$ whose canonical right congruence $\sim_F$ has infinite index over $\mathbb{U}_{1,2}^{\text{small}}$. Herein, recall that $\mathbb{U}_{1,2}^{\text{small}}$ is the set of 2-colored 1-boundaried graphs that are values of the expressions over the size-1 operators in Definition 6.3. Since graphs in $\mathbb{U}_{1,2}^{\text{small}}$ have only one boundary vertex, the operator e in Definition 6.3 cannot be used to add edges. Thus, the graphs in $\mathbb{U}_{1,2}^{\text{small}}$ consist only of isolated vertices.

Consider the graph problem $F := \{G \in \mathbb{U}_{1,2}^{\text{small}} \mid G \text{ has an equal number of vertices with color 1 and with color 2}\}$. We show that the canonical right congruence $\sim_F$ has infinite index over $\mathbb{U}_{1,2}^{\text{small}}$. To this end, consider the following graphs:

Let $G^1_\ell \in \mathbb{U}^{\text{small}}_{1,2}$ be a graph that has ℓ isolated vertices of color 1 and additionally one isolated boundary vertex of color 2 and label 1. To see that such a graph in $\mathbb{U}^{\text{small}}_{1,2}$ indeed exists, observe that the expression $u_2(u_1(u_1(\ldots(\varnothing_{1,0})\ldots)))$ generates such a graph.

Let $G^2_\ell \in \mathbb{U}^{\text{small}}_{1,2}$ be a graph that has ℓ isolated vertices of color 2, among them being one boundary vertex with color 2 and label 1. Such a graph in $\mathbb{U}^{\text{small}}_{1,2}$ exists, since it can be generated by the expression $u_2(u_2(u_2(\ldots(\varnothing_{0,1})\ldots)))$.

Observe that, for any $i,j \in \mathbb{N}$, the graphs G^1_i and G^2_j are color-compatible: in both graphs, the only boundary vertex has color 2. Moreover, since $G^1_i, G^2_j \in \mathbb{U}^{\text{small}}_{1,2}$ the definition of $\mathbb{U}^{\text{small}}_{1,2}$ implies that also $G^1_i \oplus_c G^2_j \in \mathbb{U}^{\text{small}}_{1,2}$. Now, observe that $G^1_i \oplus_c G^2_j$ has i vertices of color 1 and j vertices of color 2. It follows that, for $i \neq j$, there is a graph $H_{ij} := G^2_i \in \mathbb{U}^{\text{small}}_{1,2}$ such that $G^1_i \oplus_c H_{ij} \in F$ but $G^1_j \oplus_c H_{ij} \notin F$. Hence, for $i \neq j$, we have $G^1_i \not\sim_F G^1_j$ and, thus, F has infinite index over $\mathbb{U}^{\text{small}}_{1,2}$.

The following Myhill-Nerode theorem for colored graphs makes a statement about when a tree automaton can decide a graph problem $F \subseteq \mathbb{U}^{\text{small}}_{t,c}$. For example, it implies that the trivial graph problem F from Example 6.2 cannot be decided by tree automata. In contrast, tree automata can solve several NP-hard problems in linear time on graphs of constant treewidth—all problems expressible in monadic second-order logic. The obstacle in deciding F using tree automata is that tree automata only have a finite amount of memory and, thus, cannot count arbitrarily far.

Theorem 6.3. *Let $F \subseteq \mathbb{U}^{small}_{t,c}$ be a graph problem. The following statements are equivalent:*

i) The collection of parse trees generating the graphs in F is recognizable by a tree automaton.

ii) The canonical right congruence $\sim_F$ has finite index over $\mathbb{U}^{small}_{t,c}$.

Downey and Fellows [DF13, Theorem 12.7.2] proved Theorem 6.3 for uncolored graphs, that is, for $c = 1$. For $c > 1$, a proof was later given by Courcelle and Engelfriet [CE12, Theorems 3.62 and 4.34(2)].[1] In the following section, we will use Theorem 6.3 to prove a Myhill-Nerode theorem for hypergraphs.

[1] We follow the notation of Downey and Fellows [DF13; DF99], who first proved the Myhill-Nerode theorem for graphs. The notation used by Courcelle and Engelfriet [CE12] is different: for graph problems $F \subseteq \mathbb{U}^{\text{small}}_{t,c}$, our operator $\oplus_c$ corresponds to their operator $\square_t$ and our canonical right congruence $\sim_F$ corresponds to their equivalence relation $\equiv_F$.

6.2.3 Hypergraphs

In this section, we show how tree automata can be used to recognize hypergraph properties. In the form of a Myhill-Nerode theorem for hypergraphs, we provide a necessary and sufficient condition for hypergraph properties to be recognizable by tree automata. Herein, in the remainder of this chapter, we shall understand the term "hypergraph" in the broadest sense: we allow hypergraphs to contain hyperedges multiple times as well as to contain empty hyperedges.

Like in the setting of formal languages in Section 6.2.1 or of colored graphs in Section 6.2.2, our Myhill-Nerode theorem for hypergraphs will be based on the canonical right congruence of some hypergraph problem having finite index. Naturally, we will define canonical right congruences in terms of hypergraphs that can be glued together from smaller hypergraphs. To this end, we first define the notion of gluing for hypergraphs.

Definition 6.5. The following concepts are illustrated in Figure 6.5.

A *t-boundaried hypergraph* is a hypergraph with t distinguished vertices and hyperedges that have pairwise distinct *labels* in $[t]$.

A *boundary object* of a t-boundaried hypergraph is a labeled hyperedge or vertex.

The *boundary* $\partial(H)$ of a t-boundaried hypergraph H is the set of its boundary objects.

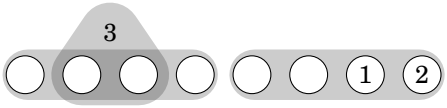
Two t-boundaried hypergraphs H_1 and H_2 are

gluable if for each label $\ell \in [t]$, the boundary object with label ℓ is a vertex in H_1 if and only if it is a vertex in H_2.

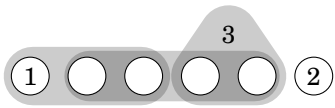
$H_1 \oplus_{\mathrm{h}} H_2$ for two gluable t-boundaried hypergraphs H_1 and H_2 is the t-boundaried hypergraph obtained by taking the disjoint union of H_1 and H_2, identifying each labeled vertex of H_1 with the vertex of H_2 with the same label, and replacing all hyperedges with the same label by their union.

Note that we use the operator $\oplus_{\mathrm{h}}$ to glue hypergraphs, whereas we use the operator $\oplus_{\mathrm{c}}$ to glue colored graphs.

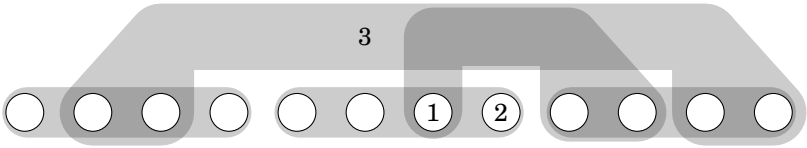
In order to apply tree automata to hypergraphs, in contrast to Section 6.2.2 for colored graphs, we will not define a set of operators for generating hypergraphs.



(a) A 3-boundaried hypergraph H_1.



(b) A 3-boundaried hypergraph H_2.



(c) The glued hypergraph $H_1 \oplus_{\mathrm{h}} H_2$.

Figure 6.5: Two hypergraphs G and H and the glued hypergraph $H_1 \oplus_{\mathrm{h}} H_2$. Vertex and hyperedge labels are represented by numbers.

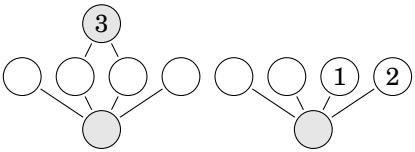
Instead, we will generate hypergraphs from 2-colored incidence graphs: vertices of one color will represent hypergraph vertices, whereas vertices of the other color will represent hyperedges. That is, instead of letting tree automata solve a hypergraph problem, we let them solve a graph problem on colored incidence graphs. The goal of the next definition is formalizing this and is illustrated in Figure 6.6.

Definition 6.6. A *t-boundaried hypergraph generator* is a 2-colored t-boundaried graph $G = (U \uplus W, E)$ such that all vertices in U have color 1, all vertices in W have color 2, and each of U and W forms an independent set.

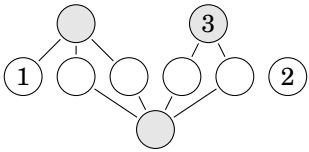
$\mathbb{H}(G)$ for a t-boundaried hypergraph generator $G = (U \uplus W, E)$ is the t-boundaried hypergraph with the vertex set U and the hyperedge set $\{N(w) \mid w \in W\}$. Moreover, each vertex of $\mathbb{H}(G)$ inherits its label from G and each hyperedge e in $\mathbb{H}(G)$ inherits its label from the vertex $w \in W$ of G that induced e.

$\mathbb{H}(F)$ for a set $F \subseteq \mathbb{U}_{t,2}^{\mathrm{small}}$ of t-boundaried hypergraph generators is the set $\bigcup_{G \in F} \{\mathbb{H}(G)\}$. Herein, recall that $\mathbb{U}_{t,2}^{\mathrm{small}}$ is the set of graphs that are the value of some expression over the size t operators in Definition 6.3. We say that F is *generator-total* if, for all t-boundaried hypergraph generators $G \in \mathbb{U}_{t,2}^{\mathrm{small}}$ with $\mathbb{H}(G) \in \mathbb{H}(F)$, we have $G \in F$.

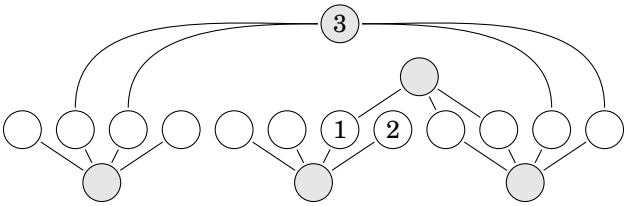
$\mathbb{H}_t^{\mathrm{large}}$ is the *large universe* of all t-boundaried hypergraphs.



(a) A 3-boundaried hypergraph generator G with $\mathbb{H}(G) = H_1$.



(b) A 3-boundaried hypergraph generator H with $\mathbb{H}(H) = H_2$.



(c) The glued graph $G \oplus_c H$ with $\mathbb{H}(G \oplus_c H) = H_1 \oplus_h H_2 = \mathbb{H}(G) \oplus_h \mathbb{H}(H)$.

Figure 6.6: Two 3-boundaried hypergraph generators G and H and their glued graph $G \oplus_c H$. White vertices have color 1, gray vertices have color 2, and vertex labels are represented as numbers. They generate the hypergraphs H_1, H_2, and $H_1 \oplus_h H_2$ from Figure 6.5.

$\mathbb{H}_t^{\text{small}}$ is the *small universe* of t-boundaried hypergraphs $H = \mathbb{H}(G)$ for some t-boundaried hypergraph generator $G \in \mathbb{U}_{t,2}^{\text{small}}$. That is, $\mathbb{H}_t^{\text{small}}$ is the set of hypergraphs that are generated by the graphs that are the values of expressions over the operators in Definition 6.3.

$F \subseteq U$ for $U \in \{\mathbb{H}_t^{\text{large}}, \mathbb{H}_t^{\text{small}}\}$ is a *hypergraph problem* if, for all $G \in F$ and $H \in U$ with $G \cong H$, we also have $H \in F$. That is, in particular, changing boundary labels does not influence membership in F.

The following observation will help us switching between hypergraph problems on graphs of constant incidence treewidth and their corresponding graph problems on graphs of constant treewidth. We will exploit them to lift the Myhill-Nerode theorem from the world of colored graphs into the world of hypergraphs.

Observation 6.1. The following observations follow from Definition 6.6.

i) A t-boundaried hypergraph generator G is isomorphic to the incidence graph of $\mathbb{H}(G)$. This is illustrated in Figures 6.5 and 6.6. Therefore, the treewidth of G equals the incidence treewidth of $\mathbb{H}(G)$.

ii) For two t-boundaried hypergraph generators $G, H \in \mathbb{U}_{t,2}^{\text{large}}$, we have $\mathbb{H}(G) \oplus_{\text{h}} \mathbb{H}(H) = \mathbb{H}(G \oplus_{\text{c}} H)$. This is also illustrated in Figures 6.5 and 6.6.

iii) For a generator-total $F \subseteq \mathbb{U}_{t,2}^{\text{small}}$ and each t-boundaried hypergraph generator $G \in \mathbb{U}_{t,2}^{\text{small}}$, we have $G \in F$ if and only if $\mathbb{H}(G) \in \mathbb{H}(F)$.[2]

iv) For every t-boundaried hypergraph $H \in \mathbb{H}_t^{\text{small}}$, by definition of $\mathbb{H}_t^{\text{small}}$, there is a t-boundaried graph $G \in \mathbb{U}_{t,2}^{\text{small}}$ such that $\mathbb{H}(G) = H$. Consequently, for every hypergraph problem $\mathscr{F} \subseteq \mathbb{H}_t^{\text{small}}$, there is a generator-total $F \subseteq \mathbb{U}_{t,2}^{\text{small}}$ with $\mathbb{H}(F) = \mathscr{F}$. Moreover, in terms of Definition 6.4, F is a graph problem.

Having defined the notion of gluing hypergraphs, we can now define canonical right congruences for hypergraphs. By showing whether the canonical right congruence for a hypergraph problem has finite index, we will then be able to show whether the problem is solvable by tree automata.

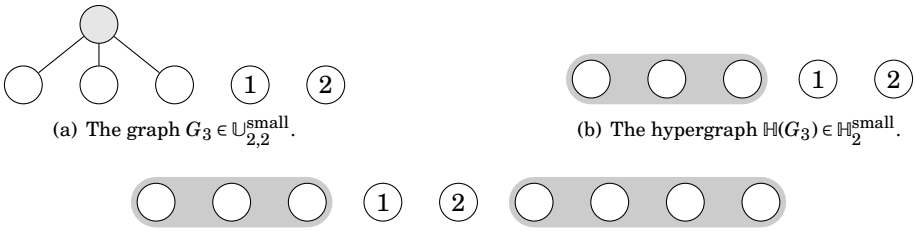
Definition 6.7. Let $\mathscr{F} \subseteq U$ for $U \in \{\mathbb{H}_t^{\text{large}}, \mathbb{H}_t^{\text{small}}\}$ be a hypergraph problem. We define the *canonical right congruence* $\sim_{\mathscr{F}}$ *over* U for $\mathscr{F}$ as follows:

$G_1 \sim_{\mathscr{F}} G_2$ for two t-boundaried hypergraphs $G_1, G_2 \in U$ holds if and only if G_1 and G_2 are gluable and for all $H \in U$ that are gluable to G_1 and G_2, $G_1 \oplus_{\text{h}} H \in \mathscr{F} \iff G_2 \oplus_{\text{h}} H \in \mathscr{F}$.

Example 6.3. We show a hypergraph problem $\mathscr{F} \subseteq \mathbb{H}_2^{\text{small}}$ for which the canonical right congruence $\sim_{\mathscr{F}}$ has infinite index. Herein, recall that $\mathbb{H}_2^{\text{small}}$ is the set of 2-boundaried hypergraphs generated by 2-boundaried hypergraph generators $G \in \mathbb{U}_{t,2}^{\text{small}}$ that, in turn, are the values of expressions over the size-2 operators in Definition 6.3.

Let $\mathscr{F} := \{H \in \mathbb{H}_2^{\text{small}} \mid \text{there is a } d \in \mathbb{N} \text{ such that } H \text{ is } d\text{-uniform}\}$, where a hypergraph is d-uniform if all of its hyperedges have cardinality d. To show that $\sim_{\mathscr{F}}$ has infinite index, consider the following family of t-boundaried hypergraph generators, of which G_3 is illustrated in Figure 6.7.

[2] If F is not generator-total, it might be that $G \notin F$ but $\mathbb{H}(G) \in \mathbb{H}(F)$ because $H \in F$ for some t-boundaried hypergraph generator $H \neq G$ with $\mathbb{H}(G) = \mathbb{H}(H)$: the graphs G and H might represent the hyperedges of $\mathbb{H}(G) = \mathbb{H}(H)$ using different mathematical objects.



(a) The graph $G_3 \in \mathbb{U}_{2,2}^{\text{small}}$.

(b) The hypergraph $\mathbb{H}(G_3) \in \mathbb{H}_2^{\text{small}}$.

(c) The hypergraph $\mathbb{H}(G_3) \oplus_{\text{h}} \mathbb{H}(G_4) \in \mathbb{H}_2^{\text{small}}$.

Figure 6.7: The graph $G_3 := u_1(i(u_1(e(u_1(e(u_1(e(\varnothing_{1,1})))))))) \in \mathbb{U}_{2,2}^{\text{small}}$, the hypergraph $\mathbb{H}(G_3) \in \mathbb{H}_2^{\text{small}}$, and $\mathbb{H}(G_3) \oplus_{\text{h}} \mathbb{H}(G_4)$. White vertices have color 1, gray vertices have color 2, and vertex labels are represented by numbers.

Let $G_i \in \mathbb{U}_{2,2}^{\text{small}}$ be a graph that has one vertex of color 2 adjacent to i vertices of color 1 and, additionally, two isolated boundary vertices of color 1. To see that such a graph in $\mathbb{U}_{2,2}^{\text{small}}$ indeed exists, observe that the expression $u_1(i(u_1(e(\ldots(u_1(e(u_1(e(\varnothing_{1,1})))))\ldots))))$ generates such a graph.

For $i \in \mathbb{N}$, the graph G_i is a 2-boundaried hypergraph generator in $\mathbb{U}_{2,2}^{\text{small}}$. Thus, $\mathbb{H}(G_i) \in \mathbb{H}_t^{\text{small}}$. For $i \neq j$, we have $G_i \in \mathbb{U}_{2,2}^{\text{small}}$ and $G_j \in \mathbb{U}_{2,2}^{\text{small}}$ and, hence, $G_i \oplus_{\text{c}} G_j \in \mathbb{U}_{2,2}^{\text{small}}$. Consequently, $\mathbb{H}(G_i) \oplus_{\text{h}} \mathbb{H}(G_j) = \mathbb{H}(G_i \oplus_{\text{c}} G_j) \in \mathbb{H}_2^{\text{small}}$.

Now, observe that, for $i, j \in \mathbb{N}$, the hypergraph $\mathbb{H}(G_i) \oplus_{\text{h}} \mathbb{H}(G_j)$ is i-uniform if and only if $i = j$. That is, for each $i \neq j$, we find a hypergraph $H_{ij} := \mathbb{H}(G_i) \in \mathbb{H}_t^{\text{small}}$ such that $\mathbb{H}(G_i) \oplus_{\text{h}} H_{ij} \in \mathscr{F}$ but $\mathbb{H}(G_j) \oplus_{\text{h}} H_{ij} \notin \mathscr{F}$. We conclude that the canonical right congruence $\sim_{\mathscr{F}}$ has infinite index over $\mathbb{H}_t^{\text{small}}$.

We now prove our Myhill-Nerode theorem for hypergraphs. It implies that the trivial problem of checking whether a hypergraph is d-uniform for some d, which we considered in Example 6.3, cannot be solved by tree automata. The obstacle is, again, that tree automata only have a constant amount of memory and therefore cannot count arbitrarily far. In contrast, we will see in Section 6.4 that tree automata can solve the problem of checking whether a hypergraph has constant cutwidth k.

Theorem 6.4. *Let $\mathscr{F} \subseteq \mathbb{H}_t^{small}$ be a hypergraph problem, that is, $\mathscr{F} = \mathbb{H}(F)$ for some generator-total $F \subseteq \mathbb{U}_{t,2}^{small}$. The following statements are equivalent:*

i) The collection of parse trees generating the graphs in F is recognizable by a tree automaton.

ii) The canonical right congruence $\sim_{\mathscr{F}}$ has finite index over $\mathbb{H}_t^{small}$.

iii) The canonical right congruence $\sim_F$ has finite index over $\mathbb{U}_{t,2}^{small}$.

Moreover, if the index p of $\sim_{\mathscr{F}}$ and the index q of $\sim_F$ are finite, they bound each other as $2^t q \geq p \geq q/2^t - 1$.

Proof. Since $F \subseteq \mathbb{U}_{t,2}^{\text{small}}$, we can apply Theorem 6.3, which states that (i) and (iii) are equivalent. It remains to show that (iii) and (ii) are equivalent. That is, we show that $\sim_{\mathscr{F}}$ has finite index over $\mathbb{H}_t^{\text{small}}$ if and only if $\sim_F$ has finite index over $\mathbb{U}_{t,2}^{\text{small}}$.

First, assume that $\sim_F$ has infinite index over $\mathbb{U}_{t,2}^{\text{small}}$. We show that $\sim_{\mathscr{F}}$ has infinite index over $\mathbb{H}_t^{\text{small}}$. Since $\sim_F$ has infinite index over $\mathbb{U}_{t,2}^{\text{small}}$, there is an infinite set $\{G_1, G_2, G_3, \ldots\} \subseteq \mathbb{U}_{t,2}^{\text{small}}$ of graphs that are pairwise nonequivalent under $\sim_F$. Since there are only 2^t possibilities to assign two colors to t boundary vertices, there is an infinite number of color-compatible graphs among $\{G_1, G_2, \ldots\}$. Moreover, notice that all graphs $G_i \in \mathbb{U}_{t,2}^{\text{small}}$ that are not t-boundaried hypergraph generators are equivalent under $\sim_F$: since F contains only t-boundaried hypergraph generators, G_i cannot be completed into graphs in F by gluing any graph onto G_i. Therefore, without loss of generality, we assume that $\{G_1, G_2, \ldots\}$ are pairwise color-compatible t-boundaried hypergraph generators. Now, for each pair G_i, G_j, there is a graph $H_{ij} \in \mathbb{U}_{t,2}^{\text{small}}$ such that, without loss of generality, $G_i \oplus_{\text{c}} H_{ij} \in F$ but $G_j \oplus_{\text{c}} H_{ij} \notin F$. From $G_i \oplus_{\text{c}} H_{ij} \in F$, it follows that H_{ij} is a t-boundaried hypergraph generator that is color-compatible with G_i. Hence, $\mathbb{H}(H_{ij}) \in \mathbb{H}_t^{\text{small}}$. Now, from $G_i \oplus_{\text{c}} H_{ij} \in F$, we get $\mathbb{H}(G_i) \oplus_{\text{h}} \mathbb{H}(H_{ij}) = \mathbb{H}(G_i \oplus_{\text{c}} H_{ij}) \in \mathbb{H}(F) = \mathscr{F}$. Moreover, since F is generator-total, from $G_j \oplus_{\text{c}} H_{ij} \notin F$ it follows that $\mathbb{H}(G_j) \oplus_{\text{h}} \mathbb{H}(H_{ij}) = \mathbb{H}(G_j \oplus_{\text{c}} H_{ij}) \notin \mathbb{H}(F) = \mathscr{F}$. That is, $\mathbb{H}(G_i) \not\sim_{\mathscr{F}} \mathbb{H}(G_j)$ over $\mathbb{H}_t^{\text{small}}$ and, therefore, $\sim_{\mathscr{F}}$ has infinite index.

Now, assume that $\sim_{\mathscr{F}}$ has infinite index over $\mathbb{H}_t^{\text{small}}$. We show that $\sim_F$ has infinite index over $\mathbb{U}_{t,2}^{\text{small}}$. Since $\sim_{\mathscr{F}}$ has infinite index over $\mathbb{H}_t^{\text{small}}$, there is a set $\{\mathbb{H}(G_1), \mathbb{H}(G_2), \mathbb{H}(G_3), \ldots\} \subseteq \mathbb{H}_t^{\text{small}}$ of hypergraphs that are pairwise nonequivalent under $\sim_{\mathscr{F}}$. Since there are only 2^t partitions of t labels into hyperedge labels and vertex labels, there is an infinite number of pairwise gluable hypergraphs among $\{\mathbb{H}(G_1), \mathbb{H}(G_2), \ldots\}$. Therefore, without loss of generality, assume that all these hypergraphs are pairwise gluable. Now, for each pair $\mathbb{H}(G_i)$, $\mathbb{H}(G_j)$, there is a hypergraph $\mathbb{H}(H_{ij}) \in \mathbb{H}_t^{\text{small}}$ such that, without loss of generality, $\mathbb{H}(G_i) \oplus_{\text{h}} \mathbb{H}(H_{ij}) \in \mathbb{H}(F) = \mathscr{F}$ but $\mathbb{H}(G_j) \oplus_{\text{h}} \mathbb{H}(H_{ij}) \notin \mathbb{H}(F) = \mathscr{F}$. Since $\mathbb{H}(G_i) \oplus_{\text{h}} \mathbb{H}(H_{ij}) = \mathbb{H}(G_i \oplus_{\text{c}} H_{ij})$ and F is generator-total, we have $G_i \oplus_{\text{c}} H_{ij} \in F$. Moreover, $G_j \oplus_{\text{c}} H_{ij} \notin F$. Since $H_{ij} \in \mathbb{U}_{t,2}^{\text{small}}$, it follows that $\sim_F$ has infinite index over $\mathbb{U}_{t,2}^{\text{small}}$.

Finally, observe that our proof yields even more information in the case that $\sim_F$, and equivalently $\sim_{\mathscr{F}}$, have finite index: we have first shown that, for any set of q graphs that are pairwise nonequivalent under $\sim_F$, there are at least $p \geq \lceil q/2^t \rceil - 1$ hypergraphs nonequivalent under $\sim_{\mathscr{F}}$. We have then shown that, for any set of p hypergraphs that are pairwise nonequivalent under $\sim_{\mathscr{F}}$, there are at least $q \geq \lceil p/2^t \rceil$ graphs nonequivalent under $\sim_F$. Hence, for the index p of $\sim_{\mathscr{F}}$ and the index q of $\sim_F$, we have $2^t q \geq p \geq q/2^t - 1$. □

6.3 Using the Myhill-Nerode theorem

This section shows how to apply our Myhill-Nerode theorem for hypergraphs in order to derive complexity-theoretical results for hypergraph problems.

First, Section 6.3.1 aims for making the usage of Theorem 6.3 less technical in the scenario where we want to use it for designing fixed-parameter linear-time algorithms.

Then, Section 6.3.2 shows implications of a hypergraph problem not being solvable by tree automata. We have seen in Example 6.3 that tree automata cannot even solve trivial hypergraph problems like checking whether all hyperedges have the same cardinality.

6.3.1 Deriving fixed-parameter linear-time algorithms

In Section 6.2.3, we have seen a tool allowing us to show when a hypergraph problem $\mathscr{F} \in \mathbb{H}_t^{\text{small}}$ can be recognized by a tree automaton, that is, any hypergraph problem $\mathscr{F}$ on hypergraphs that can be generated by t-boundaried hypergraph generators that, in turn, can be generated by operators in Definition 6.3. This, indeed, is very technical. We now show that one does not have to care about these representation issues when solving hypergraph problems that are closed under isomorphism.

For example, we will only need the following proposition in order to show that Hypergraph Cutwidth is fixed-parameter linear in Section 6.4. The proposition allows us to work with arbitrary t-boundaried hypergraphs, that is, hypergraphs in $\mathbb{H}_t^{\text{large}}$, instead of only those graphs that can be generated by the operators in Definition 6.3.

Proposition 6.1. *Let $\mathscr{F} \subseteq \mathbb{H}_t^{large}$ be a decidable hypergraph problem that is closed under isomorphism and that is restricted to hypergraphs of constant incidence treewidth $t-1$.*

Given a hypergraph $H \in \mathbb{H}_t^{large}$ and a constant upper bound on the index of $\sim_{\mathscr{F}}$ over $\mathbb{H}_t^{large}$, we can compute in constant time a tree automaton $\mathscr{A}$ and in linear time a tree T such that $\mathscr{A}$ processes T in linear time and accepts T if and only if $H \in \mathscr{F}$.

Proof. Let $F \subseteq \mathbb{U}_{t,2}^{\text{small}}$ be generator-total such that $\mathscr{F} \cap \mathbb{H}_t^{\text{small}} = \mathbb{H}(F)$ and let p be the constant given upper bound on $\sim_{\mathscr{F}}$. The proof relies on two claims:

i) $H \in \mathscr{F}$ holds if and only if all graphs $G \in \mathbb{U}_{t,2}^{\text{small}}$ with $\mathbb{H}(G) \cong H$ are in F.

ii) $\sim_F$ has index $q \leq 2^t(p+1)$ over $\mathbb{U}_{t,2}^{\text{small}}$.

From (i) then immediately follows that a tree automaton $\mathscr{A}$ deciding F decides $H \in \mathscr{F}$ correctly when fed the parse tree T_G of any $G \in \mathbb{U}_{t,2}^{\text{small}}$ with $\mathbb{H}(G) \cong H$. By Theorem 6.2, this G exists and we obtain the parse tree T_G in linear time from the incidence graph of H.

From (ii) and Theorem 6.4, it follows that the tree automaton $\mathscr{A}$ for F indeed exists. Similarly like automata for regular languages [DF13, Example 12.5.3], it can be constructed in constant time given that we know a constant upper bound s on the number of its states: the crucial observation is that $\mathscr{A}$ reaches at least one state twice when processing a parse tree of height greater than s. Thus, for any parse tree T of height greater than s, there is a parse tree T' of height at most s such that $\mathscr{A}$ accepts T if and only if it accepts T'. It follows that we only have to construct $\mathscr{A}$ so that it works correctly on all parse trees of height at most s. Since our operators in Definition 6.3 are all nullary, unary, or binary, the parse trees of expressions over them are binary trees. Thus, there are only a constant number of parse trees of height at most s. Moreover, for any parse tree T of height at most s, we can decide in constant time whether the hypergraph it generates is in $\mathscr{F}$, since $\mathscr{F}$ is decidable. Hence, we can construct in constant time by brute force a tree automaton $\mathscr{A}$ that correctly answers for all parse trees of height at most s and, consequently, recognizes F. Moreover, since $\mathscr{A}$ has a constant number of states, it takes only linear time for $\mathscr{A}$ to process any parse tree.

The constant upper bound s on the number of states of $\mathscr{A}$ we obtain as follows: the index of $\sim_F$ is bounded by (ii), which, in turn bounds the number of states of $\mathscr{A}$ [CE12, Theorem 4.34(2) and Proposition 3.73(2)]. Thus, it only remains to prove (i) and (ii).

(i) First, assume that there is some graph $G \in \mathbb{U}_{t,2}^{\text{small}}$ with $\mathbb{H}(G) \cong H$ in F. Then, since G is a t-boundaried hypergraph generator in $\mathbb{U}_{t,2}^{\text{small}}$, $\mathbb{H}(G) \in \mathbb{H}(F)$.

Since $\mathbb{H}(F) \subseteq \mathscr{F}$ and $\mathscr{F}$ is closed under isomorphism, we conclude $H \in \mathscr{F}$. If, for the opposite direction, $H \in \mathscr{F}$, then let $G \in \mathbb{U}_{t,2}^{\text{small}}$ be any graph such that $\mathbb{H}(G) \cong H$. Since $G \in \mathbb{U}_{t,2}^{\text{small}}$, we have $\mathbb{H}(G) \in \mathbb{H}_t^{\text{small}}$. Moreover, since $\mathscr{F}$ is closed under isomorphism, $\mathbb{H}(G) \in \mathscr{F}$ and, hence, $\mathbb{H}(G) \in \mathbb{H}(F)$. Finally, since F is generator-total, we have $G \in F$.

(ii) We show that the index p' of $\sim_{\mathbb{H}(F)}$ over $\mathbb{H}_t^{\text{small}}$ is at most p. Then, from Theorem 6.4, it follows that $p \geq p' \geq q/2^t p - 1$ and, hence, $q \leq 2^t(p+1)$. Thus, we only have to show, for any $\mathbb{H}(G_1), \mathbb{H}(G_2) \in \mathbb{H}_t^{\text{small}}$ equivalent under $\sim_{\mathscr{F}}$, that they are also equivalent under $\sim_{\mathbb{H}(F)}$. This is trivial, since, for $i \in \{1,2\}$ and any $\mathbb{H}(H) \in \mathbb{H}_t^{\text{small}}$, we have $\mathbb{H}(G_i) \oplus_{\text{h}} \mathbb{H}(H) \in \mathbb{H}(F) \iff \mathbb{H}(G_i) \oplus_{\text{h}} \mathbb{H}(H) \in \mathscr{F}$, since $\mathbb{H}(G_i) \oplus_{\text{h}} \mathbb{H}(H) = \mathbb{H}(G_i \oplus_{\text{c}} H) \in \mathbb{H}_t^{\text{small}}$ and $\mathbb{H}(F) = \mathscr{F} \cap \mathbb{H}_t^{\text{small}}$. □

6.3.2 Expressibility in monadic second-order logic

In Example 6.3 of Section 6.2.3, we have seen that tree automata cannot even solve trivial hypergraph problems like checking whether all hyperedges have the same cardinality. Thus, the question naturally arises whether not being solvable by a tree automaton gives any information about a hypergraph problem at all. It does: we will see that properties that cannot be checked by tree automata also cannot be expressed in monadic second-order logic, which is a standard tool for showing linear-time solvability of graph problems parameterized by treewidth. We have introduced the monadic second-order logic for graphs and shown examples for its applications in Section 2.5.1. For hypergraphs, we use the following logic. The only difference to the monadic second-order logic for graphs defined in Definition 2.11 in Section 2.5.1 is that our logic for hypergraphs does not have the predicate $\text{col}_i(v)$ to check whether a vertex v has color i.

Definition 6.8 (Monadic second-order logic for hypergraphs)**.** A formula φ of the monadic second-order logic for hypergraphs may consist of the logic operators $\vee, \wedge, \neg$, vertex variables, hyperedge variables, set variables, quantifiers $\exists$ and $\forall$ over vertices, hyperedges, and sets, and the predicates

$x \in X$ for a vertex or hyperedge variable x and a set X,

$\text{inc}(e,v)$, being true if e is a hyperedge incident to the vertex v,

$\text{adj}(v,w)$, being true if v and w occur in the same hyperedge, and

equality of vertex variables, edge variables, and set variables.

We use upper-case letters for set variables and lower-case letters for vertex and edge variables.

Proposition 6.2. *Let $\mathscr{F} \subseteq \mathbb{H}_t^{small}$ be a hypergraph problem such that $\sim_{\mathscr{F}}$ has infinite index over $\mathbb{H}_t^{small}$.*

Then there is no monadic second-order logic formula that a hypergraph $H \in \mathbb{H}_t^{small}$ satisfies if and only if $H \in \mathscr{F}$.

Proof. Let $F \subseteq \mathbb{U}_{t,2}^{\text{small}}$ be generator-total such that $\mathbb{H}(F) = \mathscr{F}$ and assume, towards a contradiction, that there is a formula φ in monadic second-order logic for hypergraphs such that $\mathscr{F} = \{H \in \mathbb{H}_t^{\text{small}} \mid H \text{ satisfies } \varphi\}$. We will turn φ into a formula φ^* in monadic second-order logic for graphs such that a hypergraph $H \in \mathbb{H}_t^{\text{small}}$ satisfies φ if and only if all graphs $G \in \mathbb{U}_{t,2}^{\text{small}}$ with $\mathbb{H}(G) = H$ satisfy φ^*. That is, $F = \{G \in \mathbb{U}_{t,2}^{\text{small}} \mid G \text{ satisfies } \varphi^*\}$. Courcelle's theorem [CE12, Theorem 6.3(2)] shows that φ^* can be turned into a tree automaton $\mathscr{A}_{\varphi^*}$ such that the parse tree of a graph $G \in \mathbb{U}_{t,2}^{\text{small}}$ is accepted by $\mathscr{A}_{\varphi^*}$ if and only if G satisfies φ^*. Consequently, A_{φ^*} recognizes F. This, by Theorem 6.4, contradicts $\sim_{\mathscr{F}}$ having infinite index.

It remains to describe the transformation from φ to φ^*. To this end, recall that, by Definition 6.6 of t-boundaried hypergraph generators, the color-1 vertices in a graph $G \in \mathbb{U}_{t,2}^{\text{small}}$ with $\mathbb{H}(G) = H$ represent vertices of H while the color-2 vertices in G represent hyperedges of H. Hence, the vertex and hyperedge variables in φ both become vertex variables in φ^*. Moreover, the formula φ^* makes sure that the graph $G \in \mathbb{U}_{t,2}^{\text{small}}$ satisfying φ^* is a t-boundaried hypergraph generator, that is, vertices of the same color are nonadjacent in G and each object (vertex or edge) in G either has a color or is an edge. Thus, we let

$$\begin{aligned}
\varphi^* &:= \text{all-labeled} \wedge \text{bipartite} \wedge \varphi',\\
\text{all-labeled} &:= \forall v[\text{col}_1(v) \vee \text{col}_2(v) \vee \text{is-edge}(v)],\\
\text{is-edge}(v) &:= \exists w[\text{inc}(v,w)],\\
\text{bipartite} &:= \forall v \forall w[(\text{col}_1(v) \wedge \text{col}_2(w)) \vee (\text{col}_2(v) \wedge \text{col}_1(w)) \vee \neg \text{adj}(v,w)],
\end{aligned}$$

where we obtain φ' by replacing terms in φ referring to hypergraphs by equivalent terms referring to incidence graphs. The term replacement makes sure that φ' uses only the vertex variables of the incidence graph (the edge variables have no correspondence in the hypergraph) and, of course, translate incidence and adjacency of hypergraph objects into adjacency of the corresponding incidence graph objects. Specifically, we replace the following hypergraph expressions on the left-hand side by the equivalent graph expressions on the right-hand side:

$$\exists x[\psi] \equiv \exists x[(\mathrm{col}_1(x) \vee \mathrm{col}_2(x)) \wedge \psi],$$
$$\forall x[\psi] \equiv \forall x[(\neg\,\mathrm{col}_1(x) \wedge \neg\,\mathrm{col}_2(x)) \vee \psi],$$
$$\exists X[\psi] \equiv \exists X[\forall x[x \notin X \vee \mathrm{col}_1(x) \vee \mathrm{col}_2(x)] \wedge \psi],$$
$$\forall X[\psi] \equiv \forall X[\exists x[x \in X \wedge \neg\,\mathrm{col}_1(x) \wedge \neg\,\mathrm{col}_2(x)] \vee \psi],$$
$$\mathrm{inc}(e,v) \equiv \mathrm{col}_2(e) \wedge \mathrm{col}_1(v) \wedge \mathrm{adj}(e,v), \text{ and}$$
$$\mathrm{adj}(v,w) \equiv \mathrm{col}_1(v) \wedge \mathrm{col}_1(w) \wedge \exists e[\mathrm{col}_2(e) \wedge \mathrm{adj}(e,v) \wedge \mathrm{adj}(e,w)].$$ □

6.4 Hypergraph Cutwidth is fixed-parameter linear

In this section, we use the Myhill-Nerode theorem for hypergraphs to show that Hypergraph Cutwidth is fixed-parameter linear. We first formally define this NP-hard problem [Gav77] and give an example for an application.

Let $H = (V,E)$ be a hypergraph. A *linear layout* of H is an injective function $l : V \to \mathbb{R}$ of vertices onto the real line. The *cut at position* $i \in \mathbb{R}$ in H with respect to l, denoted $\mathrm{Cut}^l_H(i)$, is the set of hyperedges that contain at least two vertices v,w such that $l(v) < i < l(w)$. We will also say that v is to the *left* of i and that w is to the *right* of i. The *cutwidth* of the layout l is $\max_{i\in\mathbb{R}} |\mathrm{Cut}^l_H(i)|$. The *cutwidth* of the hypergraph H is the minimum cutwidth over all linear layouts of H. The Hypergraph Cutwidth problem is defined as follows.

Hypergraph Cutwidth
Input: A hypergraph $H = (V,E)$ and a natural number k.
Question: Does H have cutwidth at most k?

Example 6.4. Prasad, Chong, and Keutzer [PCK99] and Wang et al. [WCZK01] noticed that Hypergraph Cutwidth is helpful in solving the NP-hard k-Sat problem.

k-Sat
Input: A Boolean formula φ in conjunctive normal form with at most k literals per clause.
Question: Does φ have a satisfying assignment?

Wang et al. [WCZK01], for example, map a formula φ to a hypergraph H as illustrated in Figure 6.8: each variable of φ is a vertex in H and, for each clause of φ, there is a hyperedge in φ containing the variables contained in the clause.

(a) A layout of width three.

(b) A layout of width two.

Figure 6.8: Two layouts of the hypergraph obtained from the Boolean formula $(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (x_2 \vee x_4)$.

Wang et al. [WCZK01] noticed that k-SAT is fixed-parameter tractable with respect to the cutwidth of the hypergraph obtained from the input formula. Their algorithm is a simple search tree algorithm that iterates over the variables of φ from left to right in a minimum cutwidth layout and, for each variable, branches into the two possibilities of setting it to true or false. For example, in Figure 6.8, their algorithm would iterate over the variables in order x_1, x_3, x_2, x_4. For an experimental evaluation of their algorithm, Wang et al. [WCZK01] compute layouts of small cutwidth heuristically.

In this section, we will not solve the problem of finding a layout of constant cutwidth at most k. Instead, we will only show that it is possible to decide in linear time whether a layout of constant width k *exists*. To this end, we employ the Myhill-Nerode theorem for hypergraphs. In the remainder of this section, we consider a constant k and the class k-HCW of all hypergraphs with cutwidth at most k. Since k-HCW is closed under isomorphism, we can solve it in linear time using Proposition 6.1. This will yield the main result of this section:

Theorem 6.5. HYPERGRAPH CUTWIDTH *is fixed-parameter linear.*

In order to use Proposition 6.1 to prove Theorem 6.5, we proceed as follows. First, Section 6.4.1 shows that the hypergraphs in k-HCW have a constant upper bound on their incidence treewidth. It then remains to show that the canonical right congruence $\sim_{k\text{-HCW}}$ has finite index. By Proposition 6.1, it then follows that there is an algorithm to solve k-HCW in linear time, completing the proof of Theorem 6.5.[3]

[3] The running time is linear for constant k but grows at least exponentially in the index of $\sim_{k\text{-HCW}}$, for which, in turn, we will only give a bound that is doubly-exponential in k. For this reason, we do not exactly analyze the growth in k and it does not seem worthwhile implementing and experimentally evaluating the resulting algorithm.

To show that $\sim_{k\text{-HCW}}$ has finite index, Section 6.4.2 shows that two hypergraphs are equivalent under $\sim_{k\text{-HCW}}$ if they pass the same set of *tests*. Section 6.4.3 then shows that each test can be replaced by an equivalent test of constant size and, thus, that the number of pairwise nonequivalent sets of tests is finite. It follows that $\sim_{k\text{-HCW}}$ has finite index.

6.4.1 Hypergraph cutwidth bounds incidence treewidth

We now show that all hypergraphs in k-HCW, that is, all hypergraphs of cutwidth at most k, have a constant upper bound on their treewidth. To obtain a fixed-parameter linear-time algorithm for Hypergraph Cutwidth using Proposition 6.1 and, thus, to prove Theorem 6.5, it then remains to show that $\sim_{k\text{-HCW}}$ has finite index.

Lemma 6.1. *Let H be a hypergraph. If H has cutwidth at most k, then H has incidence treewidth at most* $\max\{k,1\}$.

Proof. Suppose that $H=(V,E)$ has cutwidth at most k. Let $H'=(V,E')$ denote the hypergraph obtained from H by removing all hyperedges of size at most one. Consider a linear layout l of cutwidth at most k of the vertices of H'. Without loss of generality, assume that l maps to the natural numbers $[n]$ and let $V=\{v_1,\ldots,v_n\}$ be such that $l(v_i)=i$. We construct a path decomposition for the incidence graph $\mathscr{I}(H')$ of H' with the bags $L_1,R_1,L_2,R_2,\ldots,L_n,R_n$ that are connected by a path in this order. For every $i\in[n]$, let $L_i := \mathrm{Cut}^l_{H'}(i-1/2)\cup\{v_i\}$ and $R_i := \mathrm{Cut}^l_{H'}(i+1/2)\cup\{v_i\}$, that is, L_i contains v_i and all hyperedges cut at $i-1/2$, while R_i contains v_i and all hyperedges cut at $i+1/2$. Herein, recall that the hyperedges of H' are vertices in $\mathscr{I}(H')$. We now prove that this is a path decomposition for $\mathscr{I}(H')$. To this end, we verify that Definition 2.10 in Section 2.5 is satisfied.

First, we show that each edge of $\mathscr{I}(H')$ is contained in at least one bag. Let $\{v_i,e\}$ be any edge in $\mathscr{I}(H')$ for some vertex $v_i\in V$ and a hyperedge $e\in E'$. We show that v_i and e occur together in at least one bag. Since $v_i\in e$ and $|e|\geq 2$, the hyperedge e contains at least one vertex to the left or to the right of v_i. Hence, we have $e\in\mathrm{Cut}^l_{H'}(i-1/2)$ or $e\in\mathrm{Cut}^l_{H'}(i+1/2)$. Therefore, it holds that $e\in R_i$ or $e\in L_i$. Since $v_i\in R_i\cap L_i$, the vertices v_i and e occur together in at least one bag.

Now, we show that the bags containing a vertex of $\mathscr{I}(H')$ induce a subpath in this path decomposition. Obviously, each vertex $v_i\in V$ is contained in two bags of the path decomposition: in L_i and R_i. These bags are consecutive and thus induce a path. Finally, consider a hyperedge $e\in E'$. It occurs in

all bags $R_i, L_{i+1}, R_{i+1}, \ldots, L_{j-1}, R_{j-1}, L_j$, where v_i is the leftmost vertex in the layout l occurring in e and v_j is the rightmost vertex in l occurring in e. These bags are all consecutive on the path and, thus, induce a path.

The width of this path decomposition is $\max_{0 \leq i \leq n} |\operatorname{Cut}^l_{H'}(i+1/2)| \leq k$. To obtain a tree decomposition for H from the path decomposition of H', we only need to take care of hyperedges of size at most one. For every hyperedge $e \in E$ of size one, add a new bag $\{e, v\}$, where v is the unique vertex contained in e, and make it adjacent to an arbitrary bag containing v. For every empty hyperedge $e \in E$, add a new bag $\{e\}$, and make it adjacent to an arbitrary bag. In this way, we obtain a tree decomposition for the incidence graph $\mathcal{I}(H)$ of H of width at most $\max\{k, 1\}$. Thus, H has incidence treewidth at most $\max\{k, 1\}$. □

6.4.2 Tests for constant cutwidth

We have shown that all hypergraphs in the set k-HCW of hypergraphs of constant cutwidth at most $k \geq 1$ have treewidth at most k. To obtain a linear-time algorithm for k-HCW using Proposition 6.1, and, thus, to prove Theorem 6.5, it remains, for all $t \leq k+1$, to prove that the canonical right congruence $\sim_{k\text{-HCW}}$ of k-HCW has finite index over the set $\mathbb{H}_t^{\text{large}}$ of all t-boundaried hypergraphs.

To show that $\sim_{k\text{-HCW}}$ has finite index over $\mathbb{H}_t^{\text{large}}$, we show that, given a t-boundaried hypergraph G, only a finite number of bits of information about a t-boundaried hypergraph H is needed in order to decide whether $G \oplus_{\text{h}} H \in k$-HCW. To this end, we employ the method of test sets [DF13, Section 12.7]: let $\mathcal{T}$ be a set of objects called *tests* (we will formally define a test later). A t-boundaried graph can *pass* a test. For t-boundaried hypergraphs G_1 and G_2, let $G_1 \sim_{\mathcal{T}} G_2$ if and only if G_1 and G_2 pass the same subset of tests in $\mathcal{T}$. Obviously, $\sim_{\mathcal{T}}$ is an equivalence relation.

In this section, our aim is to find a set $\mathcal{T}$ of tests such that $\sim_{\mathcal{T}}$ refines $\sim_{k\text{-HCW}}$, that is, such that $G_1 \sim_{\mathcal{T}} G_2$ implies $G_1 \sim_{k\text{-HCW}} G_2$. Then, if $\sim_{\mathcal{T}}$ has finite index, so has $\sim_{k\text{-HCW}}$.

To show that $\sim_{k\text{-HCW}}$ has finite index, Section 6.4.3 then shows that the set $\mathcal{T}$ of tests can be replaced by an equivalent *finite* set of tests. Thus, the index of $\sim_{\mathcal{T}}$ and, hence, the index of $\sim_{k\text{-HCW}}$ will be finite.

Intuitively, we will define, for a hypergraph H, an *H-test*, which a hypergraph G satisfies if $G \oplus_{\text{h}} H \in k$-HCW. We define the H-test so that it contains only the necessary information of H and so that we can later shrink all tests to equivalent tests of constant size. We now formally define a test.

Definition 6.9. A *size-n test* T for k-HCW over $\mathbb{H}_t^{\text{large}}$ is a triple (π, S, k), where

$\pi\colon [t] \to [n]$ is a map of boundary labels to integer positions, and

$S = (w_i, E_i) \in \{0,\dots,k\} \times 2^{[t]}$ such that, if $\ell \in E_p$ and $\ell \in E_q$, then $\ell \in E_i$ for all $i \in \{p,\dots,q\}$.

As a special case of a test, we now formally define an H-test; it is illustrated in Figure 6.9. After the definition, we give an intuitive description of the introduced concepts.

Definition 6.10. Let G and H be t-boundaried hypergraphs such that $G \oplus_{\mathrm{h}} H \in k$-HCW. Moreover, let l be a linear layout for $G \oplus_{\mathrm{h}} H$ with minimum cutwidth, which, without loss of generality, maps vertices of the n-vertex hypergraph H to the integer positions $[n]$.

An *H-test* is a size-n test $T = (\pi, S, k)$ for k-HCW with $S = (S_1, \dots, S_n)$, where

$\pi(\ell)$ $:= l(v)$ for the vertex $v \in \partial(H)$ with label ℓ, and

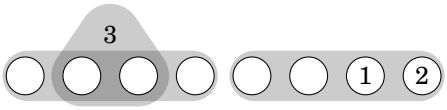
S_i $:= (w_i, E_i)$ for $i \in \{0,\dots,n\}$, where

w_i is the number of unlabeled hyperedges of H containing vertices v, w of H with $l(v) \le i < l(w)$, and

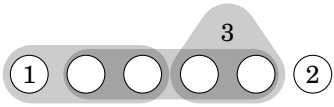
E_i is the set of labels of hyperedges in H containing vertices v, w of H with $l(v) \le i \le l(w)$.

The goal of Definition 6.10 is that, if a hypergraph G passes an H-test for k-HCW, then $G \oplus_{\mathrm{h}} H \in k$-HCW. More precisely, we want that, if a hypergraph G passes an H-test, then $G \oplus_{\mathrm{h}} H$ has a linear layout l of cutwidth at most k that lays out the vertices of H in the same way as the layout used to create the H-test. Note that the H-test does not record the precise structure of H but only the most important information:

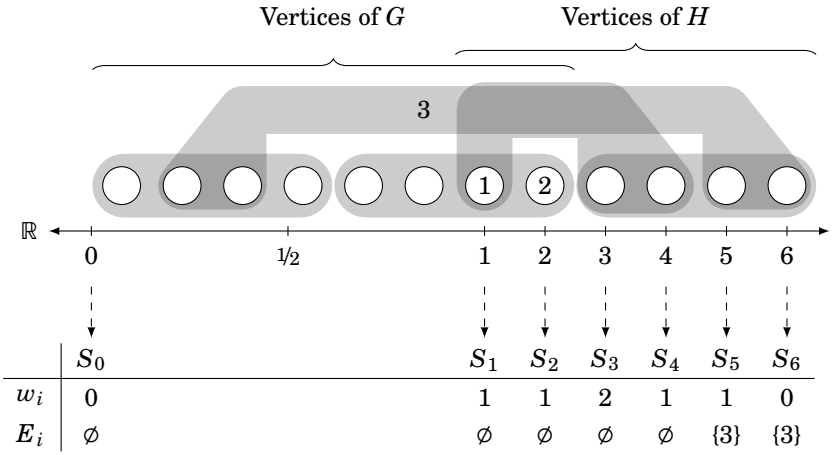
Assume that we want to verify that the cutwidth of the layout l of $G \oplus_{\mathrm{h}} H$ is at most k without knowing H but only knowing G and the H-test. Then, for any non-integer position i, the value $w_{\lfloor i \rfloor}$ counts the unlabeled hyperedges of H cut at i. Thus, to the size of any cut for G at position $i \in \mathbb{R} \setminus \mathbb{Z}$, we have to add the value $w_{\lfloor i \rfloor}$. For labeled hyperedges of H, things are more difficult: they contain vertices of $G \oplus_{\mathrm{h}} H$ that originate from G as well as from H. Since an H-test corresponds to a fixed layout for H, to count a hyperedge with label ℓ of $G \oplus_{\mathrm{h}} H$ that is cut at some position, it is sufficient to know the vertices of the



(a) A 3-boundaried hypergraph G.



(b) A 3-boundaried hypergraph H.



(c) The glued hypergraph $G \oplus_{\mathrm{h}} H$ laid out on the real number line and an H-test.

Figure 6.9: Construction of the H-test from a glued hypergraph $G \oplus_{\mathrm{h}} H$. The vertices of H lie at the positions $\{1,2,\ldots,6\}$ and the non-boundary vertices of G lie in the open interval $(0,1)$. Moreover, in the H-test, we have $\pi(1) = 1$ and $\pi(2) = 2$.

hyperedge with label ℓ in G and the positions of the leftmost and the rightmost vertex contained in the hyperedge of H with label ℓ. However, in order to easier shrink all tests to constant size later, we choose a more convenient way to keep this information in the H-test: for any position i between the leftmost and the rightmost vertex of a hyperedge e in H with label ℓ, we have $\ell \in E_i$. We now precisely define what it means to pass a test.

Definition 6.11. Let $G = (V, E)$ be a t-boundaried hypergraph and $T = (\pi, S, k)$ be a test of size n, where $S = (S_0, \dots, S_n)$ and $S_i = (w_i, E_i)$.

A *T-compatible layout for G* is an injective function $f : V \to \mathbb{R}$ such that each vertex $v \in \partial(G)$ with label ℓ is mapped to $\pi(\ell)$ and such that each vertex $v \in V \setminus \partial(G)$ is mapped into some open interval $(i, i+1)$ for $0 \leq i \leq n$.

For a hyperedge e in G, we define the set of its *positions*

$$\mathrm{Pos}(e) := \begin{cases} \{f(v) \mid v \in e\} & \text{if } e \text{ is unlabeled,} \\ \{f(v) \mid v \in e\} \cup \{i \mid \ell \in E_i\} & \text{if } e \text{ has label } \ell. \end{cases}$$

The *combined cut at i* in G with respect to f is the set $\mathrm{Ccut}_G^f(i)$ of hyperedges e of G for which there are positions $j, k \in \mathrm{Pos}(e)$ with $j < i < k$.

The *combined cutwidth* of f is

$$\max_{i \in \mathbb{R} \setminus \mathbb{Z}} (|\mathrm{Ccut}_G^f(i)| + w_{\lfloor i \rfloor}).$$

Finally, *G passes the test T* if there is a T-compatible layout f for G whose combined cutwidth is at most k.

We now conclude this section by showing that, indeed, if two graphs satisfy the same tests, then they are equivalent under $\sim_{k\text{-HCW}}$. Section 6.4.3 will then show that, actually, there is only a finite set of pairwise nonequivalent tests, thus showing that $\sim_{k\text{-HCW}}$ has finite index and, consequently, that HYPERGRAPH CUTWIDTH is fixed-parameter linear.

Lemma 6.2. *For $\mathcal{T}$ being the set of all tests for k-HCW, the equivalence relation $\sim_{\mathcal{T}}$ refines $\sim_{k\text{-HCW}}$.*

To prove Lemma 6.2, we show that, if two t-boundaried hypergraphs G_1 and G_2 pass the same subset of tests of $\mathcal{T}$, then, for all t-boundaried hypergraphs H, $G_1 \oplus_{\mathrm{h}} H \in k\text{-HCW}$ if and only if $G_2 \oplus_{\mathrm{h}} H \in k\text{-HCW}$. The proof is based on the following two claims.

Claim 6.1. *If $G_1 \oplus_h H \in k$-HCW, then G_1 passes some H-test.*

Claim 6.2. *If G_2 passes any H-test, then $G_2 \oplus_h H \in k$-HCW.*

From these two claims, Lemma 6.2 then easily follows: let H be a t-boundaried hypergraph such that $G_1 \oplus_h H \in k$-HCW. By Claim 6.1, G_1 passes some H-test T. Since G_1 and G_2 pass the same tests, also G_2 passes T. By Claim 6.2, it follows that $G_2 \oplus_h H \in k$-HCW. The reverse direction is proven symmetrically.

Thus, it only remains to prove Claim 6.1 and Claim 6.2. The final step to a fixed-parameter linear-time algorithm for HYPERGRAPH CUTWIDTH is then made by the next section: it shows that any test can be replaced by an equivalent test of constant size.

Proof of Claim 6.1. Let T be the H-test obtained from an optimal layout l for $G_1 \oplus_h H$, which, without loss of generality, maps the vertices of the n-vertex graph H to the integer positions $[n]$ and the vertices of $V(G_1) \setminus \partial(G_1)$ to non-integer positions in the interval $(0, n+1)$. We show that H passes T.

First, observe that l is a T-compatible layout for G_1. It remains to show that the combined cutwidth $\max_{i \in \mathbb{R} \setminus \mathbb{Z}}(|\operatorname{Ccut}^l_{G_1}(i)| + w_{\lfloor i \rfloor})$ of l from Definition 6.11 is at most k.

To this end, for an $i \in \mathbb{R} \setminus \mathbb{Z}$, consider the set $A := \operatorname{Cut}^l_{G_1 \oplus_h H}(i)$ of hyperedges of $G_1 \oplus_h H$ containing two vertices v, w with $l(v) < i < l(w)$. Since $G_1 \oplus_h H \in k$-HCW, we have $|A| \leq k$. Thus, it is sufficient to show that $|\operatorname{Ccut}^l_{G_1}(i)| + w_{\lfloor i \rfloor} \leq |A|$. We partition A into two sets B and C, where B are the unlabeled hyperedges in H.

Since $i \notin \mathbb{Z}$, by Definition 6.10, $w_{\lfloor i \rfloor}$ counts exactly the hyperedges in B. It remains to show that $|\operatorname{Ccut}^l_{G_1}(i)| \leq |C|$. Recall from Definition 6.11 that $\operatorname{Ccut}^l_{G_1}(i)$ is the set of hyperedges e of G_1 for which $\operatorname{Pos}(e)$ contains two positions j, k with $j < i < k$. If e is unlabeled, then, by Definition 6.11 of $\operatorname{Pos}(e)$, the hyperedge e of G_1 contains vertices v, w with $l(v) = j$ and $l(w) = k$. Since e, v, and w are also in $G_1 \oplus H$, we have $e \in C$. If e is labeled, then $G_1 \oplus_h H$ instead of e contains a hyperedge $e' \supseteq e$. Now, since $j \in \operatorname{Pos}(e)$, the hypergraph G_1 contains a vertex v with $l(v) = j$ or $\ell \in E_j$, which, by Definition 6.10, implies that there is a hyperedge with label ℓ containing a vertex v in H with $l(v) \leq i$. Likewise, G_1 contains a vertex w with $l(w) = k$ or $\ell \in E_k$, which implies that there is a hyperedge with label ℓ containing a vertex w in H with $k \leq l(w)$. In all cases, we have that $l(v) < i < l(w)$. Since the hyperedge e' of $G_1 \oplus_h H$ contains v and w, we get $e' \in C$. □

Proof of Claim 6.2. Let T be an H-test obtained from a cutwidth-k linear layout l for some $G^* \oplus_h H$ and assume that G_2 passes T. We show that $G_2 \oplus_h H \in k$-HCW.

Since G_2 passes T, there is a T-compatible layout f for G_2 with combined cutwidth at most k. First note that l and f agree on the layout of vertices in $\partial(G_2)$ and $\partial(H)$ and that, apart from these, f lays out vertices at non-integer positions, whereas l lays out vertices of H at integer positions. Because of this, in a layout g for $G_2 \oplus_\mathrm{h} H$ that lays out vertices v of H at position $l(v)$ and vertices v of G_2 at position $f(v)$, every two vertices in $G_2 \oplus_\mathrm{h} H$ are laid out at distinct positions by g. Hence, g is injective and, therefore, a layout.

We show that g is a layout of cutwidth at most k for $G_2 \oplus_\mathrm{h} H$. That is, we show $\max_{i\in\mathbb{R}} |\operatorname{Cut}^g_{G_2\oplus_\mathrm{h} H}(i)| \leq k$. To this end, note that

$$\max_{i\in\mathbb{R}} |\operatorname{Cut}^g_{G_2\oplus_\mathrm{h} H}(i)| \leq \max_{i\in\mathbb{R}\setminus\mathbb{Z}} |\operatorname{Cut}^g_{G_2\oplus_\mathrm{h} H}(i)|,$$

since, for every $i \in \mathbb{Z}$, we have $\operatorname{Cut}^g_{G_2\oplus_\mathrm{h} H}(i) \subseteq \operatorname{Cut}^g_{G_2\oplus_\mathrm{h} H}(i+\varepsilon)$ for $0 < \varepsilon < 1$ chosen so that no vertex is mapped by g to the interval $(i, i+\varepsilon]$. That is, we only have to show that, for each $i \in \mathbb{R}\setminus\mathbb{Z}$, we have $|\operatorname{Cut}^g_{G_2\oplus H}(i)| \leq |\operatorname{Ccut}^f_{G_2}(i)| + w_{\lfloor i\rfloor}$, since f is a layout for G_2 with combined cutwidth $\max_{i\in\mathbb{R}\setminus\mathbb{N}}(|\operatorname{Ccut}^f_{G_2}(i)| + w_{\lfloor i\rfloor}) \leq k$.

For some position $i \in \mathbb{R}\setminus\mathbb{Z}$, consider the set $A := \operatorname{Cut}^g_{G_2\oplus_\mathrm{h} H}(i)$ of hyperedges of $G_2 \oplus_\mathrm{h} H$ containing vertices v, w with $g(v) < i < g(w)$ and let it be partitioned into two sets B and C, where B contains the unlabeled hyperedges of H. We show that $|A| \leq w_{\lfloor i\rfloor} + |\operatorname{Ccut}^f_{G_2}(i)|$. By Definition 6.10, we clearly have $|B| \leq w_{\lfloor i\rfloor}$. Hence, it remains to show that $|C| \leq |\operatorname{Ccut}^f_{G_2}(i)|$.

To this end, consider a hyperedge $e \in C$. If e contains only vertices of G_2, then for any vertex $v \in e$, we have $g(v) = f(v) \in \operatorname{Pos}(e)$. It follows that $\operatorname{Pos}(e)$ contains positions $j = g(v)$ and $k = g(w)$ with $j < i < k$ and, therefore, $e \in \operatorname{Ccut}^f_{G_2}(i)$. If e additionally contains a vertex v of H, then e has a label ℓ and H has a hyperedge with label ℓ containing v. Hence, in this case, we have $\ell \in E_{l(v)}$. It follows that $g(v) = l(v) \in \operatorname{Pos}(e')$ for the hyperedge $e' \subseteq e$ of G_2 with label ℓ. Hence, for any vertex $v \in e$, we have $g(v) \in \operatorname{Pos}(e')$, independently of whether v is a vertex of G_2 or of H. Since e contains vertices v, w with $g(v) < i < g(w)$, it follows that $\operatorname{Pos}(e')$ contains positions $j = g(v)$ and $k = g(w)$ with $j < i < k$ and, therefore, that $e' \in \operatorname{Ccut}^f_{G_2}(i)$. □

6.4.3 Shrinking tests to constant size

Towards our goal of showing that $\sim_{k\text{-HCW}}$ has finite index, Section 6.4.2 has shown a set $\mathcal{T}$ of tests such that $\sim_{\mathcal{T}}$ refines $\sim_{k\text{-HCW}}$, where two hypergraphs are equivalent with respect to $\sim_{\mathcal{T}}$ if and only if they pass the same subset of tests of $\mathcal{T}$. However, since the set $\mathcal{T}$ is infinite, we cannot yet conclude that $\sim_{\mathcal{T}}$, and therefore $\sim_{k\text{-HCW}}$, has finite index. We now replace $\mathcal{T}$ by an equivalent finite set.

To this end, recall that, in order to apply Proposition 6.1 to obtain a fixed-parameter linear-time algorithm for HYPERGRAPH CUTWIDTH, it is enough, by Lemma 6.1, to give a constant upper bound on the index of $\sim_{k\text{-HCW}}$ over $\mathbb{H}_t^{\text{large}}$ for constant k and $t \leq k+1$. The following lemma will, for every test $T \in \mathscr{T}$, find a test $T' \in \mathscr{T}$ such that a hypergraph G passes T' if and only if it passes T and such that T' has size at most $2t(t+1)(2k+2)$. Thus, the equivalence relation $\sim_{\mathscr{T}'}$ for $\mathscr{T}'$ being the set of all tests of size at most $2t(t+1)(2k+2)$ is the same as $\sim_{\mathscr{T}}$ and, by Lemma 6.2, refines $\sim_{k\text{-HCW}}$. Since there is only a constant number of tests of size $2t(t+1)(2k+2)$, the size of $\mathscr{T}'$ is constant. Since $\sim_{\mathscr{T}'}$, and therefore $\sim_{k\text{-HCW}}$, has at most $2^{|\mathscr{T}'|}$ equivalence classes, it follows that $\sim_{k\text{-HCW}}$ has finite index. Thus, the following lemma finishes our proof of Theorem 6.5: HYPERGRAPH CUTWIDTH is fixed-parameter linear.

Lemma 6.3. *Let G be a t-boundaried hypergraph. For every test T_1, there is a test T_2 of size $2t(t+1)(2k+2)$ such that G passes T_1 if and only G passes T_2.*

Proof. Let the size of the test $T_1 = (\pi, S, k)$ be n. For $E \subseteq [t]$, we call a maximal subsequence $S_j = (E_j, w_j), \ldots, S_k = (E_k, w_k)$ of S with $E = E_j = \cdots = E_k$ a *strait*. We first show that there are at most $2t$ straits, and then show that we can shorten each strait to length at most $(t+1)(2k+2)$ by removing some elements from S without changing the satisfiability of the test. For a label $\ell \in [t]$, let $I_\ell := \{i \in \{0,\ldots,n\} \mid \ell \in E_i\}$. By Definition 6.9, each I_ℓ for some label $\ell \in [t]$ is an interval of the natural numbers with a minimum element and a maximum element, which we both call *events*. Hence, the I_ℓ for all $\ell \in [t]$ in total have at most $2t$ events. Since straits can only start at an event, and since only one strait can start at a fixed event, it follows that S is partitioned into at most $2t$ straits.

It remains to shorten the straits. To this end, we apply data reduction rules already used by Downey and Fellows [DF13, Theorem 12.7.5] for the cutwidth problem on graphs. Let $S_j = (E, w_j), \ldots, S_k = (E, w_k)$ be a strait in T_1. We call a maximal sequence of $w_i, w_{i+1}, \ldots, w_{i'}$ of that strait such that π maps no boundary label to $\{i, i+1, \ldots, i'\}$ a *load pattern*. Hence, each strait decomposes into at most $t+1$ load patterns, each of which we will shorten to length at most $2k+2$.

To this end, first observe that, if the test T_1 passed by G contains a pair $S_i = (E, w_i)$, then G also passes the test obtained from T_1 by replacing S_i by $S_i' = (E, w_i')$ with $w_i' \leq w_i$. Moreover, assume that, as illustrated in Figure 6.10, T_1 contains two pairs $S_i = (E, w_i), S_{i+1} = (E, w_{i+1})$ with $w_i = w_{i+1}$ such that π maps no boundary label to $i+1$. Then G passes the test obtained from T_1 by removing S_{i+1}. Moreover, G then also passes the test obtained from T_1 by adding a copy of S_i behind S_i.

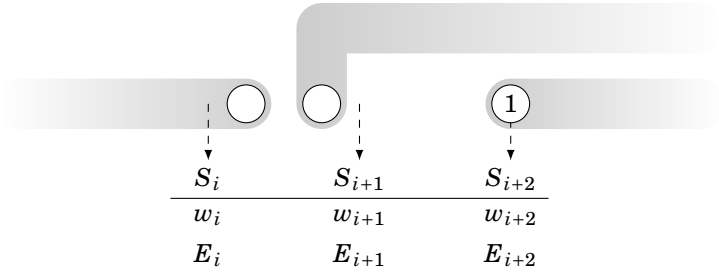


Figure 6.10: Shown are two unlabeled vertices and one labeled vertex of a hypergraph G laid out according to a T-compatible layout for some test $T = (\pi, S, k)$. That is, the label 1 is mapped to the integer position $i+2$ by π, while the others vertices are laid out at non-integer positions. Assume that $S_i = S_{i+1}$ and that no label is mapped to position $i+1$ by π. Then, we can assume that no vertex of G lies in the interval $[i+1, i+2)$: moving it to the interval $(i, i+1)$ would yield a T_1-compatible layout with equal combined cutwidth. Thus, the combined cutwidth is not altered by deleting S_{i+1} or adding copies of S_i behind S_i.

Based on these observations, Downey and Fellows [DF13, Theorem 12.7.5] gave a proof that the following three data reduction rules applied to a load pattern s of the strait $S_j, \ldots, S_k$ turn T_1 into a test T_2 that G passes if and only if it passes T_1:

(R1) If $s = (\ldots, w_i, w_{i+1}, w_{i+2}, \ldots)$ such that $w_i \leq w_{i+1} \leq w_{i+2}$ or $w_i \geq w_{i+1} \geq w_{i+2}$, then delete S_{i+1}.

(R2) If $s = (\ldots, a, s_i, \ldots, s_{i'}, b, \ldots)$ such that each of the numbers $s_i, \ldots, s_{i'}$ is at least $\max(a, b)$, then replace $S_i, \ldots, S_{i'}$ by $S^* := (E, w)$, where w is the maximum of $s_i, \ldots, s_{i'}$.

(R3) If $s = (\ldots, a, s_i, \ldots, s_{i'}, b, \ldots)$ such that each of the numbers $s_i, \ldots, s_{i'}$ is at most $\min(a, b)$, then replace $S_i, \ldots, S_{i'}$ by $S^* := (E, w)$, where w is the minimum of $s_i, \ldots, s_{i'}$.

Downey and Fellows [DF13, Theorem 12.7.5] showed that any load pattern that cannot be reduced with these data reduction rules has length at most $2k+2$. □

6.5 Hypertree width and variants

In in this section, we point out another result that can be shown using our Myhill-Nerode theorem for hypergraphs. One can show that it is not expressible in monadic second-order logic whether a hypergraph has constant generalized hypertree width.

GENERALIZED HYPERTREE WIDTH
Input: A hypergraph $H = (V, E)$ and a natural number k
Question: Does H have generalized hypertree width at most k?

The precise definition of generalized hypertree width is rather technical. We do not provide it here since the only aim of this section is illustrating the applicability of our Myhill-Nerode theorem for hypergraphs. For this purpose, it is enough to know that hypertree width is, similarly to the incidence treewidth parameter we considered so far, a generalization of the treewidth concept to hypergraphs. It might be a promising parameter for use in fixed-parameter algorithms: the generalized hypertree width of a hypergraph is at most its incidence treewidth plus one and can be arbitrarily smaller [FGT09].

A drawback of this parameter is that already checking whether a hypergraph has hypertree width at most three is NP-hard [GMS09]. One could hope to find a fixed-parameter algorithm for GENERALIZED HYPERTREE WIDTH parameterized by incidence treewidth; however, by Proposition 6.2, such a result cannot be obtained using a monadic second-order logic formulation of the problem, since the following can be shown.

Theorem 6.6 ([BFGR14])**.** *Let k-*GHTW *be the set of hypergraphs with generalized hypertree width at most k. The canonical right congruence $\sim_{k\text{-GHTW}}$ has infinite index over $\mathbb{H}_t^{small}$ for $k = 4$ and $t \geq 41$.*

One has to point out that the proof of Theorem 6.6, unlike our Example 6.3, does not rely on trivially exploiting that tree automata cannot count. However, in other respects, the proof of Theorem 6.6 follows a similar approach as Example 6.3: for each $i \geq 1$, we give a construction of a t-boundaried hypergraph $H_i \in \mathbb{H}_t^{\text{small}}$. Then, we show that $H_i \oplus_{\text{h}} H_j$ has generalized hypertree width four if and only if $i = j$. This implies that the canonical right congruence $\sim_{4\text{-HTW}}$ has infinite index over $\mathbb{H}_t^{\text{small}}$. Note that we have not excluded the possibility of a monadic second-order logic expression for having generalized hypertree width at most three.

6.6 Conclusion

We have extended the Myhill-Nerode theorem from graphs to hypergraphs, making the methodology more widely applicable. We applied it to show that HYPERGRAPH CUTWIDTH is fixed-parameter linear. Moreover, one can show that GENERALIZED HYPERTREE WIDTH cannot be expressed in monadic second-order logic.

With our proof that HYPERGRAPH CUTWIDTH is fixed-parameter linear, we pushed open the door to future research: for the applications of HYPERGRAPH CUTWIDTH in, for example, SAT solving, it is crucial to construct a linear layout of minimum cutwidth. Our algorithm does not solve this task, but shows that it can be checked in linear time whether a hypergraph has constant cutwidth k. Therefore, it seems worthwhile searching for a fixed-parameter linear-time algorithm that constructs minimum-cutwidth layouts and, in this way, possibly enables SAT solvers to solve instances that have previously been out of reach.

7 Conclusion and outlook

We investigated to which extent fixed-parameter linear-time algorithms can help to solve NP-hard graph and hypergraph problems arising in industrial applications.

For example, for COLORFUL INDEPENDENT SET WITH LISTS in Chapter 3, we developed a fixed-parameter linear-time algorithm that in a few minutes processes instances of a scale that can model complex scheduling tasks.

In contrast, our algorithm for DAG PARTITIONING in Chapter 4 could not solve the clustering task posed by Leskovec, Backstrom, and Kleinberg [LBK09]. However, it allowed for first insights into the quality of known heuristics.

Our focus on algorithms whose running times grow only linearly in the input size brought aspects into the design of fixed-parameter algorithms that have previously been neglected:

Data structures. The development of efficient algorithms is inevitably tied to the use and development of efficient data structures. However, in fixed-parameter algorithmics, and in exact algorithms for NP-hard problems in general, data structures are often neglected since the algorithms run in exponential time, to which data structures contribute only polynomial factors. For example, the three books on parameterized complexity theory avoid the topic [DF13; FG06; Nie06]. In this thesis, we have seen that the explicit search for fixed-parameter *linear-time* algorithms draws the aspect of efficient data structures into algorithms for NP-hard problems.

Fully multivariate algorithmics. Parameterized complexity theory is often advertised as a multivariate variant of the classical computational complexity theory: instead of measuring problem complexity only in the input size, it measures problem complexity in additional parameters [FJR13; KN12; Nie10]. Herein, the focus often lies on the exponential running time dependence on the additional parameters. Thus, fixed-parameter algorithms are threatened to slip from one univariate extreme—measuring problem complexity only in the

input size—into another univariate extreme—measuring the problem complexity only in the additional parameter. With the explicit search for fixed-parameter linear-time algorithms, we hope to prevent multivariate algorithmics from turning univariate.

Most of our fixed-parameter linear-time algorithms have been developed with the explicit goal of their experimental evaluation. The only exception herein is our algorithm for HYPERGRAPH CUTWIDTH presented in Chapter 6, whose value is in the universality of the approach. The development of fixed-parameter linear-time algorithms in parallel to their experimental evaluation also influenced the theoretical outcomes:

Algorithm simplicity. Most of our presented algorithms are simple. Simple algorithms are easier to understand, easier to implement and, as a result, should contain fewer bugs. However, not all algorithms started out in the simple form presented in this thesis: for example, consider the dynamic programming algorithm for COLORFUL INDEPENDENT SET WITH LISTS presented in Chapter 3. Initially, this algorithm, instead of working on the natural representation of interval graphs—the intervals—worked on their maximal cliques [BMNW12]. As a result, before implementing the algorithm, it was more technical and required a lengthy correctness proof.

Provable speedups. The search for implementation tricks that would yield (not necessarily provable) speedups led to provable speedups: neither our fixed-parameter linear-time algorithm for COLORFUL INDEPENDENT SET WITH LISTS in Chapter 3 nor our fixed-parameter linear-time algorithm for DAG PARTITIONING in Chapter 4 started out as fixed-parameter linear-time algorithms. Instead, their running times could grow quadratically in the input size [Bev+13; BMNW12].

In the following, we want to give some ideas for future developments in the fields of fixed-parameter algorithms and linear-time algorithmics for NP-hard problems.

Data-driven parameterization. The success in applying fixed-parameter algorithms for solving NP-hard problems strongly depends on the choice of the right parameters [FJR13; KN12; Nie10]. Indeed, the reason why our algorithm for DAG PARTITIONING in Chapter 4 cannot cluster the data provided by Leskovec, Backstrom, and Kleinberg [LBK09] is that the parameter "weight k of an optimal

solution," although it is a priori plausible to be small, is not small *enough* on the instances of interest. Thus, for solving large instances occurring in practical applications, the search for small parameters should ideally be based not only on a priori considerations, but, instead, be supported by measuring parameters in the actual application data. In the case of DAG PARTITIONING, we relied on a priori considerations because of the lack of polynomial-time algorithms for approximating the optimal k and because the quality of available heuristics was unknown. Many other parameters in graph data sets, however, can be automatically measured by tools like Graphana developed in our research group.[1]

However, when aiming for *applicable* fixed-parameter algorithms, one should not forget that it is often not enough to have a fixed-parameter algorithm for a problem parameterized by some practically well-motivated parameter: in extreme cases, computing the parameter can be harder than solving the actual problem. For example, formulas occurring in the automatic testing of digital hardware are often generated from Boolean circuits that have small hypergraph cutwidth (often below 20) when transformed into a hypergraph [PCK99]. However, in order take full advantage of this in a fixed-parameter algorithm, Prasad, Chong, and Keutzer [PCK99] require, in terms of the HYPERGRAPH CUTWIDTH problem considered in Chapter 6, a minimum-cutwidth vertex layout for a hypergraph. Finding this layout seems more challenging than solving CNF-SAT itself: in Chapter 6, we have only made the first steps towards the efficient computation of such layouts by showing that the cutwidth of a hypergraph is computable in fixed-parameter linear time.

Linear-time reducibility among problems. All NP-complete problems are reducible to each other using polynomial-time many-one reductions. That is, from the viewpoint of polynomial-time solvability, all NP-complete problems are equally "frightening." On the theoretical side, parameterized complexity theory shows that some NP-complete problems are less frightening than others. Moreover, there are NP-complete problems, large instances of which can often be solved optimally and much faster than one might assume from theoretical bounds. An example is the Boolean satisfiability problem (SAT): there is an annual contest for the fastest SAT solver [BBHJ13] and an annual conference on the sole topic of solving SAT [JG13].

Taking this into account, it seems worthwhile searching for linear-time many-one reductions from NP-hard problems to problems for which extensively en-

[1] http://fpt.akt.tu-berlin.de/graphana/version_2_0/

gineered solvers exist. The race for the fastest many-one reduction from one problem into another is not only a theoretically challenging task, it might also have significant practical impact. A useful tool herein could be linear-time data reduction, as we describe in the following.

Linear-time data reduction. In Chapter 4, we have seen the importance of effective data reduction: without data reduction, none of the DAG PARTITIONING instances in our experiments could be solved optimally. However, in the search for *effective* data reduction, one should not lose out of focus the *time* taken by the data reduction.

Particularly *linear-time* data reduction allowed us to shrink even large instances quickly. Moreover, particularly *fast* kernelization algorithms bring a lot of versatility into the whole concept of problem kernelization:

- As we have seen in Chapter 5 for d-HITTING SET, linear-time kernelization algorithms can be combined with a slower but more effective kernelization algorithm to get the speed and effectiveness of both.
- Linear-time kernelization algorithms can speed up polynomial-time many-one reductions of an NP-complete problems into problems with better engineered solvers like SAT. Additionally, it can ensure that the resulting instance of, say, SAT is small.
- Potentially, linear-time kernelization algorithms could also speed up algorithms for problems in P.

Naturally arises the question for lower bounds: can we show, under reasonable complexity-theoretical assumptions, a lower bound on the sizes of problem kernels computable in linear time? Can we show that computing a problem kernel of certain size for some problem is as hard as sorting, as matrix multiplication, or as maximum matching?

Bibliography

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009 (cited on page 17).

[Abs+14] M. Abseher, B. Bliem, G. Charwat, F. Dusberger, M. Hecher, and S. Woltran. *D-FLAT: Progress Report*. Technical report DBAI-TR-2014-86. Institut für Informationssysteme, Vienna University of Technology, Vienna, Austria, 2014 (cited on pages 29, 112).

[Abu10] F. N. Abu-Khzam. A kernelization algorithm for d-Hitting Set. *Journal of Computer and System Sciences* 76(7), 2010, pages 524–531 (cited on pages 123, 125, 126, 129, 145, 147, 149, 150).

[AF06] F. N. Abu-Khzam and H. Fernau. Kernels: annotated, proper and induced. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation*. Volume 4169 of LNCS. Springer, 2006, pages 264–275 (cited on page 127).

[AF93] K. R. Abrahamson and M. R. Fellows. Finite automata, bounded treewidth, and well-quasiordering. In *Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference on Graph Minors, Graph Structure Theory*. Volume 147 of Contemporary Mathematics. AMS, 1993, pages 539–564 (cited on pages 151, 153, 154).

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983 (cited on pages 61, 136, 138).

[AM12] S. Alamdari and A. Mehrabian. On a DAG partitioning problem. In *Proceedings of the 9th International Workshop on Algorithms and Models for the Web Graph (WAW'12)*. Volume 7323 of LNCS. Springer, 2012, pages 17–28 (cited on pages 84, 85, 87, 88, 98–100, 112, 117).

[AYZ95] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM* 42(4), 1995, pages 844–856 (cited on page 65).

[BA99] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science* 286(5439), 1999, pages 509–512 (cited on page 105).

[Bab53] W. C. Babcock. Intermodulation interference in radio systems. Frequency of occurrence and control by channel selection. *Bell Systems Technical Journal* 32, 1953, pages 63–73 (cited on pages 2, 3).

[Bar+06] R. Bar-Yehuda, M. M. Halldórsson, J. Naor, H. Shachnai, and I. Shapira. Scheduling split intervals. *SIAM Journal on Computing* 36(1), 2006, pages 1–15 (cited on pages 33, 34, 36, 46, 81).

[BBHJ13] A. Balint, A. Belov, M. J. H. Heule, and M. Järvisalo, editors. *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*. Volume B-2013-1 of Department of Computer Science Series of Publications B. University of Helsinki, 2013 (cited on page 189).

[Ben86] J. L. Bentley. *Programming Pearls*. Addison-Wesley, 1986 (cited on page 18).

[Bet+11] N. Betzler, R. van Bevern, C. Komusiewicz, M. R. Fellows, and R. Niedermeier. Parameterized algorithmics for finding connected motifs in biological networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 8(5), 2011, pages 1296–1308 (cited on page xi).

[Bev12] R. van Bevern. Towards optimal and expressive kernelization for d-Hitting Set. In *Proceedings of the 18th Annual International Computing and Combinatorics Conference (COCOON'12)*. Volume 7434 of LNCS. Springer, 2012, pages 121–132 (cited on page xiv).

[Bev14] R. van Bevern. Towards optimal and expressive kernelization for d-Hitting Set. *Algorithmica* 70(1), 2014, pages 129–147 (cited on pages xi, xiv).

[Bev+12] R. van Bevern, S. Hartung, F. Kammer, R. Niedermeier, and M. Weller. Linear-time computation of a linear problem kernel for Dominating Set on planar graphs. In *Proceedings of the 6th International Symposium on Parameterized and Exact Computation (IPEC'11)*. Volume 7112 of LNCS. Springer, 2012, pages 194–206 (cited on pages xi, xiii, 5, 126).

[Bev+13] R. van Bevern, R. Bredereck, M. Chopin, S. Hartung, F. Hüffner, A. Nichterlein, and O. Suchý. Parameterized complexity of DAG Partitioning. In *Proceedings of the 8th International Conference on Algorithms and Complexity (CIAC'13)*. Volume 7878 of LNCS. Springer, 2013, pages 49–60 (cited on pages xi, xiii, 112, 188).

[Bev+14a] R. van Bevern, R. Bredereck, L. Bulteau, J. Chen, V. Froese, R. Niedermeier, and G. J. Woeginger. Star partitions of perfect graphs. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP'14)*. Volume 8572 of LNCS. Springer, 2014, pages 174–185 (cited on page xi).

[Bev+14b] R. van Bevern, R. Bredereck, J. Chen, V. Froese, R. Niedermeier, and G. J. Woeginger. Network-based dissolution. In *Proceedigns of the 39th International Symposium on Mathematical Foundations of Computer Science (MFCS'14)*. Volume 8635 of LNCS. Springer, 2014, pages 69–80 (cited on page xi).

[BF12] L. Brankovic and H. Fernau. Parameterized approximation algorithms for Hitting Set. In *Proceedings of the 9th International Workshop on Approximation and Online Algorithms (WAOA'11)*. Springer, 2012, pages 63–76 (cited on page 125).

[BFGR13] R. van Bevern, M. R. Fellows, S. Gaspers, and F. A. Rosamond. Myhill-Nerode methods for hypergraphs. In *Proceedings of the 24th International Symposium on Algorithms and Computation (ISAAC'13)*. Volume 8283 of LNCS. 2013, pages 372–382 (cited on pages xi, xv).

[BFGR14] R. van Bevern, M. R. Fellows, S. Gaspers, and F. A. Rosamond. Myhill-Nerode methods for hypergraphs, January 2014. arXiv: 1211.1299v4 [cs.DM]. Submitted (cited on pages xv, 160, 184).

[BFSS14] R. van Bevern, A. E. Feldmann, M. Sorge, and O. Suchý. On the parameterized complexity of computing balanced partitions in graphs. *Theory of Computing Systems*, 2014. In press (cited on page xi).

[BFT09] H. L. Bodlaender, M. R. Fellows, and D. M. Thilikos. Derivation of algorithms for cutwidth and related graph layout parameters. *Journal of Computer and System Sciences* 75(4), 2009, pages 231–244 (cited on page 154).

[BFW92] H. L. Bodlaender, M. R. Fellows, and T. J. Warnow. Two strikes against perfect phylogeny. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP'92)*. Volume 623 of LNCS. Springer, 1992, pages 273–283 (cited on page 151).

[BHNS14] R. van Bevern, S. Hartung, A. Nichterlein, and M. Sorge. Constant-factor approximations for Capacitated Arc Routing without triangle inequality. *Operations Research Letters* 42(4), 2014, pages 290–292 (cited on page xi).

[Bie13] A. Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*. Volume B-2013-1 of Department of Computer Science Series of Publications B. University of Helsinki, 2013, pages 51–52 (cited on page 2).

[BJK14] H. L. Bodlaender, B. M. P. Jansen, and S. Kratsch. Kernelization lower bounds by cross-composition. *SIAM Journal on Discrete Mathematics* 28(1), 2014, pages 277–305 (cited on pages 25, 26).

[BKMN10] R. van Bevern, C. Komusiewicz, H. Moser, and R. Niedermeier. Measuring indifference: Unit Interval Vertex Deletion. In *Proceedings of the 36th International Workshop on Graph Theoretic Concepts in Computer Science (WG'10)*. Volume 6410 of LNCS. Springer, 2010, pages 232–243 (cited on page xi).

[BLS99] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: A Survey*. SIAM, 1999 (cited on pages 17, 71, 77).

[BMN12] R. van Bevern, H. Moser, and R. Niedermeier. Approximation and tidying—a problem kernel for s-Plex Cluster Vertex Deletion. *Algorithmica* 62(3-4), 2012, pages 930–950 (cited on page xi).

[BMNW12] R. van Bevern, M. Mnich, R. Niedermeier, and M. Weller. Interval scheduling and colorful independent sets. In *Proceedings of the 23rd International Symposium on Algorithms and Computation (ISAAC'12)*. Volume 7676 of LNCS. 2012, pages 247–256 (cited on pages xi, xii, 188).

[BMNW14] R. van Bevern, M. Mnich, R. Niedermeier, and M. Weller. Interval scheduling and colorful independent sets. *Journal of Scheduling*, 2014. Accepted for publication (cited on page xii).

[BNR96] V. Bafna, B. Narayanan, and R. Ravi. Nonoverlapping local alignments (weighted independent sets of axis-parallel rectangles). *Discrete Applied Mathematics* 71(1-3), 1996, pages 41–53 (cited on pages 36, 81).

[BNSW14] R. van Bevern, R. Niedermeier, M. Sorge, and M. Weller. Complexity of arc routing problems. In *Arc Routing: Problems, Methods, and Applications*. Edited by Á. Corberán and G. Laporte. SIAM, 2014. In press (cited on page xi).

[Bod09] H. L. Bodlaender. Kernelization: new upper and lower bound techniques. In *Proceedings of the 4th International Workshop on Parameterized and Exact Computation (IWPEC'09)*. Volume 5917 of LNCS. Springer, 2009, pages 17–37 (cited on page 4).

[Bod96] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing* 25(6), 1996, pages 1305–1317 (cited on pages 27, 157).

[CE12] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic—A Language-Theoretic Approach*. Volume 138 of Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2012 (cited on pages 28, 29, 154, 156, 162, 170, 172).

[CFJ05] B. Chor, M. Fellows, and D. Juedes. Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps. In *Proceedings of the 30th International Conference on Graph-Theoretic Concepts in Computer Science (WG'04)*. Volume 3353 of LNCS. Springer, 2005, pages 257–269 (cited on page 5).

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 2nd edition. MIT Press, 2001 (cited on pages 55, 91, 97, 137, 139).

[CLSZ07] J. Chen, S. Lu, S.-H. Sze, and F. Zhang. Improved algorithms for path, matching, and packing problems. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*. SIAM, 2007, pages 298–307 (cited on page 66).

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC'71)*. ACM, 1971, pages 151–158 (cited on page 1).

[COR06] J. Chuzhoy, R. Ostrovsky, and Y. Rabani. Approximation algorithms for the job interval selection problem and related scheduling problems. *Mathematics of Operations Research* 31(4), 2006, pages 730–738 (cited on pages 36, 37, 81).

[COS09] D. G. Corneil, S. Olariu, and L. Stewart. The LBFS structure and recognition of interval graphs. *SIAM Journal on Discrete Mathematics* 23(4), 2009, pages 1905–1953 (cited on pages 54, 55).

[COS11] M. R. Cerioli, F. d. S. Oliveira, and J. L. Szwarcfiter. On counting interval lengths of interval graphs. *Discrete Applied Mathematics* 159(7), 2011, pages 532–543 (cited on page 53).

[Cyg+14] M. Cygan, D. Lokshtanov, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. On cutwidth parameterized by vertex cover. *Algorithmica* 68(4), 2014, pages 940–953 (cited on page 154).

[Dah+94] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiterminal cuts. *SIAM Journal on Computing* 23(4), 1994, pages 864–894 (cited on pages 85, 118).

[Dam06] P. Damaschke. Parameterized enumeration, transversals, and imperfect phylogeny reconstruction. *Theoretical Computer Science* 351(3), 2006, pages 337–350 (cited on pages 125, 128, 129).

[DF13] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013 (cited on pages xiv, 22, 23, 32, 151, 152, 157, 160, 162, 170, 176, 182, 183, 187).

[DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999 (cited on pages 4, 5, 162).

[Die10] R. Diestel. *Graph Theory*. 4th edition. Volume 173 of Graduate Texts in Mathematics. Springer, 2010 (cited on page 10).

[Dim02] A. Dimitromanolakis. Analysis of the Golomb ruler and the Sidon set problems, and determination of large, near-optimal Golomb rulers. Diploma thesis. Department of Electronic and Computer Engineering, Technical University of Crete, 2002 (cited on page 141).

[DM14] H. Dell and D. van Melkebeek. Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. *Journal of the ACM*, 2014. Accepted for publication (cited on page 124).

[Dra09] K. Drakakis. A review of the available construction methods for Golomb rulers. *Advances in Mathematics of Communications* 3(3), 2009, pages 235–250 (cited on pages 2, 124, 140).

[ER60] P. Erdős and R. Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society* 35(1), 1960, pages 85–90 (cited on pages 126, 130, 134).

[ET41] P. Erdős and P. Turán. On a problem of Sidon in additive number theory, and on some related problems. *Journal of the London Mathematical Society* 16(4), 1941, pages 212–215 (cited on page 2).

[FBNS13] V. Froese, R. van Bevern, R. Niedermeier, and M. Sorge. A parameterized complexity analysis of combinatorial feature selection problems. In *Proceedings of the 38th International Symposium on Mathematical Foundations of Computer Science (MFCS'13)*. Volume 8087 of LNCS. Springer, 2013, pages 445–456 (cited on page xi).

[Fer06] H. Fernau. Edge Dominating Set: efficient enumeration-based exact algorithms. In *Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC'06)*. Volume 4169 of LNCS. Springer, 2006, pages 142–153 (cited on page 125).

[Fer10] H. Fernau. Parameterized algorithms for d-Hitting Set: the weighted case. *Theoretical Computer Science* 411(16-18), 2010, pages 1698–1713 (cited on page 125).

[FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006 (cited on pages 22, 23, 123–126, 130, 134, 187).

[FG65] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics* 15(3), 1965, pages 835–855 (cited on page 54).

[FGR12] M. R. Fellows, S. Gaspers, and F. A. Rosamond. Parameterizing by the number of numbers. *Theory of Computing Systems* 50(4), 2012, pages 675–693 (cited on pages 37, 53).

[FGT09] F. V. Fomin, P. A. Golovach, and D. M. Thilikos. Approximating acyclicity parameters of sparse hypergraphs. In *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS'09)*. Volume 3 of LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2009, pages 445–456 (cited on page 184).

[FJR13] M. R. Fellows, B. M. Jansen, and F. Rosamond. Towards fully multivariate algorithmics: Parameter ecology and the deconstruction of computational complexity. *European Journal of Combinatorics* 34(3), 2013, pages 541–566 (cited on pages 4, 187, 188).

[FK14] S. Fafianie and S. Kratsch. Streaming kernelization. In *Proceedings of the 39th International Symposium on Mathematical Foundations of Computer Science (MFCS'14)*. Volume 8635 of LNCS. Springer, 2014, pages 275–286 (cited on pages 125, 126).

[FL92] M. R. Fellows and M. A. Langston. On well-partial-order theory and its application to combinatorial problems of VLSI design. *SIAM Journal on Discrete Mathematics* 5(1), 1992, pages 117–126 (cited on page 154).

[FL94] M. R. Fellows and M. A. Langston. On search, decision, and the efficiency of polynomial-time algorithms. *Journal of Computer and System Sciences* 49(3), 1994, pages 769–779 (cited on page 154).

[FSV13] F. V. Fomin, S. Saurabh, and Y. Villanger. A polynomial kernel for Proper Interval Vertex Deletion. *SIAM Journal on Discrete Mathematics* 27(4), 2013, pages 1964–1976 (cited on pages xiv, 123, 126, 128, 130).

[Gav77] F. Gavril. Some NP-complete problems on graphs. In *Proceedings of the 11th Conference on Information Sciences and Systems*. Johns Hopkins Univeristy, Baltimore, USA, 1977, pages 91–95 (cited on pages 154, 173).

[GGJK78] M. R. Garey, R. L. Graham, D. S. Johnson, and D. E. Knuth. Complexity results for bandwidth minimization. *SIAM Journal on Applied Mathematics* 34(3), 1978, pages 477–495 (cited on page 153).

[GH10] R. Ganian and P. Hliněný. On parse trees and Myhill-Nerode-type tools for handling graphs of bounded rank-width. *Discrete Applied Mathematics* 158(7), 2010, pages 851–867 (cited on page 154).

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979 (cited on pages 3, 17, 22).

[GJS76] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science* 1(3), 1976, pages 237–267 (cited on page 47).

[GMS09] G. Gottlob, Z. Miklós, and T. Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *Journal of the ACM* 56(6), 2009, article 30 (cited on pages 154, 184).

[GN07] J. Guo and R. Niedermeier. Invitation to data reduction and problem kernelization. *ACM SIGACT News* 38(1), 2007, pages 31–45 (cited on page 4).

[GNNW13] S. Gaspers, V. Naroditskiy, N. Narodytska, and T. Walsh. Possible and necessary winner problem in social polls. In *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'13)*. IFAAMAS, 2013, pages 1131–1132 (cited on page 151).

[GW95] A. Gyárfas and D. B. West. Multitrack interval graphs. *Congressus Numerantium* 109, 1995, pages 109–116 (cited on page 36).

[Hag12] T. Hagerup. Simpler linear-time kernelization for Planar Dominating Set. In *Proceedings of the 6th International Symposium on Parameterized and Exact Computation (IPEC'11)*. Volume 7112 of LNCS. Springer, 2012, pages 181–193 (cited on pages 5, 126).

[HK06] M. M. Halldórsson and R. K. Karlsson. Strip graphs: recognition and scheduling. In *Proceedings of the 32nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG'06)*. Volume 4271 of LNCS. 2006, pages 137–146 (cited on pages xii, 33, 34, 37, 38, 42, 44, 46, 57, 59–63).

[HKML11] W. Höhn, F. G. König, R. H. Möhring, and M. E. Lübbecke. Integrated sequencing and scheduling in coil coating. *Management Science* 57(4), 2011, pages 647–666 (cited on pages 1, 3, 33, 34, 36, 80–82).

[Hli06] P. Hliněný. Branch-width, parse trees, and monadic second-order logic for matroids. *Journal of Combinatorial Theory, Series B* 96(3), 2006, pages 325–351 (cited on page 154).

[IPZ01] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences* 63(4), 2001, pages 512–530 (cited on pages 52, 53, 100).

[JG13] M. Järvisalo and A. V. Gelder, editors. *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing*. Volume 7962 of LNCS. Springer, 2013 (cited on page 189).

[Jia10] M. Jiang. On the parameterized complexity of some optimization problems related to multiple-interval graphs. *Theoretical Computer Science* 411(49), 2010, pages 4253–4262 (cited on pages 36, 38, 67, 72, 74).

[Jia13] M. Jiang. Recognizing d-interval graphs and d-track interval graphs. *Algorithmica* 66(3), 2013, pages 541–563 (cited on page 36).

[JNB03] H. Jeong, Z. Néda, and A. L. Barabási. Measuring preferential attachment in evolving networks. *Europhysics Letters* 61(4), 2003, pages 567–572 (cited on pages 105, 106).

[Kam13] F. Kammer. A linear-time kernelization for the Rooted k-Leaf Outbranching Problem. In *Proceedings of the 39th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'13)*. Volume 8165 of LNCS. Springer, 2013, pages 310–320 (cited on page 126).

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*. The IBM Research Symposia Series. Springer, 1972, pages 85–103 (cited on pages 2, 131).

[Kar+04] D. R. Karger, P. Klein, C. Stein, M. Thorup, and N. E. Young. Rounding algorithms for a geometric embedding of Minimum Multiway Cut. *Mathematics of Operations Research* 29(3), 2004, pages 436–461 (cited on pages 85, 88).

[Kas08] N. Kashyap. Matroid pathwidth and code trellis complexity. *SIAM Journal on Discrete Mathematics* 22(1), 2008, pages 256–272 (cited on page xiv).

[KK91] A. W. J. Kolen and L. G. Kroon. On the computational complexity of (maximum) class scheduling. *European Journal of Operational Research* 54(1), 1991, pages 23–38 (cited on page 40).

[KK93] A. W. J. Kolen and L. G. Kroon. On the computational complexity of (maximum) shift class scheduling. *European Journal of Operational Research* 64(1), 1993, pages 138–151 (cited on page 40).

[KK94] A. W. J. Kolen and L. G. Kroon. An analysis of shift class design problems. *European Journal of Operational Research* 79(3), 1994, pages 417–430 (cited on page 40).

[Klo94] T. Kloks. *Treewidth. Computations and Approximations*. Volume 842 of LNCS. Springer, 1994 (cited on page 27).

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM* 22(4), 1975, pages 469–476 (cited on page 75).

[KLPS07] A. W. J. Kolen, J. K. Lenstra, C. H. Papadimitriou, and F. C. R. Spieksma. Interval scheduling: a survey. *Naval Research Logistics* 54(5), 2007, pages 530–543 (cited on pages 33, 34).

[KLR11] J. Kneis, A. Langer, and P. Rossmanith. Courcelle's theorem—a game-theoretic approach. *Discrete Optimization* 8(4), 2011, pages 568–594 (cited on page 29).

[KN12] C. Komusiewicz and R. Niedermeier. New races in parameterized algorithmics. In *Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS'12)*. Volume 7464 of LNCS. Springer, 2012, pages 19–30 (cited on pages 4, 5, 187, 188).

[KR08] S. Khot and O. Regev. Vertex cover might be hard to approximate to within $2-\varepsilon$. *Journal of Computer and System Sciences* 74(3), 2008, pages 335–349 (cited on page 125).

[Kra12] S. Kratsch. Polynomial kernelizations for MIN $\mathrm{F}^{+}\Pi_1$ and MAX NP. *Algorithmica* 63(1-2), 2012, pages 532–550 (cited on pages xiv, 123, 125–127, 130).

[Kra14] S. Kratsch. Recent developments in kernelization: A survey. *Bulletin of the European Association for Theoretical Computer Science* 113, 2014, pages 58–97 (cited on page 4).

[KV00] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences* 61(2), 2000, pages 302–332 (cited on page 153).

[KW09] I. Koutis and R. Williams. Limits and applications of group algebras for parameterized problems. In *Proceedings of the 36th International Colloquium on Automata, Languages, and Programming (ICALP'09)*. Volume 5555. LNCS. Springer, 2009, pages 653–664 (cited on pages 66, 82).

[KW87] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence* 32(1), 1987, pages 97–130 (cited on pages 123, 124).

[LBK09] J. Leskovec, L. Backstrom, and J. Kleinberg. Meme-tracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'09)*. ACM, 2009, pages 497–506 (cited on pages xiii, 1, 3, 83–87, 91, 105–112, 118, 122, 187, 188).

[Lev73] L. A. Levin. Универсальные задачи перебора (Universal'nye zadači perebora). *Проблемы передачи информации (Problemy peredači informacii)* 9(3), 1973, pages 115–116 (cited on page 1). In Russian. Translation in B. A. Trakhtenbrot. A survey of Russian approaches to Perebor (brute-force search) algorithms. *IEEE Annals of the History of Computing* 6(4), 1984, pages 384–400.

[LMS11] D. Lokshtanov, D. Marx, and S. Saurabh. Lower bounds based on the Exponential Time Hypothesis. *Bulletin of the European Association for Theoretical Computer Science* 105, 2011, pages 41–72 (cited on pages 52, 100).

[Loh09] S. Lohr. Study measures the chatter of the news cycle. *New York Times*, New York, July 13, 2009, page B1 (cited on pages 1, 83).

[MK13] E. Moreno-Centeno and R. M. Karp. The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. *Operations Research* 61(2), 2013, pages 453–468 (cited on page 2).

[MMNS11] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review* 5(2), 2011, pages 119–161 (cited on page 128).

[Möh11] R. H. Möhring. Algorithm engineering and industrial applications. *it – Information Technology* 53(6), 2011, pages 302–311 (cited on page 1).

[Mos10] H. Moser. Finding Optimal Solutions for Covering and Matching Problems. Dissertation. Institut für Informatik, Friedrich-Schiller-Universität Jena, 2010 (cited on pages 123, 125–127, 129, 145–147, 149, 150).

[MS91] Z. Miller and I. H. Sudborough. A polynomial algorithm for recognizing bounded cutwidth in hypergraphs. *Mathematical Systems Theory* 24(1), 1991, pages 11–40 (cited on page 154).

[Myh57] J. Myhill. *Finite automata and representation of events*. Technical report WADD TR-57-624. Wright-Patterson Air Force Base, Ohio, USA, 1957, pages 122–137 (cited on pages 154–156).

[Nag12] H. Nagamochi. Linear layouts in submodular systems. In *Proceedings of the 23rd International Symposium on Algorithms and Computation (ISAAC'12)*. Volume 7676 of LNCS. Springer, 2012, pages 475–484 (cited on page 154).

[Ner58] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society* 9(4), 1958, pages 541–544 (cited on pages 154–156).

[NH82] K. Nakajima and S. L. Hakimi. Complexity results for scheduling tasks with discrete starting times. *Journal of Algorithms* 3(4), 1982, pages 344–361 (cited on page 36).

[Nie06] R. Niedermeier. *Invitation to Fixed Parameter Algorithms*. Oxford University Press, 2006 (cited on pages 22, 23, 187).

[Nie10] R. Niedermeier. Reflections on multivariate algorithmics and problem parameterization. In *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science (STACS'10)*. Volume 5 of LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010, pages 17–32 (cited on pages 4, 187, 188).

[NR00] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters* 73(3-4), 2000, pages 125–129 (cited on page 97).

[NR03] R. Niedermeier and P. Rossmanith. An efficient fixed-parameter algorithm for 3-Hitting Set. *Journal of Discrete Algorithms* 1(1), 2003, pages 89–102 (cited on pages xiii, 125, 126, 150).

[NRT04] N. Nishimura, P. Ragde, and D. M. Thilikos. Smaller kernels for Hitting Set problems of constant arity. In *Proceedings of the 1st International Workshop on Parameterized and Exact Computation (IWPEC'04)*. Volume 3162 of LNCS. Springer, 2004, pages 121–126 (cited on page 125).

[OC03] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*. ACM, 2003, pages 167–178 (cited on pages 2, 3, 123, 124).

[Pap94] C. M. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994 (cited on pages 17, 18, 20).

[PCK99] M. R. Prasad, P. Chong, and K. Keutzer. Why is ATPG easy? In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference (DAC'99)*. ACM, 1999, pages 22–28 (cited on pages 2, 3, 154, 173, 189).

[PDS09] F. Protti, M. Dantas da Silva, and J. Szwarcfiter. Applying modular decomposition to parameterized cluster editing problems. *Theory of Computing Systems* 44(1), 2009, pages 91–104 (cited on pages 5, 126).

[Pri76] D. D. S. Price. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American Society for Information Science* 27(5), 1976, pages 292–306 (cited on page 105).

[Rei87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence* 32(1), 1987, pages 57–95 (cited on pages 123, 124).

[SBNW11] M. Sorge, R. van Bevern, R. Niedermeier, and M. Weller. From few components to an Eulerian graph by adding arcs. In *Proceedings of the 37th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'11)*. Volume 6986 of LNCS. Springer, 2011, pages 307–318 (cited on page xi).

[SBNW12] M. Sorge, R. van Bevern, R. Niedermeier, and M. Weller. A new view on Rural Postman based on Eulerian Extension and matching. *Journal of Discrete Algorithms* 16, 2012, pages 12–33 (cited on page xi).

[SC10] L. Shi and X. Cai. An exact fast algorithm for Minimum Hitting Set. In *Proceedings of the 3rd International Joint Conference on Computational Science and Optimization (CSO'10)*. IEEE Computer Society, 2010, pages 64–67 (cited on page 125).

[Sch03] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Volume A. Springer, 2003 (cited on pages 47, 150).

[Sch88] E. Scheinerman. Random interval graphs. *Combinatorica* 8(4), 1988, pages 357–371 (cited on page 78).

[SMNW14] M. Sorge, H. Moser, R. Niedermeier, and M. Weller. Exploiting a hypergraph model for finding Golomb rulers. *Acta Informatica*, 2014. In press (cited on pages 2, 3, 123, 124, 140, 144).

[Spi99] F. C. R. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling* 2(5), 1999, pages 215–227 (cited on pages 34, 36, 37, 42, 81).

[SS10] M. Samer and S. Szeider. Constraint satisfaction with bounded treewidth revisited. *Journal of Computer and System Sciences* 76(2), 2010, pages 103–114 (cited on page 153).

[Sue+13] C. Suen, S. Huang, C. Eksombatchai, R. Sosic, and J. Leskovec. NIFTY: a system for large scale information flow tracking and clustering. In *Proceedings of the 22nd International Conference on World Wide Web (WWW'13)*. IW3C2, 2013, pages 1237–1248 (cited on pages 1, 3, 84, 87, 91, 105, 122).

[TSB05] D. M. Thilikos, M. Serna, and H. L. Bodlaender. Cutwidth I: a linear time fixed parameter algorithm. *Journal of Algorithms* 56(1), 2005, pages 1–24 (cited on pages 153, 154).

[WCZK01] D. Wang, E. Clarke, Y. Zhu, and J. Kukula. Using cutwidth to improve symbolic simulation and Boolean satisfiability. In *Proceedings of the 6th IEEE International High-Level Design Validation and Test Workshop (HLDVT'01)*. IEEE, 2001, pages 165–170 (cited on pages 2–4, 154, 173, 174).

[Wes01] D. B. West. *Introduction to Graph Theory*. 2nd edition. Prentice Hall, 2001 (cited on page 10).

[Xia10] M. Xiao. Simple and improved parameterized algorithms for multiterminal cuts. *Theory of Computing Systems* 46(4), 2010, pages 723–736 (cited on pages 85, 88).

Universitätsverlag der TU Berlin

Fixed-Parameter Linear-Time Algorithms for NP-hard Graph and Hypergraph Problems Arising in Industrial Applications

This thesis aims for the development of efficient algorithms to exactly solve four selected NP-hard graph and hypergraph problems arising in the fields of scheduling, steel manufacturing, software engineering, radio frequency allocation, computer-aided circuit design, and social network analysis.

NP-hard problems presumably cannot be solved exactly in a running time growing only polynomially with the input size. In order to still solve the considered problems efficiently, this thesis develops linear-time data reduction and fixed-parameter linear-time algorithms—algorithms that can be proven to run in linear time if certain parameters of the problem instances are constant.

Besides proving linear worst-case running times, the efficiency of most of the developed algorithms is evaluated experimentally. Moreover, the limits of fixed-parameter linear-time algorithms and provably efficient and effective data reduction are shown.

ISBN 978-3-7983-2705-4 (print)
ISBN 978-3-7983-2706-1 (online)