FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Tailored Tree Decompositions for Efficient Problem Solving

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

DI Michael Abseher, BSc
Matrikelnummer 0828282

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Co-Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Nysret Musliu

Diese Dissertation haben begutachtet:

Prof. Dr. Rolf Niedermeier	Prof. Luca Di Gaspero, PhD

Wien, 5. März 2017

Michael Abseher

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Tailored Tree Decompositions for Efficient Problem Solving

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

DI Michael Abseher, BSc
Registration Number 0828282

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Co-Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Nysret Musliu

The dissertation has been reviewed by:

Prof. Dr. Rolf Niedermeier	Prof. Luca Di Gaspero, PhD

Vienna, 5th March, 2017

Michael Abseher

Erklärung zur Verfassung der Arbeit

DI Michael Abseher, BSc
Walpersbach 198, A-2822 Walpersbach

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. März 2017

Michael Abseher

Danksagung

Zuallererst möchte ich meinen beiden Betreuern, Stefan Woltran und Nysret Musliu, für ihre exzellente Unterstützung danken. Die Förderung, die ich durch sie erfahren durfte, sowie auch ihre Expertise, waren von unschätzbarem Wert für den Fortschritt dieser Arbeit. Es war immer eine Freude, mit ihnen zu arbeiten und ich bin sehr dankbar für ihre hilfreichen Ratschläge und die geduldige Anleitung in den vergangenen Jahren.

Ebenfalls möchte ich meine große Dankbarkeit gegenüber allen Mitarbeiterinnen und Mitarbeitern der Arbeitsgruppe für Datenbanken und Künstliche Intelligenz am Institut für Informationssysteme der Technischen Universität Wien ausdrücken. Die Zusammenarbeit mit ihnen war großartig. An diesem Punkt möchte ich speziell Günther Charwat danken. Durch seine vielen Ideen zur Verbesserung und Erweiterung der für diese Arbeit entwickelten Programmbibliothek, sowie des Testens selbiger, hat er einen wichtigen Beitrag zu deren Umsetzung geleistet.

Ich bin meinen Eltern zu tiefster Dankbarkeit verpflichtet. Sie haben mich in den letzten 28 Jahren durch alle Höhen und Tiefen des Lebens begleitet. Selbst in schweren Zeiten waren sie immer für mich da und von meiner Kindheit an haben sie mich immer darin bestärkt, niemals etwas aufzugeben, was einmal begonnen wurde. Ohne ihre Unterstützung und Geduld wäre diese Arbeit nicht möglich gewesen.

Weiters möchte ich auch all meinen Freundinnen, Freunden, Kolleginnen und Kollegen danken, die mein Leben jeden Tag aufs Neue bereichern. In diesem Zusammenhang möchte ich speziell meine beiden besten Freundinnen, Theresa Rasinger und Nicole Wagner, hervorheben. Die Mountainbiketouren mit Theresa waren immer lustig und halfen mir, den Kopf frei zu bekommen für anstehende Publikationstätigkeiten. Auch möchte ich keinen einzigen Tag mit Nicole vermissen, denn sie bringt mich immer zum Lachen und vermutlich ist sie meine Seelenverwandte. Mit jeder meiner beiden besten Freundinnen durfte (und darf) ich Jahre voll Freude, Abenteuern und unvergesslichen Momenten genießen. Mögen diese Zeiten niemals enden.

Zu guter Letzt möchte ich Julia Laubender dafür danken, dass sie mich an jedem einzelnen Tag der letzten Jahre immer mit der notwendigen Portion Motivation ausgestattet hat. Durch ihre unvergleichliche, wundervolle Art nimmt Julia eine spezielle Position in meinem Leben ein.

Acknowledgements

First of all, I want to thank my advisors, Stefan Woltran and Nysret Musliu, for their excellent support. Their encouragement as well as their expertise were invaluable for the progress of this thesis. It was always a pleasure to work with them and I am very grateful for their helpful advice and patient guidance throughout the last years.

Further, I also want to express my gratitude to all the members of the DBAI group of the Institute of Information Systems at TU Wien. I had a great time working with them. Many thanks, at this point, to Günther Charwat for constantly providing me with new ideas and suggestions regarding the software library I developed for this thesis and for carefully testing all of its features.

I am deeply grateful to my parents for their support and patience during the past 28 years. Even in hard times, the always have been there for me and, from my childhood on, they inspired me to never give up anything I have started. Without them, this work would not have been possible.

Furthermore, I thank all my friends and colleagues for enriching my life. In this context, I want to pay special tribute to my two very best friends, Theresa Rasinger and Nicole Wagner. The mountain bike tours with Theresa were always fun and they helped me to keep my head free in order to be prepared for upcoming publication tasks. Also, I do not want to miss a single day with Nicole because she always knows how to make me laugh and she is probably my soul mate. With both of my best friends, I enjoy(ed) years full of fun, adventures and unforgettable moments. May these times never end.

Last but not least, I would like to thank Julia Laubender for always providing me with the right amount of motivation on all days in the last couple of years. Because of her unmatched, marvelous nature, she takes a special position in my life.

Kurzfassung

Zerlegungen von Graphen und Hypergraphen spielen eine wichtige Rolle im Bereich der Forschung zum Thema “Parametrisierte Komplexität”. Baumzerlegungen stellen in diesem Zusammenhang ein wichtiges Konzept dar, da – sofern die Probleminstanzen gewissen Eigenschaften genügen – dynamische Programmierung basierend auf diesen Zerlegungen eine gängige Methode zur effizienten Lösung einer Vielzahl von Problemen ist, obwohl diese Probleme aus Sicht der grundlegenden Komplexitätstheorie oftmals als inhärent schwer gelten.

Hierbei ist der Parameter *Baumweite* von entscheidender Bedeutung. Genauer gesagt ist, aus theoretischer Sicht, die *Weite* der verwendeten Zerlegung einer Probleminstanz der essenzielle Parameter für die Laufzeit eines Algorithmus, welcher das Konzept der dynamischen Programmierung auf Baumzerlegungen implementiert. Dennoch ist es in der Praxis oft der Fall, dass diese Algorithmen sehr sensibel auf die verwendeten Zerlegungen reagieren. Das zeigt sich unter anderem dadurch, dass, obwohl zwei Zerlegungen von der gleichen Probleminstanz stammen und sie die gleiche Weite aufweisen, die Laufzeit des Algorithmus oftmals stark unterschiedlich ausfällt, abhängig davon welche der beiden Zerlegungen schlussendlich verwendet wird.

Daher ist eindeutig erwiesen dass in der Praxis die Qualität einer Baumzerlegung nicht allein aufgrund ihrer Weite bestimmt werden kann. Vielmehr müssen auch die weitere Gestalt der Zerlegung und der konkrete Algorithmus, in dem diese verwendet wird, für die Beurteilung der Qualität herangezogen werden. Daher stellen wir in dieser Arbeit das Konzept der sogenannten *Maßgeschneiderten Baumzerlegungen* vor. Darunter sind Baumzerlegungen zu verstehen, die entsprechend spezieller, benutzerdefinierter Kriterien modifiziert und optimiert wurden.

Diese Abschlussarbeit stellt zwei Ansätze vor, um sowohl Effizienz als auch Robustheit von Algorithmen basierend auf dynamischer Programmierung zu steigern. Der erste Ansatz verwendet Techniken aus dem Bereich des maschinellen Lernens zur Vorhersage der Laufzeit von Algorithmen unter Berücksichtigung der konkreten Baumzerlegung die diese verwenden. Zu diesem Zweck identifizieren wir in dieser Arbeit eine Vielzahl an Parametern zur Charakterisierung von Baumzerlegungen. Ausführliche Experimente, welche ebenfalls in dieser Arbeit vorgestellt werden, zeigen, dass die Effizienz von Algorithmen, die dem Konzept der dynamischen Programmierung auf Baumzerlegungen entsprechen, signifikant gesteigert werden kann, wenn diese auf Baumzerlegungen, welche sich auf Basis der

Analyse mittels maschinellem Lernen als vorteilhaft herausstellen, operieren. Der zweite Ansatz verfolgt die Idee, Algorithmen direkt mit solch vorteilhaften Baumzerlegungen zu versorgen ohne den Umweg über eine Vorselektion basierend auf maschinellem Lernen. Daher präsentieren wir in dieser Arbeit auch ein freies, quelloffenes Softwareframework für die effiziente Berechnung von maßgeschneiderten Baumzerlegungen. Diese flexibel erweiterbare Programmbibliothek erlaubt die Generierung von Zerlegungen welche exakt auf den jeweiligen Algorithmus zugeschnitten sind. Auch in den Experimenten bezüglich dieses zweiten Ansatzes zeigt sich ein signifikanter, positiver Einfluss auf die Laufzeit von Algorithmen wenn diese maßgeschneiderte Baumzerlegungen verwenden.

Abstract

Decompositions of graphs and hypergraphs play a central role in the field of parameterized complexity theory. Tree decompositions are a prominent concept in this context since – given that instances enjoy certain structural properties – dynamic programming on tree decompositions allows to solve many computational problems efficiently although they are intractable in the general case. This is because tree decompositions are the basis for many fixed-parameter tractable algorithms for solving NP-hard problems.

From a theoretical point of view, the parameter *treewidth* is crucial for the efficiency of such algorithms. To be more precise, the *width* of the tree decomposition actually used is assumed to be a key ingredient towards performance. However, experience shows that dynamic programming algorithms often exhibit a high runtime variance when using different tree decompositions; in fact, given an instance of the problem at hand, even decompositions of the same width might yield extremely diverging solving times. This means that, besides the width there must be other features of tree decompositions which affect the performance of dynamic programming algorithms.

Hence, in practice, the quality of a tree decomposition cannot be judged without taking its shape and the concrete algorithm in which the decomposition is used into account. We thus propose in this thesis the concept of what we call *customized tree decompositions*, i.e., tree decompositions which reflect certain preferences with regard to custom quality criteria.

We present in this work two approaches which allow to reliably boost both efficiency and robustness of dynamic programming algorithms. The first approach employs techniques from the area of machine learning. We identify a large set of tree decomposition features and use machine learning for predicting the runtime of dynamic programming algorithms based on these features. Extensive experiments conducted in this context underline that customized tree decompositions obtained by selecting promising decompositions according to this strategy are highly beneficial. The second approach then aims for the fast computation of such beneficial tree decompositions. For this reason we present here a flexible and efficient software framework for computing graph decompositions which allows to easily obtain decompositions perfectly tailored towards the algorithm in which they are used. Also in the experiments concerning this second approach, the use of customized tree decompositions significantly improves the performance of the dynamic programming algorithms.

Contents

CHAPTER 1

Introduction

Although they are not always recognized as such, graphs are omnipresent in everyday life. Basically, any situation in which there exist specific connections between certain objects can be seen and analyzed from a graph-theoretic perspective. For instance, road and public transport networks, computer networks as well as social networks all can be interpreted as graphs. Strictly speaking, any imaginable example of a network represents in principle a graph.

Indeed, the fact that graphs allow to provide abstractions of real-world networks is not an end in itself. Moreover, many tasks which occur in practical application scenarios are reducible to computational problems defined on graphs. For this reason, graphs are probably one of the most important concepts in computer science.

Fulfilling a specific task, like, e.g., "Given a geographical map and a list of cities, find the shortest combination of transportation routes such that all the cities are connected!", often consumes a lot of time especially when the search space, i.e., the number of possibilities to choose from, is large. A prominent approach to boost the performance of algorithms for solving such computational problems is based on so-called *graph decompositions*.

Graph decompositions are an important concept in the field of parameterized complexity theory. As the name suggests, the term refers to the decomposition of the input graph into a number of smaller parts which are assumed to be easier to handle than the whole graph at once. A wide variety of approaches for graph decomposition can be found in the literature including tree decompositions [BB73, Hal76, RS84], branch decompositions [RS91], and hypertree decompositions [GLS02] (of hypergraphs), to mention just a few.

The concept of tree decompositions gained special attention since many NP-hard search problems become tractable when the parameter *treewidth* (which refers to the minimum

width[1] over all possible tree decompositions of the given problem instance) is bounded by some constant k [AP89, Nie06, BK08]. A problem which exhibits tractability by bounding a problem-inherent constant is also called fixed-parameter tractable (FPT) [DF99].

A promising technique for solving problems on the basis of structural graph decompositions is the computation of a tree decomposition followed by a dynamic programming (DP) algorithm which traverses the nodes of the decomposition and consecutively solves the respective sub-problems [Nie06]. For problems that are FPT with respect to treewidth, the general runtime of such algorithms for an instance of size n is $f(k) \cdot n^{\mathcal{O}(1)}$, where f is an arbitrary function over width k of the tree decomposition used. In fact, this approach has been used for several applications, including inference problems in probabilistic networks [LS88], frequency assignment [KvHK99], computational biology [XJB05], logic programming [MMP+12] as well as the Steiner Tree problem [FBN15].

From a theoretical point of view, the actual width k is the crucial parameter towards efficiency for FPT algorithms that use tree decompositions. In general, there exists a multitude of tree decompositions of the same width which can be obtained from a problem instance and it is often unclear, which one to choose for highest performance. In terms of FPT algorithms for treewidth, practical experience shows that repeated experiments with heuristically generated tree decompositions often suffer from poor robustness. Moreover, even selecting a different root node for exactly the same tree decomposition can dramatically change the running time of dynamic programming algorithms.

To overcome this issue of sometimes extremely diverging solving times for the very same problem instance, Morak et al. [MMP+12], for instance, suggested that the consideration of further features of tree decompositions is important for the actual runtime of dynamic programming algorithms for answer set programming. In another paper, Jégou and Terrioux [JT14] observed that the existence of multiple connected components in the same tree node (bag) may have a negative impact on the efficiency of solving constraint satisfaction problems.

A commonality of these observations is the fact that there is strong evidence that the width of the concrete tree decomposition at hand is not the only parameter which has influence on the actual runtime of a dynamic programming algorithm. This underlines that the quality of a tree decomposition cannot be judged without taking its shape and the concrete application scenario, i.e., the dynamic programming algorithm in which the given decomposition is used, into account.

Due to the inherent complexity of a variety of computational problems, the approach of decomposing problem instances and solving them in a divide-and-conquer manner (like it is done in dynamic programming algorithms based on tree decompositions) becomes important. Recently, several practical implementations of algorithms for computing tree decompositions participated in the "First Parameterized Algorithms and Computational

[1] Roughly speaking, the *width* of a tree decomposition is defined by the size of its largest component (*bag*). For more details regarding tree decompositions and related notions, see Section 2.2.

Experiments Challenge" (PACE 2016)[2]. According to its organizers, the primary goal of the PACE challenge is to unite fixed-parameter tractability and practice. Furthermore, also an open database for the computation, storage and retrieval of tree decompositions was initiated [vWK17] very recently.

1.1 Research Questions and Methodology

The fact that, given a problem instance and a dynamic programming algorithm, even tree decompositions of exactly the same width often lead to a significantly different runtime behavior, can be very problematic in practical application scenarios in which solving time is crucial. Therefore we see a big need to gain deeper insights into the actual structure of tree decompositions in order to increase robustness and efficiency of dynamic programming algorithms.

This strongly calls for detailed and extensive experiments which allow to precisely analyze the runtime behavior of dynamic programming algorithms in the presence of different tree decompositions. Furthermore, we do not want to restrict ourselves to a passive analysis of the effects of different tree decompositions (of the same width) on the efficiency of DP algorithms. We also want to be able to exploit our findings to stabilize and improve the runtime of those algorithms by providing efficient ways to customize tree decompositions effectively so that they fit optimally to the concrete dynamic programming algorithms in which they are used.

In some sense, the thesis at hand is therefore in line with the opinion of Gutin [Gut15], who recently stressed that, to turn the concept of fixed-parameter tractability to practical success, more empirical work is required.

In particular, in this thesis we want to answer the following research questions:

1. How does the shape of tree decompositions affect the performance of dynamic programming algorithms? That is, are there features of tree decompositions other than the plain width which allow to discriminate between different decompositions (of the same instance) in terms of their performance impact on DP algorithms?

   If there are, how can we exploit this valuable knowledge in order to improve the efficiency and robustness of dynamic programming algorithms which rely on tree decompositions?

2. What types of customization of tree decompositions exist and how can we make them easily accessible and practically usable for the growing community of developers of dynamic programming algorithms?

3. Do customized tree decompositions indeed give us a significant advantage in terms of a reduced overall running time of dynamic programming algorithms which use them instead of non-customized ones?

[2] See `https://pacechallenge.wordpress.com/track-a-treewidth/` for more details.

To answer Question 1, we characterize tree decompositions by means of several features and we use well-established techniques from the area of machine learning to predict the running time of dynamic programming algorithms based on these features. A large series of experiments is conducted in which the prediction is exploited in such a way that the tree decomposition of minimum predicted runtime is used by the dynamic programming algorithm instead of an arbitrary one of similar width. Ideally, the actual solving time for a given problem instance is then significantly lower for the "tailored" tree decomposition than for the random one.

Question 2 requires the development of a decomposition software library which is able to directly customize the computed tree decompositions as, to the best of our knowledge, there does not yet exist a framework for tree decompositions which allows for a problem-specific customization of the resulting decompositions. We will analyze the effectiveness and efficiency of our framework by means of a thorough experimental comparison between our system and other state-of-the-art tree decomposition frameworks.

To answer Question 3, we use our new software framework to investigate its impact on the running time of dynamic programming algorithms. This is done by an experimental comparison between the time needed to solve different problem instances using tree decompositions tailored towards the concrete dynamic programming algorithms and the time needed to solve the given problems based on "standard", non-customized tree decompositions.

1.2 Contribution

Our main contributions can be summarized as follows:

1. We identify a large set of tree decomposition features which allow to characterize the structure of a given tree decomposition. The proposed features include properties related to the decomposition size, features specific to node types as well as several structural parameters.

2. We show that, on the basis of well-established machine learning algorithms, one can achieve significant improvements in terms of the runtime of dynamic programming algorithms when choosing a tree decomposition from a pool of heuristically generated ones based on the identified features.

3. We provide a powerful, free and open-source software framework, called *htd*, which allows to efficiently compute and fully customize tree decompositions while still retaining the desired property of low width. The framework is developed with the goal of utmost flexibility and extensibility. For this reason, it is not only designed for easy usage, it also allows to integrate own implementations of algorithms so that developers can contribute to the framework and share their implementations while still benefiting from the huge variety of built-in utility functionality.

4. We underline the importance of tree decomposition customization by means of two case studies in which tree decompositions reflecting certain preferences of the developer of the dynamic programming algorithm lead to a significantly better and much more robust runtime behavior. The case studies are based on the problems Quantified Boolean Satisfiability (QSAT) and Steiner Tree.

1.3 Publications and Systems

The following publications serve as the basis for this thesis:

[ADMW15] Michael Abseher, Frederico Dusberger, Nysret Musliu, and Stefan Woltran. Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 275–282. AAAI Press, 2015.

[AMW17b] Michael Abseher, Nysret Musliu, and Stefan Woltran. Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. *Journal of Artificial Intelligence Research*, 2017. To appear.

[AMW17a] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd – A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. In *Proceedings of the 14th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR 2017)*, 2017. To appear.

An extended version of the publication [AMW17a] is provided in the report [AMW16b].

The following software artifacts were developed in the context of this thesis:

- **htd** – A free, open-source software framework for computing (customized) tree decompositions. It offers several efficient implementations of tree decomposition algorithms and it provides a rich set of built-in manipulation and customization operations which can be used to directly compute tree decomposition which adhere to a custom quality criterion.

  Apart from its built-in functionality for computing tree decompositions, *htd* also supports other kinds of graph and hypergraph decompositions, like, e.g., hypertree decompositions. Furthermore, *htd* offers well-documented interfaces so that the framework can be flexibly adapted to the actual needs. *htd* is available for download at `http://dbai.tuwien.ac.at/research/project/decodyn/htd/`.

The author is a coauthor of the following publications which are not part of this thesis:

[ABC+14b] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT System for Dynamic Programming on Tree Decompositions. In *Proceedings of the 14th European Conference On Logics In Artificial Intelligence (JELIA 2014)*, volume 8761 of *LNCS*, pages 558–572. Springer, 2014.

[ABC+15] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, and Stefan Woltran. Computing Secure Sets in Graphs using Answer Set Programming. *Journal of Logic and Computation*, 2015. Available at `https://doi.org/10.1093/logcom/exv060`.

[AMW16a] Michael Abseher, Marius Moldovan, and Stefan Woltran. Providing Built-In Counters in a Declarative Dynamic Programming Environment. In *Proceedings of the 39th German Conference on Artificial Intelligence (KI 2016)*, volume 9904 of *LNCS*, pages 3–16. Springer, 2016.

[AGM+16] Michael Abseher, Martin Gebser, Nysret Musliu, Torsten Schaub, and Stefan Woltran. Shift Design with Answer Set Programming. *Fundamenta Informaticae*, 147(1):1–25, 2016.

1.4 Thesis Outline

The remainder of this thesis is structured as follows: In Chapter 2 we give details regarding important concepts and notions which are used in the thesis at hand. Afterwards, in Chapter 3 we answer our first research question by investigating the impact of tree decompositions on the performance of dynamic programming algorithms. Then, in Chapter 4 we introduce the *htd* software framework for the computation of customized tree decompositions. In Chapter 5 we utilize the capabilities of the *htd* framework to boost the efficiency and robustness of dynamic programming algorithms. Thereafter, in Chapter 6 we present an overview of related work and Chapter 7 finally concludes our work.

CHAPTER 2

Background

This chapter provides the background information for the thesis at hand by introducing the basic concepts which will follow us throughout the remainder of this work. Apart from facts regarding the definition and application areas of tree decompositions, this chapter also comprises a short introduction to the topic of machine learning because techniques from this area are a vital ingredient for one of our two approaches for tree decomposition customization.

The chapter is structured as follows: At first, we give an overview of important topics in the area of computational complexity theory in Section 2.1. Then, in Section 2.2 we present the concept of structural graph decomposition (to which tree decompositions belong) and afterwards, in Section 2.3, we illustrate the approach of dynamic programming on tree decompositions. Finally, in Section 2.4 we introduce machine learning due to the fact that algorithms from this area are needed in Chapter 3.

2.1 Computational Complexity Theory

Complexity theory is one of the most prominent areas of research in theoretical computer science. The aim of this research discipline is to gain a deeper understanding of the complexity of given problems by classifying them into groups, so-called *complexity classes*, according to their "hardness". The term *problem* in the context of computational complexity theory refers to a specific task which is to be carried out following a strict, mathematical scheme, like, for instance, a (computer) algorithm. A manifestation of a problem by means of a concrete example is called an *instance* of the problem.

To give an exemplification of the two concepts *problem* and (problem) *instance*, let us consider the problem "Given a map or a globe, find the shortest connection between two cities A and B!". A concrete instance of the problem is then given by "Find the shortest connection between Berlin, Germany and Vienna, Austria!".

A long time, the definition of what it means that a problem is computationally harder to solve than another was rather vague and the rating was often based solely on intuition due to the lack of formal theory. This changed once Turing in the late 1930s laid the foundations of huge parts of complexity theory. In his seminal work [Tur36, Tur37] he developed a rigorous theory of computing machines. These abstract computing machines became known as *Turing machines*.

Turing machines are a vital concept in theoretical computer science as they represent a mathematical abstraction of "real" computers which allows for a thorough, formal investigation of the behavior of algorithms due to their mathematical nature. Basically, a Turing machine consists of a so-called *working tape* which is manipulated according to a table of rules (*instructions*) and a controllable read/write head which is moved over the working tape depending on the given rules. The working tape consists of an, in general, infinite number of cells where each cell can hold exactly one symbol of the machine's alphabet and the table of rules corresponds to the algorithm which is to be executed. The rules define transitions from the states of the Turing machine to the permitted successor states and so they define the algorithm the machine encodes. When a given Turing machine for each state has at most one successor state for each symbol of the machine's alphabet, we call the Turing machine *deterministic*. If for any state there is a symbol which leads to two or more distinct successor states, we call the Turing machine *non-deterministic*. If at least one computation path of a Turing machine leads to an accepting state in presence of an input instance, the machine *accepts* the given instance. Although there are several types of Turing machines, like deterministic and non-deterministic ones and also Turing machines with multiple tapes, it is a well-known fact they all have the same expressive power. However, the efficiency of computation can vary depending on the actual type of Turing machine at hand. A famous conjecture, the so-called *Church-Turing thesis*, states that every intuitively computable function is computable by a Turing machine.

The concept of Turing machines allowed for an alternative proof showing that there is no general solution to the Entscheidungsproblem (German for Decision Problem). The Entscheidungsproblem was introduced by Hilbert and Ackermann [HA28, HA38, HA50] and it asks, roughly speaking, whether there exists an algorithm which decides for any possible statement in first-order (predicate) logic whether this statement is valid.

While the proof showing the undecidability of the Entscheidungsproblem by Turing is based on Turing machines, Church [Chu36] presented a proof with the same result based on lambda calculus a few months before Turing. An important thing to note at this point is the fact that most of the modern programming languages like Java, C# or C++ are *Turing-complete*, i.e., their expressive power is identical to that of Turing machines. This means that the theoretical complexity of problems indeed carries over to practical implementations.

The observations made by Church and Turing immediately lead to the fact that there are problems which are undecidable in general. Hence, there is a strong separation between problems which are *effectively calculable* (Church [Chu36]) or *computable* (Turing [Tur36,

Tur37]) and those which are not. Once this border is drawn, there soon arise questions concerning the representatives of the large group of decidable problems how hard it actually is to solve them. The search for answers to these questions led to the foundation of the research area of (classical) complexity theory.

The complexity of a given problem is often measured in terms of the asymptotic worst-case runtime of an optimal[1] algorithm for the problem at hand with respect to the input size. If the dimension of interest in the process of complexity analysis is time, we refer to that dimension by the term *time complexity*. Another possible dimension of complexity analysis is memory consumption. In such a case we speak of *space complexity*. Throughout the remainder of this thesis, we will refer to time complexity unless stated otherwise. In classical complexity theory (see Section 2.1.1), the other dimension of the complexity analysis is the size of the input instance whereas in parameterized complexity theory (see Section 2.1.2) also additional parameters are considered.

It is important to note that computational complexity is also highly relevant in practice as the theoretical tools in many cases allow to judge whether there is potential for efficiency improvements beyond optimizations which are not relevant from a theoretical point of view. That is, complexity theory allows us, for instance, to investigate whether a problem can be solved in polynomial time or if (most likely) no such algorithm exists. If one can proof that no such algorithm exists under the assumption that $\mathsf{P} \neq \mathsf{NP}$ (see Section 2.1.1 for more details regarding these complexity classes), the problem is called *intractable*[2] and there is no need to waste time on trying to find a polynomial algorithm.

A detailed introduction to complexity theory can be found in [GJ79] as well as in [Pap94]. In the remainder of this chapter we will subsequently give an overview of important concepts in both classical and parameterized complexity theory in order to introduce the necessary preliminaries for this thesis.

2.1.1 Classical Complexity Theory

As mentioned earlier, classical complexity theory investigates computational problems of different kind with the goal to determine their complexity in terms of time or space requirements with respect to the input size.

The basic category of problems in the context of complexity theory is the category of *decision problems*. A decision problem simply asks whether there is a solution for a given problem instance. The formal definition of problems of such kind is given in Definition 1.

[1] In this context, an algorithm is considered optimal if the growth in terms of the runtime of the algorithm at hand in response to growing instance sizes is not higher than for any other algorithm for the same problem.

[2] The group of decidable problems can be roughly separated into *tractable* and *intractable* problems by investigating the growth in terms of the solving time any potential algorithm for a problem needs dependent on the size of the input instance. Informally speaking, a problem is called intractable if for each possible algorithm a small increase of the input size causes an exponential increase in terms of solving time (often referred to as "exponential explosion"). In case that there exists at least one algorithm for the problem whose runtime is polynomially bounded by the input size, the problem is called tractable.

Definition 1 (Decision Problem)**.** *Given a finite alphabet Σ, let Σ^* be the set of all finite strings over Σ. A* decision problem *is a language $\mathcal{L} \subseteq \Sigma^*$. Any $x \in \Sigma^*$ is an* instance *of the problem, that is, any $x \in \Sigma^*$ can act as input to the problem. If and only if $x \in \mathcal{L}$, x is called a* positive instance *of the problem, otherwise x is a* negative instance *of the problem.*

A *solution* to a problem in general is a special manifestation of a data object which acts as *witness* for the fact that the given problem instance is indeed a positive one. In the literature, the term *certificate* is often used as a synonym for a witness.

One possible way to obtain a decision variant of our example problem of finding the shortest route between two cities is to ask whether one can go by car, i.e., by using the road network, from one city to the other. A potential witness for a positive instance of this problem is then a sequence of roads that indeed connect the given cities.

Defining computational problems and, accordingly, the corresponding problem instances in terms of a formal language specification, like required by Definition 1, soon becomes rather complex and impractical. Therefore, problems are usually defined by means of the following two parts: A specification of the expected input instances followed by a problem statement asking a question about an input instance. This, in general, enhances readability significantly. The decision variant of our example problem could be, for instance, be defined simply as follows:

Input: A road network $\mathcal{G} = (V, E)$ and two cities $c_1, c_2 \in V$
Question: Is there a sequence of street sections $r_i \in E$ connecting c_1 and c_2?

The answer we obtain when solving a decision problem is either “yes” or “no”. Hence, when we are interested in the actual solution for the problem at hand in terms of a witness we need a different problem category. For this reason, although decision problems are the most prominent kind of problems in complexity theory, there are several additional categories of problems. Among them we find *search problems*, where one is interested in finding a certificate for the problem at hand, *counting problems*, where one wants to know the total number of solutions, as well as *enumeration problems*, where one wants to enumerate all possible solutions for a given problem instance.

Another very important type of problems is the one of *optimization problems*. For each of the aforementioned categories, except for decision problems (where such a refinement, in general, is not meaningful), we can define a variant where we ask for an optimal solution with respect to a given quality measure. Unless stated otherwise, in the remainder of this work we refer to the complexity of the decision variant of a given problem whenever we talk about complexity-theoretic facts.

A *complexity class* is, roughly speaking, a collection of problems with related complexity which contains all problems that are solvable by means of a given formal model of

computation (like Turing machines) using a bounded amount of resources. The most prominent and possibly most well-studied complexity classes are P and NP. Subsequently we provide the definitions for these important complexity classes.

Definition 2. *A problem is in* P *if it can be decided by a deterministic Turing machine in polynomial time with respect to the size of the input instance.*

Definition 3. *A problem is in* NP *if it can be decided by a non-deterministic Turing machine in polynomial time with respect to the size of the input instance.*

A problem is called *tractable* if the runtime of an algorithm solving the problem is bounded by a polynomial of the input size. Conversely, any problem where no algorithm with polynomial bound on the runtime with respect to the input size can be established is called *intractable*. Hence, all problems in P are tractable by definition. Based on the fact that a non-deterministic Turing machine can simultaneously reach multiple successor states from a given state when observing a symbol while a deterministic Turing machine can only reach a single successor state, it is obvious that $\mathsf{P} \subseteq \mathsf{NP}$. Currently, it is one of the biggest open questions in computer science whether the inclusion is proper, i.e., whether $\mathsf{P} \subset \mathsf{NP}$. It is widely believed that $\mathsf{P} \neq \mathsf{NP}$ which would on the one hand imply that the inclusion is indeed proper and on the other hand this also implies that there must exist problems in NP which are intractable in their nature. These problems then must have the property that any algorithm solving them requires (at least) exponential time with respect to the input size.

For any decision problem there is also its complement which inverts the question the decision problem asks, e.g., when the original decision problem asks whether a certain property holds for a given input instance, the complement problem would be the one of whether the property does not hold. Therefore, we can for each complexity class define its complement class. To indicate that a given complexity class is a complement of another, the prefix co- is used, e.g., the complement class of NP is named co-NP. The requirement for accepting an instance of a problem in the complexity class co-NP is that all computation paths of the corresponding non-deterministic Turing machine must lead to an accepting state in presence of the problem instance at hand. While it is obvious that $\mathsf{P} = \text{co-}\mathsf{P}$, the question whether the complexity classes NP and co-NP coincide is still open.

So-called *reductions* are the most prominent tool in computational complexity theory to show the actual relationship between two different problems in terms of their associated complexity classes.

Definition 4. *Given two problems A and B over the alphabets Σ_A and Σ_B, a function $R : \Sigma_A^* \to \Sigma_B^*$ is called a polynomial-time many-to-one reduction from the problem A to the problem B if and only if for any finite string $i \in \Sigma_A^*$ the following conditions hold: $i \in A \Leftrightarrow R(i) \in B$ and $R(i)$ is computable in polynomial time by a deterministic algorithm.*

The definition of many-to-one reductions allows us to show that, from a complexity-theoretical viewpoint, a given problem A is at least as hard to solve as a problem B by imposing a reduction from B to A. Note at this point that there are also other types of reductions than the one from Definition 4. For a detailed overview, see for instance [Pap94]. Apart from membership of a problem P in a complexity class C, subsequently denoted by $P \in \mathsf{C}$, two other very important notions in the context of complexity classes are the terms *hardness* and *completeness*.

Definition 5. *Given a problem P and a complexity class C, P is called C-hard if, for any problem $P' \in \mathsf{C}$, a reduction from P' to P exists. Whenever it is the case that P is C-hard and, additionally, $P \in \mathsf{C}$, P is called complete for the complexity class C and we say that P is C-complete.*

The prototypical example of a computational problem which is complete for the complexity class NP is the problem of Boolean Satisfiability (SAT). The definition of this problem is given as follows:

Input:	A propositional formula ϕ
Question:	Is ϕ satisfiable?

More precisely, the SAT problem asks, given a propositional formula ϕ, whether there exists an interpretation I over the boolean variables in ϕ under which the formula at hand evaluates to **True**. If this is the case, we call ϕ *satisfiable*. To the contrary, if there is no such interpretation, ϕ is called *unsatisfiable*.

Apart from P, NP and co-NP, there also exists a wide range of further complexity classes. For instance, the generalization of the aforementioned classes to so-called *oracle (Turing) machines*[3] allows to define the *polynomial hierarchy*. For further details and an overview of the variety of complexity classes, we refer to [GJ79, Pap94].

Although the polynomial hierarchy is assumed to be infinite (unless $\mathsf{P} = \mathsf{NP}$), it is rather coarse-grained due to the fact that the complexity is measured only with respect to the input size. To get deeper insights into the actual complexity of problems, one probably wants to consider also further parameters of a problem instance.

2.1.2 Parameterized Complexity Theory

In practical application scenarios it turns out that sometimes even very large instances of NP-hard problems can be solved relatively efficiently although the classical complexity theory tells us the contrary under the assumption that NP-hard problems are indeed

[3] By the term *oracle (Turing) machines* we refer to Turing machines which may call an oracle in order to solve a sub-problem of a given problem instance. An *oracle* is assumed to provide an answer to a given sub-problem in constant time, regardless of its actual complexity.

intractable, i.e., no polynomial-time algorithm exists for them. Therefore, in order to better understand the complexity of computational problems, it seems to be a worthwhile investment to search for further influence factors on the runtime (or space-consumption) of algorithms for solving the given problems instead of focusing merely on the input size.

For this reason, the research area of parameterized complexity theory [DF99, FG06, Nie06] was initiated. The goal of this relatively new field of research is to describe the actual complexity of computation problems not only by the input size but also by additional, problem-inherent *parameters*.

The incentive to do so is to capture those parameters which are relevant for actual real-world performance of algorithms. This motivation is based on the hopeful assumption that when a problem instance has small values for the parameters having crucial influence on the efficiency of the algorithm at hand, the problem instance is probably easy to solve although its size might be large. In this way, parameters can help to determine the properties of a problem instance which are responsible for the intractability.

Based on Definition 1 we define a parameterized decision problem as follows:

Definition 6 (Parameterized Decision Problem)**.** *Given a finite alphabet Σ, let Σ^* be the set of all finite strings over Σ. A* parameterized decision problem *is a language $\mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$. Any $(x, k) \in \mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$ is an* instance *of the problem with x being the* main part *of the instance and the* parameter *k.*

The main part of an instance of a parameterized decision problem correlates to the instance of the corresponding decision problem. The associated parameter is always a property of the main part of a given problem instance, like the size of a solution or the treewidth of a graph underlying the instance (see Section 2.2).

In general, there are various parameters of a problem instance which can be considered in order to potentially isolate the parts which are relevant for the complexity. Note that a problem can also be parameterized by a combination of multiple parameters derived from the main part, i.e., sometimes not a single parameter captures the intractable core of the problem, but a combination of them may do so.

A very important concept from the area of parameterized complexity theory which will follow us throughout the remainder of the thesis is the one of *fixed-parameter tractability*. This notion is defined as follows:

Definition 7 (Fixed-Parameter Tractability)**.** *Given a finite alphabet Σ, let Σ^* be the set of all finite strings over Σ. A parameterized decision problem $\mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$ is called* fixed-parameter tractable *if there exists a deterministic Turing machine that is able to decide for all instances $(x, k) \in \Sigma^* \times \mathbb{N}$ whether $(x, k) \in \mathcal{L}$ in a time frame bounded by the formula $f(k) * |x|^{\mathcal{O}(1)}$, where f is an arbitrary, computable function solely dependent on the instance parameter k.*

When showing the fixed-parameter tractability of a problem, it is crucial to state for which parameter (or combination thereof) the tractability of the problem at hand is given. This is due to the fact that a problem might be fixed-parameter tractable with respect to some parameter but this may not be valid for other parameters.

Another important thing to note is the fact that it is possible that from two different problems being complete for a complexity class, like NP, one of them is fixed-parameter tractable whereas the other is not. Therefore, parameterized complexity theory allows for a much more fine-grained investigation of the complexity of problems than it is possible in classical complexity theory.

In the remainder of this thesis we will focus on problems which are fixed-parameter tractable with respect to the parameter *treewidth*. In the following sections of this chapter we will first introduce this important parameter as well as the related concepts from the area of structural decomposition and afterwards we will illustrate how fixed-parameter tractability with respect to treewidth can be exploited in order to obtain efficient algorithms.

2.2 Structural Decompositions of Graphs

Graphs, i.e., collections of *vertices* (respectively, *nodes*) which are connected via edges, are an important concept to represent arbitrary items and their relationships in an abstract, computer-processable format. The following list gives a few real-world examples whose graph nature is more or less obvious:

- Road networks:

  In road networks, cities, villages or junctions can act as the vertices of the graph while the roads connecting the aforementioned concepts are forming the edges. Depending on the actual problem at hand, the distances between two vertices are often used as *edge weights*. Weights in this context help to discriminate between two alternatives. In this way, they allow to solve optimization tasks like finding the shortest connection between two endpoints.

- Public transport networks:

  Not only road networks, but also train, metro and bus networks are real-world examples for graphs. In these cases, the respective terminal stations act as the vertices of the graph and the underlying track system determines the edges. One common possibility for assigning edge weights is by measuring the travel time between the different stations.

- Social networks:

  The intention of social networks is to connect people all over the world to their fans, friends and colleagues. Based on this fact, the graph nature of social networks is

obvious when we consider the individuals as vertices and the relationships between them as the edges of a graph. One possibility to define edge weights is by considering the frequency of interactions between two individuals.

- Computer networks:

  Another omnipresent example for graphs in everyday life are computer networks in which computers and network components (like, for example, routers, switches or hubs) act as vertices and the wired or wireless connections between those endpoints act as edges. In the case of computer networks, the weight of an edge is often determined by the speed of the underlying physical connection.

While there are many other real-world examples for graphs, also many problems in the area of computer science whose graph nature is not obvious can be reduced to graphs. In order to provide the preliminaries for the remainder of this work, let us first define all concepts of graphs which are relevant in context of the thesis at hand.

Definition 8 (Graph)**.** *A* graph *(in the context of graph theory) is an abstract structure depicting arbitrary objects and their relations. Formally, a graph $\mathcal{G}$ is an ordered pair (V, E) where $E \subseteq V \times V$. The set V corresponds to the well-defined objects of which the graph consists. The elements $v \in V$ are called the* vertices *or, synonymously, the* nodes *of the graph. The set of vertex pairs denoted by E refers to the* edge set *of the graph $\mathcal{G}$, where an* edge *is an abstract relation between two vertices.*

If a graph $\mathcal{G} = (V, E)$ potentially contains duplicate edges we call $\mathcal{G}$ a *multi-graph*. Whenever the order of vertices within an edge of a graph matters, we call the given graph *directed* whereas in the opposite case, i.e., when the order does not matter, we call the graph *undirected*. In the remainder of this work we will always refer to undirected graphs unless stated otherwise.

Given a graph $\mathcal{G} = (V, E)$, a *walk* is a sequence $v_0, e_0, v_1, \ldots, v_{n-1}, e_{n-1}, v_n$ of vertices $v_i \in V$ and edges $e_i \in E$ where the edges e_i are exactly those edges which connect the vertices v_i and v_{i+1}, i.e., $e_i = (v_i, v_{i+1})$. The *length* of a walk is given by the number of contributing edges. A *path* is a walk where all vertices, except for its endpoints v_0 and v_n, are distinct and where no edge is used twice[4]. A *cycle* is a path of length greater than 1 whose endpoints v_0 and v_n refer to the same vertex. A graph is called *cyclic* if it contains at least one cycle, otherwise it is called *acyclic*. A *tree* is an undirected, acyclic graph.

Given a graph $\mathcal{G} = (V, E)$ and a vertex set $X \subseteq V$, the (unique) graph $\mathcal{G}' = (V', E')$ with $V' = X$ and $E' = \{(v_i, v_j) \mid (v_i, v_j) \in E, \{v_i, v_j\} \subseteq X\}$ is called the *induced subgraph* of $\mathcal{G}$ with respect to X and the edges E' are called the *induced edges* of $\mathcal{G}$ with respect to X.

[4]Note that in the context of an undirected graph $\mathcal{G} = (V, E)$ containing the vertices v_i and v_j, the edges (v_i, v_j) and (v_j, v_i) refer to the very same instance of an edge. Hence, a path must not contain both representations simultaneously.

A generalization of the concept of graphs is the notion of hypergraphs. Hypergraphs are different to graphs in the way the define the edge relation. While for the case of a graph, an edge always connects exactly two vertices, a hypergraph allows for so-called *hyperedges*.

Definition 9 (Hypergraph)**.** *A* hypergraph *(in the context of graph theory) is an abstract structure depicting arbitrary objects and their relations. Formally, a hypergraph $\mathcal{H}$ is an ordered pair (V, H) consisting of a set of vertices V and a set of hyperedges $H \subseteq \mathcal{P}(V)$, where $\mathcal{P}(V)$ denotes the power set over V.*

According to Definition 9, a hyperedge allows to connect an arbitrary number of vertices. If we allow duplicates of hyperedges in a given hypergraph, we refer to such a hypergraph by the term *multi-hypergraph*. The *induced subgraph* $\mathcal{H}' = (V', H')$ of a hypergraph $\mathcal{H} = (V, H)$ with respect to a vertex set X is defined analogously to the *induced subgraph* of a graph, i.e., it must hold that $V' = X$ and $H' = \{h \mid h \in H, h \subseteq X\}$.

After we introduced the concepts of graphs and hypergraphs, let us now come back to computational problems in order to draw the connection between them and graphs so that we can subsequently show how to solve them efficiently using structural decompositions of their instances. One of the most well-known representatives for a problem which can be represented by means of its underlying graph is the problem of Boolean Satisfiability, often abbreviated as SAT. To illustrate the relationship between SAT and graphs, recall the definition of the satisfiability problem of propositional formulae provided in Section 2.1:

Input:	A propositional formula ϕ
Question:	Is ϕ satisfiable?

To better understand the relationship between the SAT problem and graphs, let us consider the propositional formula provided in Example 1. We can see that the ϕ is given in conjunctive normal form (CNF), i.e., the formula is a conjunction of disjunctions (*clauses*). The separate parts of a clause, the so-called *literals*, are given by atomic formulae (*atoms*) or their negation. In this simple case, the formula consists of only three clauses and it uses only three distinct atoms, namely, a, b and c.

Example 1. $\phi = \underbrace{(\neg a)}_{\text{Clause } c_1} \wedge \underbrace{(\neg b \vee c)}_{\text{Clause } c_2} \wedge \underbrace{(a \vee b \vee \neg c)}_{\text{Clause } c_3}$

*The formula ϕ has exactly two solutions: $\{a = \textbf{False}, b = \textbf{False}, c = \textbf{False}\}$ as well as $\{a = \textbf{False}, b = \textbf{True}, c = \textbf{True}\}$. This is caused by the fact that Clause c_1 forces us to set variable a to **False** and hence, Clause c_3 can only be satisfied if either b is set to **True** or c is set to **False**. Regardless of the alternative we choose, Clause c_2 then forces us to assign the same truth value to the atoms b and c.* □

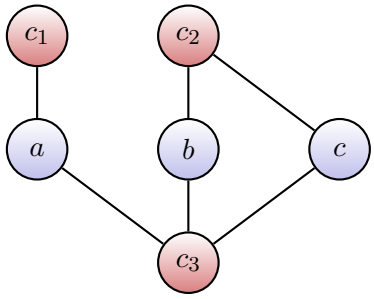


Figure 2.1: Graph Representation of the Propositional Formula ϕ from Example 1

One possibility to obtain a graph representation of the formula ϕ is to consider both the clauses and the atoms as vertices and to connect the atom vertices to the clause vertices according to their relationship in ϕ. That is, we connect the atoms to the clauses in which they occur via simple edges[5]. The outcome of this procedure, illustrated in Figure 2.1, is called the *variable-clause incidence graph* of ϕ. For a better distinction between the vertices which correspond to clauses and those which correspond to the atoms of the propositional formula the figure uses different colors for those groups of vertices.

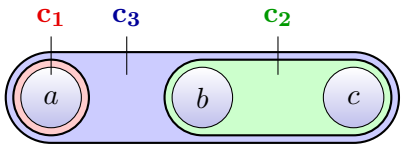


Figure 2.2: Hypergraph Representation of the Propositional Formula ϕ from Example 1

When we consider hypergraphs, i.e., graphs where the edge relation is not limited to binary connections between vertices, we can also represent the formula ϕ like illustrated in Figure 2.2. As already mentioned earlier in this section, hypergraphs generalize the concept of graphs by allowing *hyperedges* which in contrast to simple edges allow us to capture relations between multiple vertices. Exploiting this useful feature we can simply represent each clause of the propositional formula ϕ by its corresponding hyperedge as shown in Figure 2.2.

It is a well-known fact that every hypergraph $\mathcal{H} = (V, H)$ can be transformed to a graph $\mathcal{G} = (V, E)$ without hyperedges by considering the primal graph of $\mathcal{H}$. The primal graph (also known as the *2-section*, the *clique graph* or the *Gaifman graph*) of a given hypergraph $\mathcal{H}$ consists of the same vertices as $\mathcal{H}$, but instead of hyperedges it contains simple edges between any pair of vertices occurring together in a hyperedge of $\mathcal{H}$.

Note that the graph and hypergraph representations illustrated by Figures 2.1 and 2.2 do not distinguish whether an atom occurs positively or negatively in a given clause. This knowledge is indeed crucial for solving the problem at hand. Hence, the information

[5]By the term *simple edge* we refer here to binary edges, i.e., edges with exactly two distinct endpoints.

whether an atom occurs in the context of a positive or negative literal within a clause must either be maintained independently from the graph structure or, more conveniently, one could also use dedicated vertex or (hyper-)edge labels for this purpose.

Once the scene is set, we can now focus on the topic of structural decompositions of graphs. Basically, a decomposition of a graph $\mathcal{G} = (V, E)$ is an arbitrary segmentation of the input graph $\mathcal{G}$ into distinct, potentially overlapping parts, the *subgraphs* of $\mathcal{G}$. While many different types of structural decompositions of graphs exist, like tree decomposition [BB73, Hal76, RS84], branch decompositions [RS91], and hypertree decompositions [GLS02] (of hypergraphs), the concept of tree decompositions is maybe the most prominent one among them.

This is probably caused by the fact that many NP-hard search problems become tractable when they are parameterized by the parameter *treewidth*, a notion defined by means of tree decomposition (see Section 2.3 for details about the concept underlying the corresponding algorithms). In the following we give a formal definition of the terms *tree decomposition* and *treewidth*. An easily accessible introduction to the topic of tree decompositions and their applications is given in [Bod93].

Tree decomposition is a technique often applied for solving NP-hard computational problems. The underlying intuition is to obtain a tree from a (potentially cyclic) graph by subsuming multiple vertices in one node and thereby isolating the parts responsible for the cyclicity. Formally, the notions of tree decomposition and treewidth are defined as follows [RS84, BK10]:

Definition 10 (Tree Decomposition)**.** *Given a graph $\mathcal{G} = (V, E_\mathcal{G})$, a* tree decomposition *of $\mathcal{G}$ is a pair $(\mathcal{T}, \chi)$ where $\mathcal{T} = (N, E_\mathcal{T})$ is a tree and function $\chi : N \to 2^V$ assigns to each node a set of vertices (called the node's* bag*), such that the following conditions hold:*

1. *For each vertex $v \in V$, there exists a node $n \in N$ such that $v \in \chi(n)$.*
2. *For each edge $(v_i, v_j) \in E_\mathcal{G}$, there exists a node $n \in N$ with $\{v_i, v_j\} \subseteq \chi(n)$.*
3. *For each $m, n, o \in N$: If n lies on the path between m and o then $\chi(m) \cap \chi(o) \subseteq \chi(n)$.*

The width *of a given tree decomposition is defined as $\max_{i \in N}(|\chi(i)|) - 1$. In general, different tree decompositions for the same graph exist. The* treewidth *of a graph is the minimum width over all its tree decompositions.*

Indeed, a tree decomposition can also be obtained from a given hypergraph. To define tree decompositions for hypergraphs $\mathcal{H} = (V, H)$, we only have to adapt Definition 10 accordingly by replacing Criterion 2 with the following requirement: For each edge $h \in H$, there exists a node $n \in N$ with $h \subseteq \chi(n)$. In the following we consider rooted tree decompositions, i.e., tree decompositions in which one node acts as dedicated root of the tree. Choosing an explicit root node is of importance in many cases because the common graph-theoretic notions *child* and *parent* are defined only on rooted trees.

Figure 2.3 shows the graph from our introductory example and one of its (non-normalized) tree decompositions having root node n_1. Again, we use different colors for the vertices of the graph to allow for a better visual distinction between the vertices which refer to atoms and those which refer to clauses of the SAT instance. For the tree decomposition on the right-hand side of the figure, this distinction is given by emphasizing the atoms by means of underlines.

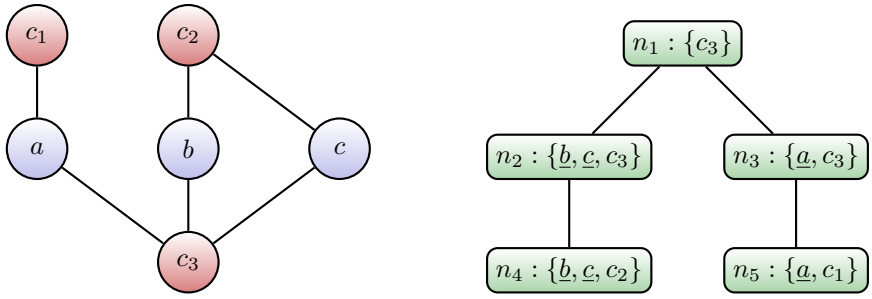


Figure 2.3: Example Graph and a Possible Tree Decomposition.

In order to avoid confusion, from now on we will always use the term "node" to refer to nodes of the tree decomposition and the term "vertex" will be used to refer to vertices of the input graph.

Definition 11 (Normalized Tree Decomposition)**.** *Given a graph $\mathcal{G} = (V, E)$, a* normalized *(sometimes also called* nice*) tree decomposition of $\mathcal{G}$ is a rooted tree decomposition $\mathcal{T} = (N, E_{\mathcal{T}})$ where each node $n \in N$ is of one of the following types:*

1. Leaf*: n has no child nodes.*
2. Introduce Node*: n has one child o with $\chi(o) \subset \chi(n)$ and $|\chi(n)| = |\chi(o)| + 1$*
3. Forget Node*: n has one child o with $\chi(o) \supset \chi(n)$ and $|\chi(n)| = |\chi(o)| - 1$*
4. Join Node*: n has two children o, p with $\chi(n) = \chi(o) = \chi(p)$*

A special type of tree decomposition are *normalized* ones (see Definition 11). Each tree decomposition can be transformed into a normalized one in linear time without increasing the width [Klo94] by introducing additional nodes. Analogously, one can perform various other normalizations without affecting the width of the input tree decomposition. Two representatives of such normalizations are what we call *semi-normalized* and *weakly-normalized* tree decompositions. The former is a relaxed variant of the normalized tree decomposition where no restrictions are imposed on nodes with exactly one child. The latter additionally allows join nodes to have an arbitrary number of children whose bags match the join nodes' bag.

It is a well-known fact that constructing a tree decomposition of minimal width for a given graph is intractable [ACP87]. In spite of this intractability result there exists

a variety of exact methods which allow to obtain minimal-width tree decompositions at least for small graphs, like, e.g., [SG97, GD04, BB06]. Researchers also proposed several efficient heuristic approaches that usually construct tree decompositions of almost optimal width for larger graphs. Maybe the most prominent representatives of greedy heuristic algorithms for computing tree decompositions are Maximum Cardinality Search (MCS) [TY84], Min-Fill heuristic [Dec03], and Minimum Degree heuristic [BHS03], to mention just a few. Metaheuristic techniques have been provided in terms of genetic algorithms [LKPM97, MS07], ant colony optimization [HM10], and techniques based on local search [Kjæ92, CMNC04, Mus08]. An overview as well as a detailed description of tree decomposition techniques is given in the recent surveys [BK10, HMS15].

For a given graph $\mathcal{G}$, the treewidth can be found from its *triangulation*. Subsequently, we will give basic definitions, explain how the triangulation of graph can be constructed and show the relation between the treewidth of a graph and treewidth of a corresponding, triangulated graph.

Two vertices u and v of a graph $\mathcal{G} = (V, E)$ are *neighbors* if they are connected by an edge $e \in E$. The *neighborhood* $N(v)$ of a vertex v is defined as $\{w | w \in V, (v, w) \in E\}$. A set of vertices is called a *clique* if there is an edge between each pair of vertices. An additional edge connecting two previously non-adjacent vertices of a cycle is called *chord*. A graph is called *triangulated* or *chordal* if there exists a chord in every cycle of length larger than 3. The term *triangulation* is based on the fact that all cycles induced by a triangulated graph contain exactly three vertices, i.e., all induced cycles of a chordal graph are triangles. Note that the term "induced cycle" here refers to a sequence of vertices in which for each pair of adjacent vertices in the sequence it holds that there is an edge connecting the two vertices in the given graph. Further, for each pair of non-adjacent vertices in the sequence there must not exist such an edge in the graph.

A vertex is called *simplicial* if its neighbors form a clique. Given a graph $\mathcal{G} = (V, E)$, an ordering $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ of the vertices of V is called a *perfect elimination ordering* for $\mathcal{G}$ if for any $i \in \{1, 2, \ldots, n\}$, σ_i is a simplicial vertex in the subgraph of $\mathcal{G}$ induced by the vertices σ_i to σ_n, denoted by $\mathcal{G}[\sigma(i), \ldots, \sigma(n)]$ [CMNC04]. In [FG65] it is shown that a graph $\mathcal{G}$ is triangulated if and only if it has a perfect elimination ordering. Given an elimination ordering of vertices, the triangulation $\mathcal{G}^t$ of graph $\mathcal{G}$ can be constructed as follows: Initially, $\mathcal{G}^t = \mathcal{G}$. Then, the next vertex which is to be eliminated according to the given elimination order is made simplicial by adding new edges connecting all its neighbors in current $\mathcal{G}$ and $\mathcal{G}^t$. Afterwards, the respective vertex is eliminated from $\mathcal{G}$. These steps are repeated for all vertices in the given elimination ordering. A more detailed description of the algorithm for constructing a graph's triangulation for a given elimination ordering is found in [KBvH01].

The treewidth $tw(\mathcal{G})$ of a triangulated graph $\mathcal{G}$ can be calculated based on its cliques. For a triangulated graph, the treewidth is known to be equal to the size of its largest clique minus 1 [Gav72]. Determining the largest clique of a triangulated graph can be done in polynomial time. The time complexity of calculating the largest clique for a triangulated graphs $\mathcal{G} = (V, E)$ is $\mathcal{O}(|V| + |E|)$ [Gav72]. For every graph $\mathcal{G} = (V, E)$ there exists a

corresponding triangulation, $\mathcal{G}^t = (V, E \cup E^t)$, with $tw(\mathcal{G}^t) = tw(\mathcal{G})$. Thus, finding the treewidth of a graph $\mathcal{G}$ is equivalent to finding a triangulation $\mathcal{G}^t$ of $\mathcal{G}$ with minimum clique size (for more information see [KBvH01]).

A famous result by Courcelle [Cou90] underlines the usefulness of tree decompositions for the solving process of a wide range of computationally hard problems which can be reduced to a graph. The result states that any graph property expressible in *monadic second-order logic*[6] (MSO) is decidable in linear time in the case that the graph's treewidth is bounded. A similar result, developed independently of the findings made by Courcelle, is provided by Borie et al. [BPT92].

Formulating a problem by means of MSO therefore establishes fixed-parameter tractability of the problem with respect to the parameter treewidth. Nevertheless, although an MSO formulation of a given problem allows us to find an algorithm for the problem at hand according to Courcelle's theorem, the algorithm obtained in this way is often impractical due to huge constant factors [Nie06]. To overcome this issue, we will introduce a more promising approach for practical application scenarios in the following section.

2.3 Dynamic Programming on Tree Decompositions

Data of almost any kind is collected in various forms in a multitude of application scenarios in a higher and higher frequency nowadays and processing this growing haystack efficiently in order to find the proverbial needle becomes a more and more challenging task. This is caused on the one hand by the sheer amount of data itself which has to be handled, but on the other hand, many reasoning problems occurring in practice are computationally intractable in their nature.

A common approach to handle complex problems efficiently by means of computer algorithms is the technique of divide-and-conquer. This approach works by recursively breaking the task at hand into separate parts until they become trivially (or at least easily) solvable. Depending on the actual application scenario, the solutions of the sub-problems sometimes can be computed in parallel which often allows for an additional performance gain. After solving the individual parts of the problem instance at hand, the partial solutions are assembled according to a given (problem-specific) strategy in order to obtain the solution(s) for the complete problem instance.

Algorithms based on the idea of divide-and-conquer come in different flavors. Tractable, polynomial time solvable cases include for instance prominent sorting algorithms like Quicksort or Mergesort. The former is an example for divide-and-conquer where the "division step" is complex and the re-combination of the partial solutions is easy while the latter one is an example where partitioning the input is easy but the re-combination of the partial solutions is, in contrast, relatively complex.

[6] Monadic second-order logic extends first-order (predicate) logic by allowing quantifiers over unary relational variables. In the case of graphs, the relational variables refer to the vertices and edges of a graph.

Another intuitive example for divide-and-conquer is the search for an element in a binary search tree. When searching for a value x in such a tree, we divide the search space by recursively deciding in each node whether we proceed by following the child at the left-hand side or the child at the right-hand side, depending on the value associated with the current node. In the last step, i.e., the step where either the value assigned to the current tree node matches our search criterion or where there is no appropriate child node to continue the search, we can immediately return the search result. Hence, in this case there is no re-combination phase needed.

Closely related to the approach of divide-and-conquer is the technique of *dynamic programming* (DP). As mentioned before, divide-and-conquer works by splitting the problem instance at hand into different, independent sub-problems. Each of these sub-problems is solved recursively and finally, the partial solutions are combined in order to obtain the complete solution(s) to a given problem instance. One commonality between divide-and-conquer and dynamic programming is the process of splitting the overall problem into sub-problems, but in contrast to divide-and-conquer where the sub-problems are independent, dynamic programming is a technique in which the sub-problems often overlap. Therefore, strategies allowing to get an efficient handle on the recurring parts of the sub-problems with the goal to solve each of those parts only once, play a central role in the context of dynamic programming algorithms. The general approach to avoid redundant computations is called *memoization* and it refers to the storage of solutions to sub-problems so that they can be looked up efficiently when it comes to solving subsequent pieces of the problem instance at hand.

One of the most prominent realizations of the dynamic programming approach is the technique of dynamic programming on tree decompositions. For graph problems and problems that can be formulated on a graph, tree decompositions permit a natural way of applying dynamic programming by traversing the tree from the leaf nodes to its root. For each node n of a given tree decomposition, the solutions for the subgraph of the instance graph induced by the vertices in $\chi(n)$ are computed. When processing a node, the (partial) solutions computed for its children are taken into account, such that only consistent solutions are computed. Thus, the partial solutions computed in the root node are consistent with the solutions for the whole instance graph.

As mentioned in the overview of parameterized complexity theory in Section 2.1.2, the key aspect of FPT algorithms is to bound the costs to compute these solution with respect to some parameter. In this case, the parameter of interest is the width of the given tree decomposition. The complete solutions for a problem instance can be obtained (with polynomial delay) in a reverse traversal from the root node to the leaves of the tree decomposition by combining the computed partial solutions.

Subsequently, we illustrate how dynamic programming on tree decompositions can be employed to solve the instance of the SAT problem acting as our introductory example. In order to do so, recall its corresponding propositional formula $\phi = (\neg a) \wedge (\neg b \vee c) \wedge (a \vee b \vee \neg c)$ with clauses $c_1 = (\neg a)$, $c_2 = (\neg b \vee c)$ and $c_3 = (a \vee b \vee \neg c)$ and consider the associated graph representation of the formula as well as the tree decomposition from Figure 2.3.

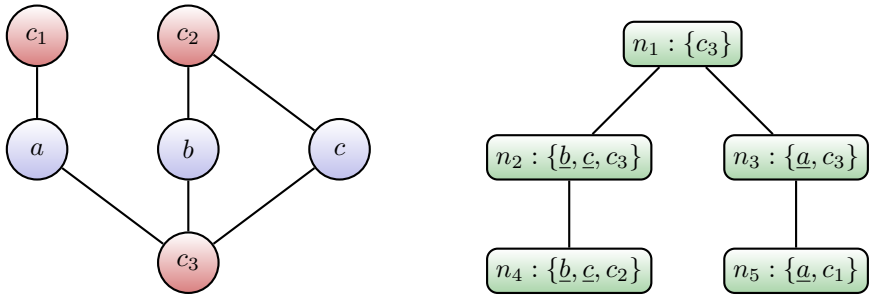


Figure 2.3: Graph Representation and a Tree Decomposition of ϕ (See also Page 19)

Figure 2.4 shows the dynamic programming tables for each of the tree decomposition nodes. The first column gives the ID of the partial solution candidates and the central columns of each table list the possible values for the vertices in the node constituting the partial solutions. Here, the value **T** (**F**) for a atom vertex stands for the respective atom being set to **True** (**False**). Analogously, the value **S** stands for the fact that the respective clause is satisfied. Unless a clause is known to be already satisfied – which is the case if at least one of its literals evaluates to **True** – its truth value is undetermined, denoted by the letter **U**.

The last column of the DP tables stores the possible partial solutions associated with the child node(s) that can be consistently extended to the given truth values. We call these links to child solutions *extension pointers*. Note that infeasibilities in the partial solutions can already lead to an early removal of solution candidates. The partial solution for Node 5 where $\{a = \mathbf{T}\}$ can, for instance, already be discarded when Clause c_1 is forgotten during the traversal to the next node n_3. This is because, due to Criterion 3 (connectedness) of Definition 10 (tree decompositions), c_1 cannot appear again in any node of the decomposition visited in later steps of the post-order traversal and, thus, Clause c_1 cannot become satisfied anymore.

The solution for the example graph can then be simply extracted by starting at the root of the tree decomposition and following the extension pointers down to the leaves. Indeed, in order to obtain only the valid solutions of the problem instance at hand, we may only start from the partial solution $n_1|0$ in our example because the solution $n_1|1$ does not satisfy Clause c_3. A common approach to make such side notes superfluous is to use a root node with empty bag in the tree decomposition and this modification is often also beneficial for the design of dynamic programming algorithms: To add a new root node to a given tree decomposition as parent of the old one is of low, constant effort, but having a root node with empty bag in many cases avoids the need for a special treatment of the tree decomposition's root node in the context of the corresponding dynamic programming table.

As expected with regard to the exemplification in Section 2.2, one possible solution for the SAT instance ϕ is given by the interpretation $\{a = \mathbf{F}, b = \mathbf{F}, c = \mathbf{F}\}$ over the atoms a, b and c. This solution is represented in the dynamic programming tables by

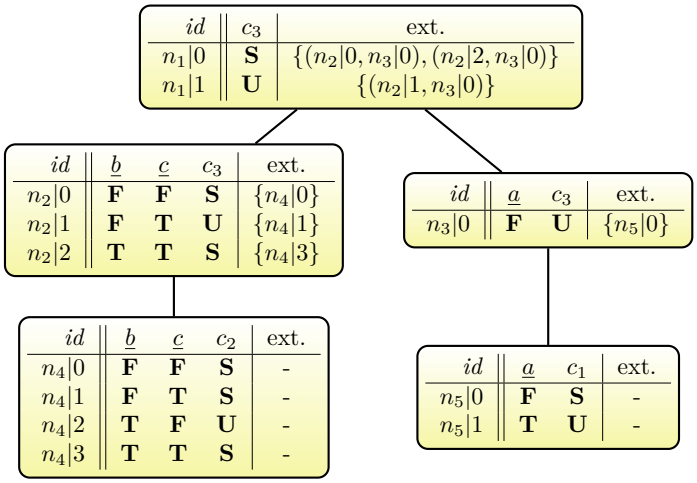


Figure 2.4: Solving SAT via DP on Tree Decompositions for the Problem Instance ϕ

the partial solutions $n_1|0, n_2|0, n_3|0, n_4|0$ as well as $n_5|0$. Analogously, we derive the second interpretation over the atoms which satisfies ϕ, namely $\{a = \mathbf{F}, b = \mathbf{T}, c = \mathbf{T}\}$, by considering the partial solutions $n_1|0, n_2|2, n_3|0, n_4|3$ as well as $n_5|0$.

Note that in the dynamic programming table associated with the join node named n_1 we derive that Clause c_3 is satisfied if we combine the partial solutions $n_2|0$ or $n_2|2$ with the partial solution for Node n_3. This is a valid conclusion due to the fact that the literals in a clause are connected by disjunctions and so there is no way that a previously satisfied clause can become unsatisfied. Hence, in our scenario of a propositional formula in CNF we can safely derive that a clause c_x is satisfied in the context of a join node with children n_i and n_j if there are two partial solutions $n_i|s_l$ and $n_j|s_m$ that agree on the truth values of all atoms and at least one of them sets c_x to satisfied.

While our example is quite small, so that without any problem one could carry out the computation of the solutions by means of simpler techniques than dynamic programming on tree decompositions, real-world instances of the SAT problem can contain millions of clauses over several hundreds of thousands of variables. Due to the fact that for n boolean variables there are 2^n possible interpretations – each variable can be set either to **True** or to **False** – it can indeed pay off to consider structural decomposition in those cases.[7] Because many large instances of the SAT problem have high treewidth, it is hard to beat dedicated, well-established SAT solvers by means of dynamic programming on tree decompositions. For this reason, we want to clarify that the exemplification of the dynamic programming approach by means of the SAT problem is given just for illustration purposes as it shows the connection between the probably most prominent problem from complexity theory and dynamic programming on tree decompositions.

[7]Solving SAT via DP on tree decompositions has an asymptotic worst-case running time of $2^w * n$ where n is the size of the input formula in terms of variables and clauses and w is the maximum bag size of the tree decomposition which is used.

The *D-FLAT* System

In many of the experiments contributing to the thesis at hand we will use the dynamic programming framework *D-FLAT*[8] [BMW12, Bli12, ABC+14a, ABC+14b]. *D-FLAT* is a state-of-the-art system that applies dynamic programming on tree decompositions and it is a general framework capable of solving any problem expressible in monadic second-order logic (MSO) in FPT time with respect to the parameter *treewidth* [BPW13].

The goal of *D-FLAT* is to make the development of dynamic programming algorithms for complex computational problems as easy as possible. For this reason, the *D-FLAT* framework allows to specify both the problem as well as the input instances by means of *answer set programming* (ASP). Two easily accessible introductions to the topic of ASP are given by Lifschitz [Lif08] and Brewka et al. [BET11]. Subsequently, we provide a short summary of the most important concepts in the domain of ASP.

Answer set programming allows to formulate computational problems up to the second level of the polynomial hierarchy in a declarative way [MT99, Nie99]. The declarative nature of ASP allows that, instead of specifying an algorithm for solving a given problem, one defines certain rules and constraints which characterize the solutions corresponding to a problem instance. A complete specification of a problem by means of ASP is called *logic program* or *encoding*. Once such an encoding is constructed, the rest, including non-determinism, is completely up to the ASP engine. This makes answer set programming a very powerful, yet easy-to-use approach to model NP-hard search problems.

An ASP encoding supposes a language with function and predicate symbols having a corresponding arity (possibly 0) as well as variables. Function symbols with an arity of 0 are also called *constants*. By convention, variables begin with upper-case letters while function and predicate symbols start with lower-case letters. Each variable and each constant is a *term*. If f is a function symbol with arity n, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is also a term. If p is an m-ary predicate symbol and $t_1, \ldots, t_m$ are terms, then we call $p(t_1, \ldots, t_m)$ an *atom*. A literal is either just an atom or an atom with the word "not" put in front of it. Roughly speaking, the word "not" here expresses the absence of the respective atom, i.e., the literal matches when the referenced atom is not contained in the answer set candidate. Based on these building blocks, an ASP encoding or logic program is a set of rules of the form

$$a \leftarrow b_1,\ \ldots,\ b_m,\ \text{not } b_{m+1},\ \ldots,\ \text{not } b_n$$

where a and $b_1, \ldots, b_n$ are atoms. Let r be a rule of a program Π. We call $h(r) = a$ the *head* of r, and $b(r) = \{b_1, \ldots, b_n\}$ its *body* which is further divided into a *positive body*, $b^+(r) = \{b_1, \ldots, b_m\}$, and a *negative body*, $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. If the body of a rule r is empty, r is called a *fact*, and the symbol $\leftarrow$ can be omitted. Sometimes, the head atom of a rule is omitted. By doing so, we obtain a special type of rule which is called *integrity constraints*. An (integrity) constraint looks as follows:

$$\leftarrow b_1,\ \ldots,\ b_m,\ \text{not } b_{m+1},\ \ldots,\ \text{not } b_n$$

[8] *D-FLAT* is available at `http://dbai.tuwien.ac.at/research/project/dflat/system/`.

Whenever a term does not refer to any variables it is called *ground*. Analogously, we also call rules and complete logic programs ground if they do not contain any variables. A given logic program can be transformed to its ground counterpart by systematically replacing all variable occurrences by appropriate ground terms.

The basic idea behind ground rules in the context of logic programs is that whenever the complete positive body is contained in an answer set candidate and, at the same time, no part of the negative body is contained in the given interpretation, then also the head atom must occur in the answer set candidate. In contrast to ground rules which allow to derive new atoms, integrity constraints are used to filter potential answer set candidates: Whenever a set contains all atoms from the positive body but none of the negative body of an integrity constraint, the respective set cannot be an answer set.

```
1 1 { map(A,true); map(A,false) } 1 ← atom(A).
2 sat(C) ← pos(C,A), map(A,true).
3 sat(C) ← neg(C,A), map(A,false).
4 ← clause(C), not sat(C).
```

Listing 2.1: ASP Encoding for Boolean Satisfiability (SAT)

In order to illustrate the declarativeness of the paradigm of answer-set programming we provide in Listing 2.1 an exemplification of a logic program that can be used to solve instances of the SAT problem. The encoding assumes that we are given a problem instance in terms of the predicates `atom` (with arity 1), `clause` (with arity 1) as well as `pos` and `neg` (both with arity 2). The argument of the predicate `atom` (`clause`) denotes an identifier of an atom (clause) of the SAT instance, the first argument used for the predicates `pos` and `neg` refers to the identifier of a clause and the second argument is again an atom identifier.

The first line of Listing 2.1 depicts a so-called *choice rule*. Choice rules extend the concept of basic rules by allowing to non-deterministically select a number of elements from a pool in a convenient and intuitive way. The number of selected elements can be restricted using a upper and lower bound. In the case at hand, both limits are set to 1 as we want to select exactly one truth value for each atom. Note that choice rules can be transformed to basic rules using approaches like presented in [BJ13]. Lines 2 and 3 then encode the idea that, whenever an atom occurs positively (negatively) in a clause and the atom is set to **True** (**False**), the corresponding clause is satisfied. All that remains in order to obtain the solutions for a SAT instance is to eliminate all solution candidates which do not satisfy some clause. This is achieved via Line 4 of the provided encoding. The solutions of the original problem instance can then be reconstructed by inspecting the atoms `map` within the resulting answer sets.

Following the short introduction to answer set programming, let us now have a look at the *D-FLAT* system itself. The general workflow of the *D-FLAT* framework for dynamic programming on tree decompositions is depicted in Figure 2.5. In this illustration we can

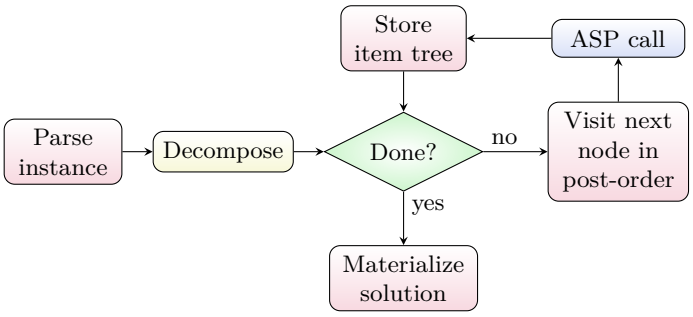


Figure 2.5: The Control Flow in *D-FLAT* (Adapted from [ABC+14a], Page 15)

see that *D-FLAT* first parses the given instance. This is done with regard to the ASP predicates that are specified to be considered for the edge relation before the invocation of *D-FLAT*. In the case at hand, the predicates defining the edge relation are `pos` and `neg`. Once the input graph is constructed, *D-FLAT* invokes a tree decomposition algorithm in order to efficiently decompose the given problem instance. After these preparatory steps, *D-FLAT* starts to traverse the obtained tree decomposition in a bottom-up manner. In each step of the traversal, the partial solutions associated with the current tree decomposition node are computed by issuing a call to the ASP engine with the provided ASP encoding, the current node's bag content and the children's partial solutions as input. Additionally, the ASP routine may access the complete input specification so that it is also possible for the dynamic programming algorithm to work with potential labels, e.g., denoting weights of vertices or edges. Finally, when the post-order traversal is finished, *D-FLAT* takes care of materializing the solutions of the given problem instance.

The following enumeration lists the main features of the *D-FLAT* system:

- *D-FLAT* allows to specify computational problems declaratively, thus relieving developers of dynamic programming algorithms from the effort to implement algorithms in an imperative way which can be error-prone and the resulting system can be hard to maintain.

- Also the input instances for the *D-FLAT* system can be specified fully declaratively in the common syntax of today's state-of-the-art ASP systems. Using this well-defined input format together with the ability to use custom edge predicates brings the advantage that, also in those cases where the original problem is not defined on a graph, *D-FLAT* can effectively be employed to solve the instances at hand.

- The task of computing a tree decomposition of the input instance is completely performed inside the system boundaries of *D-FLAT*. This means that there is no need for a developer to provide an algorithm which computes the decompositions underlying the dynamic programming steps.

The latest versions of *D-FLAT* use *htd* (see Chapter 4), the software framework for computing customized tree decompositions which was developed in the context of this thesis. This allows to tune the computed decompositions according to the actual needs.

- Maintaining the information between two steps of the dynamic programming algorithm as well as the combination of the partial solutions is also done directly "inside" *D-FLAT*. Moreover, the system also supports solving optimization problems in a convenient way based on the notion of *solution costs*. When optimization is involved, *D-FLAT* takes care that only optimal partial solutions, i.e., those with minimal cost, are retained whereas suboptimal candidates are automatically discarded.

In fact, a developer using the *D-FLAT* framework for dynamic programming on tree decompositions only has to provide a proper specification of the input instance and an ASP program which encodes the semantics of a dynamic programming algorithm, like the one shown in Listing 2.2.

After introducing the basic concepts of ASP and the *D-FLAT* framework, let us now have a look at the specifics of *D-FLAT*. Listing 2.2 shows one possible *D-FLAT* encoding for the SAT problem. It makes intensive use of various input and output predicates defined by *D-FLAT* (see [ABC+14a] for details). Those special predicates allow a developer to "communicate" with the framework in order to populate the dynamic programming tables (recall Figure 2.4), thus maintaining the partial solutions of the problem.

```
1  %dflat: --tables -e pos -e neg --no-empty-leaves -n semi
2  false(R,X) ← childRow(R,N), bag(N,X), not childItem(R,X).
3  unsat(R,C) ← childRow(R,N), bag(N,C), not childItem(R,C).
4  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
5  ← extend(R), extend(S), atom(A), childItem(R,A), false(S,A).
6  ← clause(C), removed(C), extend(R), unsat(R,C).
7  item(X) ← extend(R), childItem(R,X), current(X).
8  item(C) ← extend(R), childItem(R,C), current(C).
9  { item(A) : atom(A), introduced(A) }.
10 item(C) ← current(C), current(A), pos(C,A), item(A).
11 item(C) ← current(C), current(A), neg(C,A), not item(A).
```

Listing 2.2: *D-FLAT* Encoding for Boolean Satisfiability (SAT)

In Line 1 of Listing 2.2 we inform *D-FLAT* about the actual configuration which shall be used. In this case, because we deal with a problem on the first level of the polynomial hierarchy, we use a simple table as data structure for storing the partial solutions. For NP-complete problems which are FPT with respect to treewidth, we can always use the so-called *table mode* of *D-FLAT*. On the one hand, this mode reduces the complexity of

the task of writing the encoding due to the fact that a simplified set of input and output predicates is used and on the other hand, runtime performance is improved. To capture also problems complete for higher levels of the polynomial hierarchy, *D-FLAT* offers the possiblity to use a data structure called *item trees*. Item trees are a generalization of dynamic programming tables allowing to mimic the behavior of so-called *alternating Turing machines* [CKS81].

Apart from requesting the table mode as the data structure to use, we specify via Line 1 of Listing 2.2 that all instantiations of the predicates `pos` and `neg` in the input are considered for the edge relation of the graph. In this way, *D-FLAT* is instructed to construct a variable-clause incidence graph of the input instance which is then automatically decomposed. Per default, the leaves and the root of the computed tree decomposition are defined to be empty by *D-FLAT*. In this case, we do not need empty leaves. To reduce the cases which one has to distinguish during the traversal, we additional specify that we want the algorithm to operate on semi-normalized tree decompositions. Note that all configuration parameters of *D-FLAT* can also be specified via its rich command-line interface.

In the following explanation we will use so-called *item sets* to represent the partial solutions which are stored in the table data structure. Whenever we want to propagate information in *D-FLAT* from one step of the dynamic programming process to the next one, we have to store the respective pieces of information in the item set corresponding to an answer set. By accessing the child item sets we can access the partial solutions of the dynamic programming tables of a tree node's child. In the given encoding of the SAT problem we only store those atoms and clauses in the item set which are set to **True** or which are already satisfied, respectively. Accordingly, all atoms (clauses) which are not stored in the item set are considered to be **False** (unsatisfied).

In Lines 2 and 3 of Listing 2.2 we make this intuition explicit by stating that all bag elements which already existed in a child bag but which are not selected, are considered to be set to **False** or unsatisfied, respectively. In Line 4 we specify a non-deterministic guess which selects for each child node a table row, representing a partial solution, which shall be extended. Via Line 5 we prevent the case that in a join node the two extended child rows do not agree on the truth value of an atom. In Line 6 we get rid of all solution candidates where a clause is unsatisfied upon its removal. This constraint is crucial because a removed clause is never introduced again due to Criterion 3 (connectedness) of Definition 10 (tree decompositions) and so it can never become satisfied after its removal. Lines 7 and 8 are used to propagate relevant information about atoms set to **True** and the satisfied clauses from the child node(s) to the current node. In Line 9 we use a non-deterministic guess to select which of the introduced atoms is set to **True** and in the last two lines of Listing 2.2 we update the information which of the clauses in the current bag is already satisfied, analogously to Listing 2.1.

Note that *D-FLAT* is not limited to search problems. It can also handle optimization problems. To exemplify the convenience of using *D-FLAT* to solve this kind of problems

by means of dynamic programming on tree decompositions, consider, for instance, the enumeration variant of the problem Minimum Dominating Set defined as follows:

Input:	An undirected graph $\mathcal{G} = (V, E)$
Task:	Find all cardinality-minimal sets of vertices $S \subseteq V$ where for each $v \in V$ it holds that $v \in S$ or where there is an edge $(v, w) \in E$ such that $w \in S$!

Informally speaking, the Minimum Dominating Set problem asks for the smallest subsets of the graph's vertices such that each vertex of the graph is either contained in the set of selected vertices or adjacent to at least one vertex inside this set. The vertices which are not contained in S but adjacent to a vertex which is part of the set S are called *dominated*. It is a well-known fact that the decision variant of the problem, asking whether there exists a dominating set of size less than or equal to a limit k, is complete for the complexity class NP and that it is fixed-parameter tractable with respect to treewidth. Therefore, dynamic programming on tree decompositions is indeed a promising approach to solve the problem at hand.

```
1  %dflat: --tables -e vertex -e edge --no-empty-leaves -n semi
2  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
3  ← extend(R), extend(S), childItem(R,i(X)), not childItem(S,i(X)).
4  item(i(X)) ← extend(R), childItem(R,i(X)), not removed(X).
5  item(d(X)) ← extend(R), childItem(R,d(X)), not removed(X).
6  { item(i(X)) : introduced(X) }.
7  item(d(Y)) ← item(i(X)), edge(X,Y), current(X), current(Y).
8  ← removed(X), extend(R), not childItem(R,i(X)), not childItem(R,d(X)).
9  cost(C) ← initial, C = #count{ X : item(i(X)) }.
10 cost(CC + IC) ← numChildNodes == 1, extend(R), childCost(R,CC),
       ↪ IC = #count{ X : item(i(X)), introduced(X) }.
11 cost(C1 + C2 - CC) ← numChildNodes == 2, extend(R1), extend(R2),
       ↪ childCost(R1,C1), childCost(R2,C2), commonCost(R1,R2,CC).
12 commonCost(R1,R2,CC) ← childRow(R1,N1), childRow(R2,N2), N1 < N2,
       ↪ CC = #count { X : childItem(R1,i(X)), childItem(R2,i(X)) }.
```

Listing 2.3: *D-FLAT* Encoding for Minimum Dominating Set

Listing 2.3 shows a possible *D-FLAT* encoding for the problem Minimum Dominating Set. The first line of the provided listing is similar to the configuration statement that is used in Listing 2.2. The only difference between the two setups is the fact that in the case at hand we use input instances where the graph's edge relation is defined using predicates named `vertex` (with arity 1) and `edge` (with arity 2). The semantics of each of the two predicates follows the definition of the respective, graph-theoretic concept (see Definition 8 (graph)).

By Line 2 of Listing 2.3 we specify a non-deterministic guess which selects for each child node a table row, representing a partial solution, which shall be extended. In the next line, a constraint is used to discard all combinations of rows where the respective guesses for the vertices inside S – a vertex $x \in S$ is denoted by the item `i(x)` – are not consistent. In this way, we can be sure that we only extend child solutions which refer to the same dominating set. Line 4 then just propagates the information for all vertices which are not removed from the tree decomposition so far, i.e., those which are still in the bag of the current vertex. Line 5 works analogously for the dominated vertices $y \in S$, denoted by the item `d(y)`.

Line 6 of Listing 2.3 implements a non-deterministic guess whether an introduced vertex is in the dominating set or outside. Once it choice is made, via Line 7 we derive that all its neighbors which are occurring in the current bag are dominated. Line 8 discards all partial solutions where a removed vertex is neither inside the set acting as candidate for a dominating set nor dominated.

Lines 9 to 12 then define the costs of the solution. For each partial solution in a leaf node, the costs are determined simply by the number of vertices which are inside the dominating set. For exchange nodes, i.e., nodes with a single child, the costs of a solution are given by the sum of the corresponding solution for the child node and the number of introduced vertices which are assumed to be in the dominating set. For semi-normalized join nodes, i.e., nodes with two children having exactly the same bag content as the node under focus, the costs can be determined by simply following the inclusion-exclusion principle. That is, we have to add the costs for the partial solution from the first child to the costs for the partial solution of the second child and then we subtract the costs which they have in common regarding the current bag content. Note that the predicate `cost` (with arity 1) is interpreted by *D-FLAT* as the cost of a partial solutions and the framework automatically takes care that only solutions of minimal cost are retained.

By using the variable-clause incidence graph from Figure 2.1 we obtain a possible input instance for the *D-FLAT* encoding in Listing 2.3. The ASP encoding corresponding to the instance is given in Listing 2.4.

```
vertex(a). vertex(b). vertex(c). vertex(c1). vertex(c2). vertex(c3).
edge(a,c1). edge(a,c3). edge(b,c2). edge(b,c3). edge(c,c2). edge(c,c3).
```

Listing 2.4: An Instance of Minimum Dominating Set based on Figure 2.1

Because *D-FLAT* was instructed to consider the predicates `vertex` and `edge` as edge relation, the framework internally constructs the graph depicted in Figure 2.6 when called with the problem encoding from Listing 2.3 and the given input instance. For the reader's convenience, the only minimum dominating set (a and c_2) is highlighted in green and with thick border. In Figure 2.7 we provide a semi-normalized tree decomposition for the given instance of the Minimum Dominating Set problem.

The dynamic programming tables resulting from using the problem encoding from

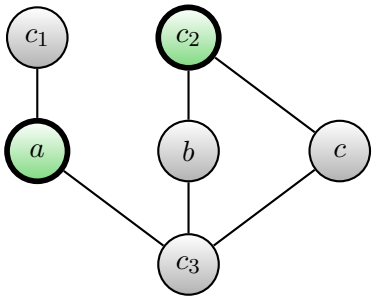


Figure 2.6: An Example Instance of Minimum Dominating Set

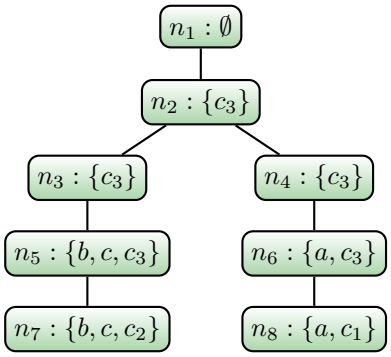


Figure 2.7: A Semi-Normalized Tree Decomposition with Empty Root n_1

Listing 2.3 together with the graph instance from Listing 2.4 and the corresponding, semi-normalized tree decomposition are shown in Figure 2.7. The tables are constructed similarly to those in the exemplification of dynamic programming on tree decompositions earlier in this chapter. The only difference is that we now use *D-FLAT*'s capabilities of handling solution costs in order to minimize the size of the dominating set. This time, the domain of values for each of the vertices is given by the assignments o, d and i, where i denotes that the corresponding vertex is part of the set of selected vertices. The value d means that the vertex is dominated and o stands for the fact that the vertex is outside the set of selected vertices and (currently) not dominated.

When we look at the tables shown in Figure 2.8, depicting the partial solutions for the Minimum Dominating Set problem we can see that in the step between the nodes n_7 and n_5, some solution candidates are not extended. For candidates $n_7|0$ this is not surprising as vertex c_2 is neither selected nor dominated. The explanation for discarding candidates $n_7|3$, $n_7|5$ as well as $n_7|7$ is given by the fact that those candidates are sub-optimal with regard to the solution size, i.e., we can always find a valid, smaller set which still fulfills the criteria for a cardinality-minimal dominating set. Note that *D-FLAT* automatically takes care that only those partial solutions are retained for which there is no counterpart with identical items but of smaller cost. In this way, when we finally reach the empty root node, we observe that there is only a single, optimal solution for

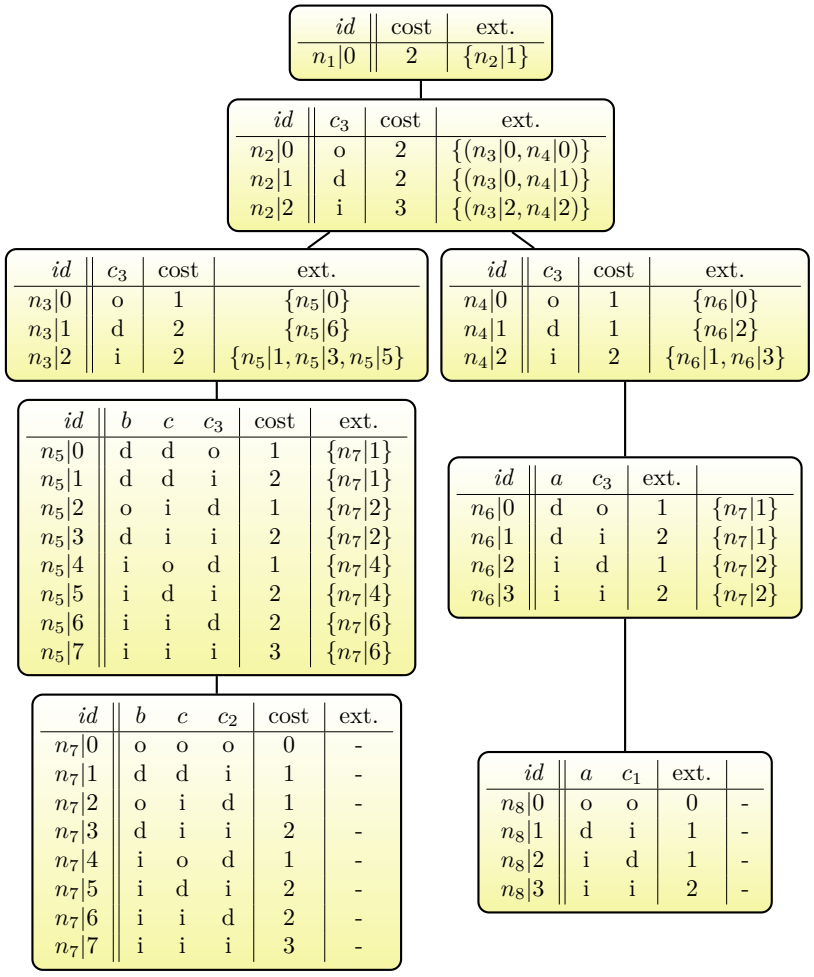


Figure 2.8: Solving Minimum Dominating Set using *D-FLAT*

the given instance of the Minimum Dominating Set problem, namely the dominating set $\{a, c_2\}$. The optimal solution can be reproduced by following the partial solutions $n_1|0, n_2|1, n_3|0, n_4|1, n_5|0, n_6|2, n_7|1, n_8|2$ and by observing that in these partial solutions the only vertices with value i are a and c_2.

As already mentioned in the introduction of this work, dynamic programming algorithms which operate on tree decompositions often suffer from poor robustness regarding their runtime behavior. In the next section we will therefore introduce an area of computer science which can help to get a handle on this issue.

2.4 Machine Learning

Machine learning is among the most prominent areas of research within the computer science discipline of artificial intelligence (AI). The foundations of this field of study were

laid in the 1950's based on seminal work by Samuel [Sam59]. This important article by Samuel – which is seen by many researchers as the publication that coined the term *machine learning* – is presenting a computer program which is able to play the game of Checkers. While, on the first glance, implementing the required software routines seems to be a straight-forward task for experienced programmers, the novelty of this program is the fact that it is able to learn. This means, Samuel's software is able to adapt itself according to the current situation without influence from the outside in terms of human input which enforces a re-configuration. Thus, the computer program by Samuel which uses machine learning techniques to increase its chance of winning the game of Checkers probably is the first piece of software capable of self-guided learning.

Since its origin, the area of machine learning research quickly gained attention in many fields of application. Such fields include medical diagnosis (e.g., detection of potential diseases), finance (e.g., predicting trends of stocks as well as of stock markets), behavioral advertising, text/image/video categorization, text and speech recognition, detection of viruses and spam e-mails, weather forecasts, computer games, search engines as well as autonomous cars/robots. Indeed, this list is not exhaustive because machine learning nowadays is finding its way into more and more areas of life. For instance, in a growing number of cities, machine learning algorithms help the police to better coordinate the officers on duty by predicting those areas of the city having an increased probability of crime being committed.

Typically, the umbrella term "machine learning" comprises the following forms of learning:

- Supervised learning:

  In tasks assigned to the area of supervised learning, the learning algorithm (or *learner* for short), in general, receives a set of labeled training examples, usually consisting of a vector of input values and the desired output value (representing the label of the training example at hand). Based on these ingredients, the goal of the learner is to infer a function, the so-called *model*, from the provided training data which allows to predict the outcome for unseen examples for which only the vector of input values is known.

- Reinforcement learning:

  Especially in the context of autonomous systems, techniques from the area of reinforcement learning are used. The idea behind this approach is that a machine, robot or software agent shall learn and improve its behavior based on the feedback it receives from its environment in terms of so-called *reinforcement signals*. For instance, a robot can learn how to navigate efficiently through a room by receiving negative feedback when a collision with an obstacle (e.g., a chair standing the robot's way) is detected.

  Note that the distinction between supervised and reinforcement learning is given by the different assessment of solution quality: In the case of a supervised learning task, the quality of the learned model can be quantified in a relatively easy way by

measuring the deviation between the predicted and the actual labels. In contrast, in the context of reinforcement learning, the quality of a model is measured by means of the collective reward or penalty which is achieved using the learned behavioral rules.

- Unsupervised learning:

  Sometimes, the data which is given to a learner is completely unlabeled and also there is no environmental feedback available. In these cases, techniques from the area of unsupervised learning come into play. The goal of such techniques is to unveil hidden structure in the provided input data and to describe it by means of functions. Probably the most prominent representative of unsupervised learning techniques is cluster analysis (or clustering for short) [And73, ELLS11].

In the remainder of this introduction to machine learning we will focus on supervised learning by means of so-called *regression algorithms*. For surveys and more details on the wide range of other topics from the area of machine learning, see, e.g., [Mit97, Bis06, MRT12].

Regression is a prominent technique from the area of supervised learning. Informally speaking, regression represents a generalization of *classification*. Given a list of allowed *classes* (potential labels) and an unlabeled example consisting of a set of input values, a classification task asks for the actual class the measurement belongs to. While the classes used in the context classification tasks are always discrete values, the domain of the output value of a regression task is continuous. In the context of regression algorithms, the different input values associated with a given example are often called *explanatory variables* (or *features*) and the value which is to be predicted is often referred to by the term *response variable*.

Body Height (cm)	**Body Mass (kg)**
170	80.0
180	90.0
190	100.0
176	?
194	?

Table 2.1: Example of a Regression Task

To illustrate the idea behind the technique of regression by means of a small example, have a look at Table 2.1. The table shows in the first column the body height of a person and in the second column the body mass of the respective person is provided. Hence, in our simple example, an individual has just two features, namely its body height and the mass. For our example, assume that the only explanatory variable is the body height and the body mass acts as the response variable which is to be predicted.

The group of three persons at the top of Table 2.1 is called the *training set*. This term is based on the fact that this is the group of individuals upon which the actual learning takes place. The so-called *evaluation set* in our example consists of the remaining two rows where the body mass is currently not known. Often, the evaluation set is also called *test set*. The goal of regression is to compute a model which can be used to predict the body mass of each of the individuals in the test set with highest possible accuracy based on the given explanatory variables. Here, the term *model* refers to the materialization of the learned relationship between the explanatory variables and the response variable.

For our running example, it is not hard to see that the body masses in the training set can be perfectly explained by the formula $h - 90$ where h denotes the individual's body height. While this seems trivial in our minimalist example, in almost all real-world application scenarios finding a function which allows to predict the response variable with high accuracy is far from easy.

When we follow the simple function we learned from the training set of our example regression task, we can assume that the body masses of the remaining two persons are 86.0 and 104.0 kilograms. As long as the coincidence between prediction and actual value is given, everything is fine, but especially when there are more features involved (like, for instance, the sex or age of the investigated persons) or when there are more individuals to consider, a regression task can relatively soon become very complex. For instance, assume that we add an additional individual having body height 160cm and a body mass of 67kg to the training set from Table 2.1. In this case, it is clear that the perfect conformity of the body mass to our formula $h - 90$ is no longer given, even when we consider only the training set, and that we observe a certain degree of deviation between our prediction and the actual measurement. This deviation is also called the *error*.

Often, the prediction quality of a regression model with respect to a certain training set is quantified in terms of the *Pearson correlation coefficient* (also called *Pearson's r* or *Pearson product-moment correlation coefficient*) [KK62]. Furthermore, common notions of error in the context of regression tasks are, for instance, the *mean (absolute) deviation* or the *root mean squared error* (see, e.g., [KK62] for details on these measures).

In order to minimize the undesired error and to increase correlation between prediction and actual observation, researchers proposed a multitude of different regression algorithms. Among the most prominent representatives of these algorithms we find, for instance, linear regression [KK62], regression trees [Qui92], nearest-neighbor algorithm [AKA91], least median of squares [RL05] as well as support-vector machines for regression [SKBM00, SS04], to name just a few. The *no-free-lunch theorem* [Wol96] states that there is no algorithm which performs best in every possible situation. Accordingly, it strongly depends on the actual application scenario which of the available regression algorithms delivers the highest accuracy. We also observe that sometimes even relatively simple approaches provide good prediction quality while in other cases, more sophisticated algorithms have to be employed in order to allow for reliable predictions.

To evaluate the quality of a given model, the technique of 10-*fold cross validation* is

common in the area of machine learning as it allows to estimate the expected error for unseen examples without manual separation of the input data set into training and evaluation set. Instead, 10-fold cross validation works by quantifying the error of the learned model with respect to the very same input data set which is split into ten distinct sets of examples. After dividing the input data set into ten parts, each of these subsets is taken once as evaluation set whereas the remainder with respect to the complete data set is considered as the training set for the so-called *fold* at hand. In this way, ten separate evaluations of the Pearson correlation coefficient and/or different error measures are performed. Hence, the estimation of the quality of a model is likely to be more accurate than in those cases where just a single split of the input data set into training and evaluation set is considered.

Once a regression model is trained and the error is acceptably small, we can use it to predict the response variable for unclassified examples, i.e., examples for which the actual value of the response variable is currently unknown. Note that although many regression algorithms perform very well concerning their prediction quality, they are always bound to the training data. Therefore, it is often crucial to prepare the input data for machine learning algorithms in order to achieve best results.

Furthermore, due to the decisive impact of the features which are considered on the learning tasks, an important step on the way towards optimal prediction quality is *feature engineering*. Roughly speaking, the term "feature engineering" refers to the process of defining the variables which are used in a machine learning task like classification or regression. A specific example from an input data set in the context of a machine learning task, often also called an *individual*, is identified by a combination of *features* and the corresponding *class value*. In our example scenario from before, the only two features of a human being are its body height and the mass where the latter acts as *class variable*. Indeed, it is apparent that there is much more to consider if we aim for good prediction quality in a real-world setting. For instance, other promising features could be the sex, age, the home country as well as the wealth of the investigated individuals. In practice, finding an appropriate set of features for a given problem statement is often far from trivial.

In Section 3 we employ regression models to predict the expected runtime of dynamic programming algorithms based on characteristic, problem-independent features of the underlying tree decomposition on which these algorithms operate. Using machine learning techniques for runtime prediction gained a lot of attention due to its wide-ranging applicability in many scenarios. Recently, researchers have successfully applied machine learning for runtime prediction and algorithm selection to several problem domains. Such problems include, among a variety of others, SAT [XHHL08, HXHL14], combinatorial auctions [LNS09], the traveling salesperson problem [SMvHL10, KCHS11, MBT$^+$13, PM14, HXHL14] as well as Graph Coloring [SWLI13, MS13].

The aim of runtime prediction is to make accurate predictions of the actual running time of a given algorithm based on features of the input instance. Going one step further, tools from the domain of algorithm selection are used to select the most promising algorithm

(i.e., the one with lowest predicted time consumption) from a pool of available ones based on the problem instance at hand. For surveys on the domain of runtime prediction and algorithm selection, see, e.g., [Smi08, HXHL14, Kot14].

We note at this point that the goal we pursue in our application scenarios presented in the thesis at hand is not primarily to achieve a perfect correspondence between actual and predicted solving time for a given problem instance (the basic intention of runtime prediction). Also, we do not follow the idea to select the optimal algorithm to process a given task like it is done in the domain of algorithm selection. Instead, our approach needs only a single dynamic programming algorithm capable of solving the problem at hand. Once such an algorithm exists, we use machine learning techniques to select the most promising tree decomposition from a pool of generated ones for a given problem instance before actually running the dynamic programming algorithm on it. In general, the most promising decomposition in this context is the one for which the predicted time consumption for the given problem instance is minimal. Hence, although there are the aforementioned differences, what is common between our approach, general runtime prediction and algorithm selection is the overall goal, namely to save valuable time in practical scenarios.

CHAPTER 3

The Impact of Tree Decomposition Selection

As mentioned in the previous chapter, from a theoretical point of view, the actual width k of a tree decomposition is the crucial parameter towards efficiency for FPT algorithms that use those decompositions. Unfortunately, experience shows that there is more to consider than just the plain width in order to achieve highest efficiency in practical application scenarios. Morak et al. [MMP+12], for instance, suggested that the consideration of further properties of tree decompositions is important for the runtime of dynamic programming algorithms for answer set programming. In another paper, Jégou and Terrioux [JT14] observed that the existence of multiple connected components in the same tree node (bag) may have a negative impact on the efficiency of solving constraint satisfaction problems.

In this chapter we now go one step further by considering a much richer set of tree decomposition features for the estimation function of the runtime of dynamic programming algorithms. More precisely, in this chapter we want to gain a deeper understanding of the impact of the shape of tree decompositions on the actual runtime of DP algorithms by employing techniques from the area of machine learning.

In Section 2.4, we already referred to some success stories in which machine learning was used for runtime prediction and algorithm selection. Our research gives new contributions in this area as, to the best of our knowledge, this is the first application of machine learning techniques towards the optimization of tree decompositions in DP algorithms. To this aim we propose new original features for tree decompositions that allow for a reliable prediction of the influence of a given tree decomposition on the performance of dynamic programming algorithms. Further, to select the most promising tree decomposition from a pool of generated ones using those features we propose an approach that applies machine learning techniques. We create the appropriate training sets by conducting extensive

experiments on different problem domains, instances and tree decompositions. Moreover, we run our experiments on a state-of-the-art system that applies dynamic programming algorithms on tree decompositions, namely *D-FLAT* [ABC+14b]. *D-FLAT* is a problem-independent general-purpose framework designed for (relatively) easy prototyping of DP algorithms (see also Section 2.3).

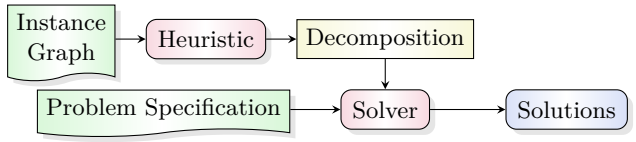
The complete picture of our evaluation shows a significant benefit of selecting a tree decomposition that is promising according to the prediction in contrast to simply choosing an arbitrary one. The results also confirm that our approach is generally applicable and independent from the particular problem domain. Hence, relying only on the width as a measure for the quality of a tree decomposition appears to be a too narrow approach, and we see the strong need for new, enhanced notions which allow for a better discrimination between different tree decompositions of the same instance. In our experimental evaluation we show that our proposed features are indeed promising candidates for these new quality measures. Finally, the results provide valuable insights for laying the foundation to construct customized decompositions optimizing the relevant features.

An additional evaluation conducted with another tree-decomposition based system, namely *SEQUOIA* [KLR11], can be found in an accompanying technical report [AMW16c]. The results for the *SEQUOIA* system are very similar to those we observe in the chapter at hand. Although in the case of *SEQUOIA*, the variation in terms of solving time between different random seeds (and hence, different tree decompositions) for the same problem instance is relatively small, predicting the runtime of DP algorithms based on tree decomposition features works very well. Because the prediction seems to be more involved in the case of *D-FLAT* and also due to the fact that the edge weights used in the context of the Steiner Tree problem (see Section 3.2.1) are not handled by *SEQUOIA*, we focus in this chapter on *D-FLAT*.

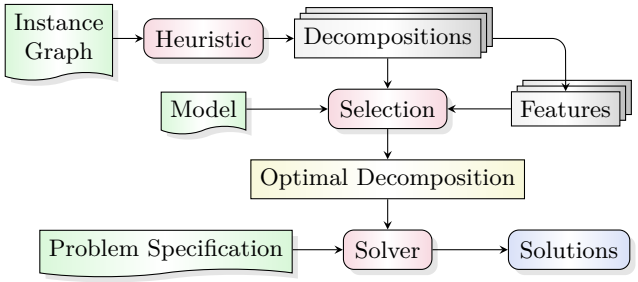
The remainder of this chapter is organized as follows. In Section 3.1 we propose a novel approach on how to improve the performance and robustness of dynamic programming on tree decompositions and in Section 3.2 we provide an extensive experimental evaluation on random input data and real-world instances. Section 3.3 finally concludes this chapter.

3.1 Improving the Efficiency of DP Algorithms

Systems such as *D-FLAT* follow a straight-forward approach for using tree decompositions: A single decomposition is generated heuristically and then fed into the DP algorithm used by the system (see Figure 3.1a). However, experiments have shown that the "quality" of such tree decompositions varies, leading to significantly differing runtimes for the same problem instance. Most interestingly, "quality" in this context does not necessarily mean low width. Even tree decompositions of exactly the same width lead to huge differences in the observed solving times. For instance, in our experiments for the Steiner Tree problem on real-world instances (see Section 3.2.3) we observe runtimes between 67 seconds and around two hours for the problem instance `vienna/metro_10terminals_46` for which all the tree decompositions used in our evaluation are of width 5.



(a) Standard Approach



(b) Improved Approach

Figure 3.1: Comparison of Approaches

3.1.1 Automated Selection of Tree Decompositions

The approach we propose in this work is illustrated in Figure 3.1b. The main idea is to generate a pool of tree decompositions for the given input instance and then to select, based on features of the decomposition, the one which promises best performance. The key aspects of the approach are as follows:

- Generating a number of tree decompositions for a given input graph can be usually done very efficiently by employing sophisticated heuristics for tree decompositions like, e.g., Min-Fill heuristic [Dec03]. Thus, the runtime overhead for computing a pool of them will be negligible in most cases compared to the overall runtime of the dynamic programming algorithm.
- Models allowing to predict the influence of a tree decomposition on the runtime behavior of a given DP algorithm are required. These models can be obtained in an off-line training-phase by running several instances with different tree decompositions and by storing the runtime and the feature values which are then processed by machine learning algorithms. For our purposes, machine learning techniques need to predict a good *ranking* of tree decompositions based on the predicted runtime for these decompositions. We note that a very accurate prediction of runtime is not crucial in our case, but rather predicting a correct order of tree decompositions. For example, if running a DP algorithm using tree decomposition *TD1* is faster than with *TD2*, it is important that machine learning algorithms predict that the runtime for *TD1* is shorter than the runtime for *TD2*.
- The main challenge and novel aspect of the approach is given by the fact that the features used to obtain these rankings need to be defined on the tree decomposition,

not on the given problem instance. This is because instance features only help to distinguish instances but they do not help us to choose a proper decomposition as they are the same for each of the generated decompositions. To successfully apply learning techniques we need powerful features that characterize tree decompositions. Moreover, the computation of these features needs to be done efficiently.

In other words, our approach works as follows. First, a number (which can be arbitrarily large) of tree decompositions of the given problem instance is computed and stored in a pool. Second, the features (acting as explanatory variables) of these decompositions are extracted and used to predict the runtime as the response variable. This gives us a ranking from which we select the decomposition with minimal predicted runtime (in case of ties, we choose randomly one of the decompositions having minimal predicted runtime). We apply several regression algorithms to generate the models for prediction. Finally, the selected decomposition is handed over to the system which runs the DP algorithm.

Note at this point that the model(s) as well as the features are crucial ingredients for the applicability of our approach in practice. Indeed, it should be possible to compute each feature efficiently. Furthermore, it can be a time-consuming task to train a regression model. Fortunately, as we will show in Section 3.2.5, it seems that models which were trained for some specific problem domain can often be re-used in different application scenarios. In the following section we will propose a set of several tree decomposition features which are all computable in polynomial time. In many cases, this is possible even in linear time.

3.1.2 Tree Decomposition Features

In what follows we address one of the main contributions of the work, namely the identification of new tree decomposition features. As it is common practice to define dynamic programming algorithms on normalized tree decompositions (see, e.g., [BK08, KLR11]), we restrict ourselves here to this type of decomposition. That this restriction does not affect the generality of the proposed approach is shown implicitly in Chapter 5. There, also features of semi-normalized and non-normalized tree decompositions are considered for the selection of promising decompositions. Subsequently we will give for each feature a short description and formal specification. Providing multiple statistical key figures like minimum, maximum, mean and median leads us to a total of 144 features.

Before we present the collection of tree decomposition features, let us fix the formal notation we will use in the corresponding formulae. We assume a given (normalized) tree decomposition TD $(\mathcal{T}, \chi)$ with $\mathcal{T} = (N, E_{\mathcal{T}})$ of a graph $\mathcal{G} = (V, E)$. One of the nodes in N thereby acts as the root of the decomposition. Each node $i \in N$ has associated a type $t_i \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\}$. Furthermore, we define the sets *Leaf*, *Introduce*, *Forget* and *Join* to contain exactly the nodes from TD where t_i matches the name of the set. The bag content of a node i is denoted by $\chi(i)$. The set *NonLeaf* is defined as $N \setminus \mathit{Leaf}$ and the set *NonEmpty* covers all nodes $i \in N$ where $|\chi(i)| > 0$. The distance between two nodes i and j (in $\mathcal{T}$) is given by the function $\mathit{distance}(i, j)$. By l_i we denote

the *level* (also called *depth*) of a node i, given by the distance between the root and i. The level of the root r is thus 0. The set of children of node i is denoted by $\mathit{Children}_i$. The set N_v associated with a vertex $v \in V$ is the set of decomposition nodes $i \in N$ such that $v \in \chi(i)$. Some of the more elaborate features require information about the neighborhood and the reachability relation. For this reason we define the following functions:

- $\mathit{neighbors}(v)$ returns the set of neighbors of vertex v (excluding v) in $\mathcal{G}$;.
- $\mathit{adjacent}(u, v)$ returns 1 whenever vertices u and v are adjacent in $\mathcal{G}$ and 0 otherwise;
- $\mathit{reachable}(u, v)$ returns 1 whenever v can be reached from u in $\mathcal{G}$ and 0 otherwise.

Most of the features we present and utilize in this chapter rely on the use of aggregates. In order to avoid redundancies we employ two slightly different sets of aggregates in the actual formulae shown subsequently. Note that we give here just the simple enumeration of the aggregates we use for our experiments. When implementing our approach one can use (almost) all possible subsets and every superset of the following sets.

- $\mathit{Agg1} = \{\mathit{count}, \mathit{min}, \mathit{max}, \mathit{mean}, \mathit{median}, \mathit{sd}$ (Standard Deviation) $\}$
- $\mathit{Agg2} = \mathit{Agg1} \setminus \{\mathit{count}\}$

The proposed features are described below.

BagSize, NonLeafNodeBagSize, NonEmptyNodeBagSize These feature shall capture the complexity of the tree decomposition by recording the size of the bags. We have for each $\alpha \in \mathit{Agg1}$

$$\begin{aligned}
\mathit{BagSize}^{\alpha} &= \alpha(\{|\chi(i)| : i \in N\}) \\
\mathit{BagSize}^{\alpha}_{t} &= \alpha(\{|\chi(i)| : i \in t\}) \quad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \\
\mathit{BagSize}^{\alpha}_{\mathit{NonLeaf}} &= \alpha(\{|\chi(i)| : i \in \mathit{NonLeaf}\}) \\
\mathit{BagSize}^{\alpha}_{\mathit{NonEmpty}} &= \alpha(\{|\chi(i)| : i \in \mathit{NonEmpty}\})
\end{aligned} \tag{3.1}$$

Additionally, to cover the overall size of the decomposition, we have

$$\begin{aligned}
\mathit{CumulativeBagSize} &= \sum_{i \in N} |\chi(i)| \\
\mathit{CumulativeBagSize}_t &= \sum_{i \in t} |\chi(i)| \quad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \\
\mathit{DecompositionOverheadRatio} &= \left(\sum_{i \in N} |\chi(i)|\right) / |V|
\end{aligned} \tag{3.2}$$

ContainerCount By the container count of a vertex v we refer to the number of bags a vertex v of the original graph appears in. For each $\alpha \in \mathit{Agg2}$ we have

$$\mathit{ContainerCount}^{\alpha} = \alpha(\bigcup_{v \in V} \{|\{i : i \in N, v \in \chi(i)\}|\}) \tag{3.3}$$

Note that $\mathit{ContainerCount}^{mean} = \mathit{DecompositionOverheadRatio}$. In this special case, the features are indeed equivalent. In practice, one can avoid redundancy by using only a single one of these two measures.

ItemLifetime This feature is very similar to *ContainerCount*, but this time we only count the number of distinct levels of the tree decomposition the vertex appears in. For each $\alpha \in \mathit{Agg2}$ we have

$$\mathit{ItemLifetime}^{\alpha} = \alpha(\bigcup_{v \in V} \{|\{l_i : i \in N, v \in \chi(i)\}|\}) \tag{3.4}$$

NodeDepth, NonLeafNodeDepth, NonEmptyNodeDepth These features of a given tree decomposition aggregate the distance from the root node to the respective nodes under focus. For each $\alpha \in \mathit{Agg2}$ we have

$$\begin{aligned} \mathit{NodeDepth}^{\alpha} &= \alpha(\{l_i : i \in N\}) \\ \mathit{NodeDepth}^{\alpha}_{t} &= \alpha(\{l_i : i \in t\}) \quad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \\ \mathit{NodeDepth}^{\alpha}_{\mathit{NonLeaf}} &= \alpha(\{l_i : i \in \mathit{NonLeaf}\}) \\ \mathit{NodeDepth}^{\alpha}_{\mathit{NonEmpty}} &= \alpha(\{l_i : i \in \mathit{NonEmpty}\}) \end{aligned} \tag{3.5}$$

Percentage This feature records the overall percentage of the respective node type in the tree decomposition at hand.

$$\mathit{Percentage}_t = |\{i : i \in t\}|/|N| \quad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \tag{3.6}$$

JoinNodeDistance Join nodes often have a strong influence on the runtime of DP algorithms as they have the potential to increase (or decrease) the number of valid solution candidates drastically. This feature keeps track of the distance between join nodes and it is 0 in the case that not more than a single join node is present in the decomposition. In case that two or more join nodes are present, the distance is measured for each pair i and j of join nodes separately by taking the length of the path between i and j in the decomposition. In case that not more than one join node is present, the following features are set to 0. Otherwise, for each $\alpha \in \mathit{Agg2}$ we have

$$\mathit{JoinNodeDistance}^{\alpha} = \alpha(\{\mathit{distance}(i, j) : i, j \in \mathit{Join}, i \neq j\}) \tag{3.7}$$

BranchingFactor This feature measures the number of children for each node within the tree decomposition. For each $\alpha \in \mathit{Agg2}$ we have

$$\mathit{BranchingFactor}^{\alpha} = \alpha(\{\mathit{Children}_i : i \in N\}) \tag{3.8}$$

BagAdjacencyFactor This feature measures the ratio of the number of pairs of vertices in the bag that are adjacent in the original graph $\mathcal{G}$ and the total number of vertex pairs in the bag. For each $\alpha \in \mathit{Agg2}$ we have

$$BAF^{\alpha} = \alpha\left(\left\{\frac{|\{(u,v) : u,v \in \chi(i), u \neq v, \mathit{adjacent}(u,v)\}|}{\mathit{max}(1, |\chi(i)| * (|\chi(i)| - 1))} : i \in N\right\}\right) \tag{3.9}$$

BagConnectednessFactor This feature relates the number of pairs of vertices in the bag that are connected in the original graph $\mathcal{G}$ to the total number of vertex pairs in the bag. For each $\alpha \in \mathit{Agg2}$ we have

$$BCF^{\alpha} = \alpha\left(\left\{\frac{|\{(u,v) : u,v \in \chi(i), u \neq v, \mathit{reachable}(u,v)\}|}{\mathit{max}(1, |\chi(i)| * (|\chi(i)| - 1))} : i \in N\right\}\right) \tag{3.10}$$

BagNeighborhoodCoverageFactor For each vertex in the bag the ratio between the number of neighbors in the bag to the number of neighbors in the original graph is computed. The value for a single bag i is computed by averaging over all values for the vertices in $\chi(i)$. For each $\alpha \in \mathit{Agg2}$ we have

$$BNCF^{\alpha} = \alpha\left(\left\{\mathit{mean}\left(\left\{\frac{|\mathit{neighbors}(v) \cap \chi(i)|}{|\mathit{neighbors}(v)|} : v \in \chi(i)\right\}\right) : i \in N\right\}\right) \tag{3.11}$$

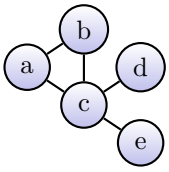
[Introduced | Forgotten]VertexNeighborCount Experience shows that the simple propagation of information is in many cases not the bottleneck for DP algorithms. In fact, most of the "real" work has to be done when vertices are introduced or forgotten and the algorithm has to evaluate rules and check constraints concerning the new (forgotten) vertices and their neighbors in the current bag. These two features are dedicated exactly to this issue. For each $\alpha \in \mathit{Agg2}$ we have, based on the introduced (forgotten) vertices X_i for bags i

$$NC^{\alpha} = \alpha\left(\{|\mathit{neighbors}(v) \cap \chi(j)| : i \in N, v \in X_i, j \in N_v\}\right) \tag{3.12}$$

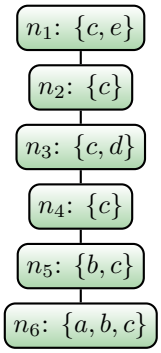
[Introduced | Forgotten]VertexConnectednessFactor Closely related to the count of neighbors for introduced and forgotten vertices is the connectedness factor. For the last two features used in this chapter we measure the ratio between the number of vertices in the bag connected to a introduced (forgotten) vertex v in the original graph and the total number of all possible connections between all nodes in the bag. For each $\alpha \in \mathit{Agg2}$ we have, based on the introduced (forgotten) vertices X_i for bags i

$$CF^{\alpha} = \alpha\left(\left\{\frac{|\{(u,v) : u,v \in \chi(j), u \neq v, \mathit{reachable}(u,v)\}|}{\mathit{max}(1, |\chi(j)| * (|\chi(j)| - 1))} : i \in N, v \in X_i, j \in N_v\right\}\right) \tag{3.13}$$

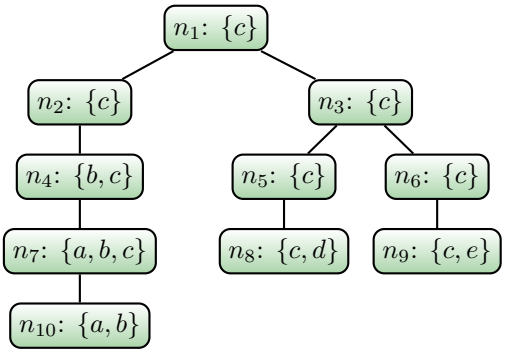
Example 2. *To demonstrate how the proposed features help to distinguish different tree decompositions of a given input graph consider Figure 3.2 which shows a graph $\mathcal{G}$ together with two corresponding, normalized tree decompositions. A selection of their features is provided in Table 3.1.*



(a) Example Graph $\mathcal{G}$



(b) *TD1*



(c) *TD2*

Figure 3.2: Graph $\mathcal{G}$ with Normalized Tree Decompositions *TD1* and *TD2*

Feature	**TD1**	**TD2**
$BagSize^{median}$	2	1.5
$CumulativeBagSize$	11	16
$DecompositionOverheadRatio$	2.2	3.2
$ContainerCount^{median}$	1	2
$ItemLifetime^{median}$	1	2
$NodeDepth^{median}$	2.5	2
$JoinNodeDistance^{median}$	0	1
$BagAdjacencyFactor^{median}$	1	1
$BagConnectednessFactor^{median}$	1	1
$BagNeighborhoodCoverageFactor^{median}$	0.5	0.19
$IntroducedVertexNeighborCount^{median}$	1	2
$ForgottenVertexNeighborCount^{median}$	1	1
$IntroducedVertexConnectednessFactor^{median}$	1	1
$ForgottenVertexConnectednessFactor^{median}$	1	1

Table 3.1: Subset of Extracted Features for Decompositions *TD1* and *TD2* of Graph $\mathcal{G}$

We can see that both decompositions TD1 and TD2 have a maximum bag size of 3; hence they have exactly the same width – namely 2 – but their actual difference is clearly reflected by the proposed features (see Table 3.1). For brevity we only provide the median for those features where multiple aggregates are applied. □

Note that the set of tree decomposition features we present here shall act as a starting point. It contains various features which quantify the structural parameters of a given (normalized) tree decomposition, but it also includes several measurements which are related to the general runtime behavior of dynamic programming algorithms, like, e.g., the neighborhood-related features. Although we focus solely on tree decomposition features in this chapter, considering also problem-specific features may improve prediction quality in concrete scenarios. In the following experiments we will use the complete set of our proposed features, but one can also try to drop some of them, e.g., by using well-established feature selection techniques from the area of machine learning (see, e.g., [GE03, CS14] for an overview of feature selection approaches).

3.2 Experimental Evaluation

In this section, we experimentally evaluate the proposed method. All our experiments were performed on a single core of an Intel Xeon E5-2637@3.5GHz processor running Debian GNU/Linux 8.3 and each test run was limited to a runtime of at most six hours and 64 GB of main memory.

We evaluate our approach based on the state-of-the-art dynamic programming framework *D-FLAT* 1.0.1. The machine learning tasks were carried out with *WEKA* 3.6.13 [HFH+09]. The full benchmark setup (including instance files, programs, configurations, problem encodings and all results) is, together with a short description on how to reproduce the results of the experiments, available at: `http://dbai.tuwien.ac.at/research/project/dflat/features_2016_03.zip`

3.2.1 Methodology

Problems In our analysis we consider the following set of problems defined on an undirected graph $\mathcal{G} = (V, E)$. Note that we use enumeration problems here as *D-FLAT* per default always computes all solutions for a given problem instance.

1. Minimum Dominating Set (MDS): Find all sets $S \subseteq V$ of minimal cardinality, such that for all vertices $u \in V$ either $u \in S$ or there is an edge $(u, v) \in E$ with $v \in S$!

2. 3-Colorability (Col): Find all valid 3-colorings of $\mathcal{G}$, i.e., all mappings of the vertices $v \in V$ to colors r, g and b such that no two adjacent vertices have assigned the same color!

3. Perfect Dominating Set (PDS): Find all sets $S \subseteq V$ of minimum size meeting the following requirements:
   - S is a dominating set of $\mathcal{G}$.
   - $\forall x \in S : x$ dominates at most one $y \in (V \setminus S)$.
   - $\forall y \in (V \setminus S) : y$ is dominated by exactly one $x \in S$.
4. Connected Vertex Cover (CVC): Find all sets $S \subseteq V$, such that for all $(u, v) \in E$, $u \in S$ or $v \in S$, and such that the vertices in S form a connected subgraph of $\mathcal{G}$!

Furthermore, we consider the following problem defined on undirected graphs $\mathcal{G} = (V, E)$ with edge weights $E \to \mathbb{N}$:

5. Steiner Tree (ST): Given a set of terminal vertices $T \subseteq V$, find all sets of edges $X \subseteq E$ of minimum total weight which meet the following requirements:
   - For every $t \in T$ there is an $e \in X$ containing t.
   - The graph formed by the edges in X is connected.

We note that the goal of this work is not to outperform existing, specialized state-of-the-art solvers for the respective problem domains but to improve the performance and robustness of dynamic programming algorithms on tree decompositions. Indeed the methods based on tree decomposition are exact techniques and currently can usually solve only problems of limited size.

Machine Learning Algorithms In our experiments we apply 16 models which are computed using *WEKA*'s regression algorithms which have been used successfully in different application domains. For each of the five problems and for each solver for the respective problem at hand, the following machine learning algorithms were considered.

1. GaussianProcesses (weka.classifiers.functions.GaussianProcesses) [Mac98]
2. IsotonicRegression (weka.classifiers.functions.IsotonicRegression) [BB72]
3. LeastMedSq (weka.classifiers.functions.LeastMedSq) [RL05]
4. LinearRegression (weka.classifiers.functions.LinearRegression) [KK62]
5. MultilayerPerceptron (weka.classifiers.functions.MultilayerPerceptron) [PBPPM09]
6. PaceRegression (weka.classifiers.functions.PaceRegression) [Wan00, WW02]
7. PLSClassifier (weka.classifiers.functions.PLSClassifier) [HK04]

8. SMOreg (weka.classifiers.functions.SMOreg) [SKBM00, SS04]
9. IBk (weka.classifiers.lazy.IBk) [AKA91]
10. KStar (weka.classifiers.lazy.KStar) [CT95]
11. LWL (weka.classifiers.lazy.LWL) [AMS97, FHP02]
12. AdditiveRegression (weka.classifiers.meta.AdditiveRegression) [Fri02]
13. Bagging (weka.classifiers.meta.Bagging) [Bre96]
14. CVParameterSelection (weka.classifiers.meta.CVParameterSelection) [Koh96]
15. M5Rules (weka.classifiers.rules.M5Rules) [HHF99]
16. M5P (weka.classifiers.trees.M5P) [Qui92]

The initial evaluation which was used to find the exact configuration for the regression algorithms considers all parameters provided by *WEKA* and was done on a separate benchmark set consisting of 500 tree decompositions (50 instances of 3-Colorability with 10 decompositions for each instance). For each parameter available in *WEKA* we experimented with different values and used 10-fold cross validation to determine the performance of each configuration. The fixed parameters, which can be found in [AMW16c], were used for all problem domains investigated in this chapter.

Training Set All aforementioned machine learning algorithms were trained separately for each problem using a training set consisting of 800 independent benchmark runs. These runs are obtained by investigating 20 satisfiable[1] instances and by considering 40 different tree decompositions (generated using the Min-Fill heuristic [Dec03]) for each of the problem instances. In addition to restricting the training set to satisfiable instances, we also ensured that the training set contains no benchmark runs which exceeded the allowed time or the memory limit to definitively rule out biased results.

The problem instances we used in our experiments are of different size and also the probability of whether an edge exists between two vertices of the input graph varies for different instances. While these variations are automatically present in the real-world instances we investigated, we applied the Erdős-Rényi random graph model to achieve an appropriate level of randomness for the constructed instances. For these random instances, we used three graph sizes and three different edge probabilities per problem to construct the corresponding training set.

[1]Not all generated instances are satisfiable for 3-Col or CVC. In our experiments for these problems, we consider only those that are, because for unsatisfiable instances, a large part of the tree decomposition might not even be visited by a DP algorithm. This is due to the fact that the algorithm terminates as soon as it is evident that no solution exists for the instance at hand. Therefore, unsatisfiable problem instances do not allow us to investigate the effect of decomposition selection on the actual runtime of DP algorithms and we thus omit them in our comparisons.

After the termination of a test run we extracted all of our proposed tree decomposition features (in our experiments, this took at most two seconds even for the largest instances) and stored the outcome together with the runtime achieved by the dynamic programming algorithm. When all test runs for a problem instance were finished, we had to normalize the results in order to make sure that each instance contributes equally to the computation of the machine learning models. This normalization step is done by standardizing these values X feature-wise based on the formula $(X-\mu)/\sigma$. An example for this normalization is the following: Assume that $2, 4$ and 6 are three evaluations for a feature X, obtained from three different tree decompositions of a problem instance. Obviously, the mean μ of these values is 4 and the standard deviation σ is 2. Hence, after standardization, we obtain the values $-1, 0$ and 1. When we consider another problem instance where feature X takes the values $1, 2$ and 3 (when given three tree decompositions of the instance), we again end up with the normalized values $-1, 0$ and 1. In this way, the previously different domains of the feature values for the two instances become comparable.

Evaluation Set The evaluation set for the computed models consists of 2000 benchmark runs per problem domain. These runs are obtained by running 50 problem instances with 40 different tree decompositions. Again, we ensured that unsatisfiable instances and such that violate the limits are excluded.

For a given instance, the actual evaluation is done by predicting the normalized runtime the problem-specific DP algorithm will need to solve the problem. We do this for each model and for each of the 40 tree decompositions. Afterwards, we select for each model the tree decomposition with the minimum runtime predicted by the respective model (ties are broken randomly). All that remains is to simply lookup the real, non-normalized runtime and compare it with the median runtime (the runtime the "average" decomposition would lead to) over all the tree decomposition for the given problem instance.

The value for the runtime improvement is computed by subtracting the quotient of the selected decomposition's actual runtime and the median runtime from 1. This means that a result of 0 implies that no improvement could be made. For the utopistic case that we are able to save 100% of the runtime, i.e., when the dynamic programming algorithm needs no time to solve the problem instance using the selected decomposition, we would obtain a result of 1. Every value less than 0 means that we observe a deterioration of the performance using the respective model as runtime predictor.

As the runtime improvement is strongly dependent on the shape of the problem instance at hand, we also investigate the predicted rank. This measurement refers to the rank the tree decomposition predicted as the optimal one achieves within the pool of 40 decompositions. The tree decomposition which led to the fastest solving time is ranked first. Hence, the closer the predicted rank is to 1, the better. One can expect a runtime improvement whenever the predicted rank is less than the median rank, which is for our pool of 40 tree decompositions between 20 and 21. In this context, it is important to mention that although rank 1 is not achieved one can still significantly improve the performance if the selected tree decomposition is ranked better than the "average" one.

Evaluation Process Indeed, the strict separation of the input data set into training and evaluation set makes the experiments prone to potential bias. To overcome this issue, we use random sub-sampling with 10 splits throughout our whole experimental evaluation. This approach constitutes a randomized adaption of the well-established technique of 10-fold cross-validation. The complete experimental setup for a problem therefore consists of 2800 benchmark runs based on different tree decompositions (70 problem instances with 40 tree decompositions for each of the instances).

For each of the ten iterations we select randomly 20 problem instances (leading to 800 tree decompositions) for the training set and the remainder of the pool is put into the evaluation set. The analysis then proceeds as described in the paragraph dedicated to the evaluation set. This process is repeated ten times to rule out bias as good as possible. By doing so, we obtain for each problem and model a total of 500 measurements from which we can draw precise conclusions about the runtime improvement achieved by using the tree decomposition predicted as the optimal one and the same holds also for conclusions about the predicted rank.

3.2.2 Experiments on Random Instances

Subsequently we provide a thorough investigation of experiments on random instances to show the potential of our approach. For every problem domain and each of the sixteen machine learning algorithms in our experimental setup we will present the predicted rank and the runtime improvement via box-plots. We also give aggregated performance measurements based on all computed models to underline the advantages our approach of selecting the optimal decomposition from a pool provides compared to the standard way of computing only one decomposition for a given problem instance.

Minimum Dominating Set

The results we obtained for this problem domain in our experiments are summarized in Figure 3.3. Before we go into the details of the figure, we first want to introduce its structure as it is crucial for interpreting the expected performance gain and therefore it will follow us throughout the remainder of this chapter.

In the header of the figure the problem name as well as information about the minimum and maximum runtime variation for the given instances are provided. These two ranges refer to the span between minimum and maximum solving time for a given problem instance when considering all random seeds which were used. In the case at hand we observe that the instance of Minimum Dominating Set with minimal runtime variation needed a solving time between 2.0 and 5.2 seconds (leading to a variation of 3.2 seconds) and that the instance with maximum variation requires solving times between 41.1 and 2877.6 seconds. The second pair of values, indicating a runtime variation of more than half an hour for the very same instance, impressively illustrates that there are huge differences in terms of runtime and so there is a significant potential for improvements which we try to exploit.

Minimum Dominating Set			
Minimum Runtime Variation:		2.0 s – 5.2 s	
Maximum Runtime Variation:		41.1 s – 2877.6 s	
Minimum Improvement:	7.39 %	Average Improvement:	21.80 %
Maximum Improvement:	31.15 %	Median Improvement:	24.25 %
Statistical Significance:		≥ 99.95 %	

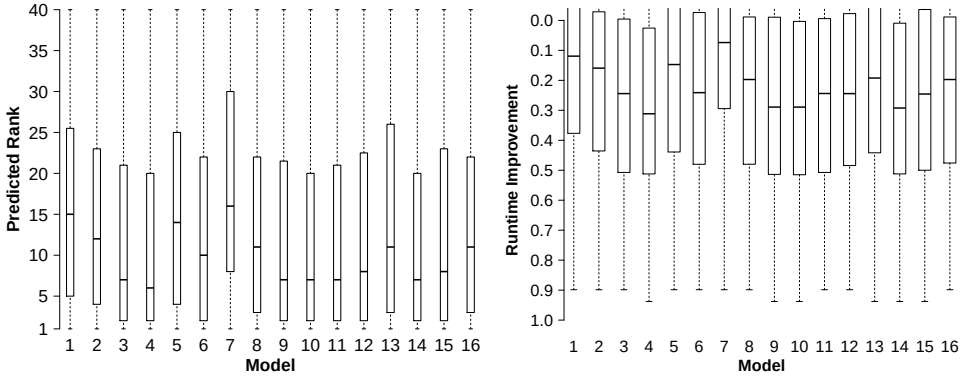


Figure 3.3: Performance Characteristics for Minimum Dominating Set

The aggregated measurements for the performance improvements achieved by using our approach are given in the subsequent rows of the header. The values are computed based on the median improvement obtained by using each Model 1–16. Furthermore, in the last row of the header we provide the results of our analysis for statistical significance. The value gives the probability that our approach leads to an improvement and is computed by taking the median significance of the one-sided t-test with the null-hypothesis that the observed performance improvement for a given model is 0. In other words, this last value in the header gives the probability that the average model, i.e., the hypothetical model ranked at the median position 8.5 among the 16 models, will indeed lead to a statistically significant performance improvement.

After this short introduction, let's have a look at the concrete values for the problem at hand. The figure headers for Minimum Dominating Set tell us that the improvement for any of the sixteen models is between 7.39% and 31.15% for *D-FLAT* while both median and average improvement are relatively close to the maximum. Please note that this is the net runtime improvement we would achieve in practice, hence we immediately see that the approach indeed pays off and that we can easily save a large portion of the total runtime. The very high statistical significance of not less than 99.95 percent – quite close to absolute certainty – for our benchmark setup finally tells us that the results are not a lucky strike and that we can also expect performance improvements in future experiments, at least in the same problem domain.

The two box-plots in Figure 3.3 are constructed on the basis of the 500 evaluations (50 instances with 10 iterations each) for each computed Model 1–16 (see Section 3.2.1).

On the left-hand side the predicted rank is illustrated and on the right-hand side we provide the box-plot of the distribution of the runtime improvement. Due to the fact that box-plots show the statistical distribution of values we gain even more insights into the capabilities of our proposed approach: By looking at the quartiles and outliers we can directly reason about the potential of our approach depending on the actual problem instances. To allow for a uniform presentation and because models leading to performance deteriorations would probably never be selected in practice, the box-plot for the performance improvement only shows the interesting range between 0 and 1.

3-Colorability

We will now present our experiments on 3-Colorability, depicted in Figure 3.4. Compared to the first problem we investigated in this chapter, this one is less "complex" because one does not have to keep track of additional, global information like the size of the dominating set in order to minimize it.

3-Colorability			
Minimum Runtime Variation:		1.7 s – 3.1 s	
Maximum Runtime Variation:		3.2 s – 1351.7 s	
Minimum Improvement:	16.90 %	Average Improvement:	29.74 %
Maximum Improvement:	36.67 %	Median Improvement:	30.99 %
Statistical Significance:		$\geq$ 99.95 %	

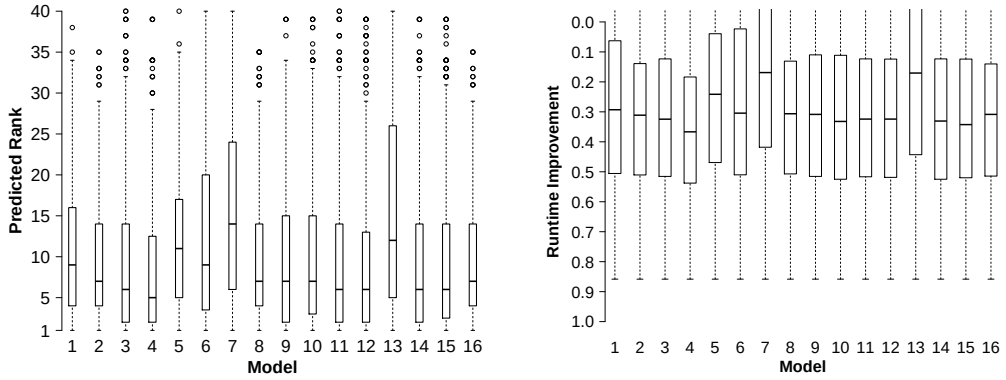


Figure 3.4: Performance Characteristics for 3-Colorability

For solving the problem at hand by means of dynamic programming on tree decompositions it is sufficient to look at each vertex in the input graph separately and simply check for each introduced neighbor if it is assigned the same color as the vertex under focus. Hence, there is almost no propagation of information needed, except for keeping track of the vertex color within the current tree decomposition node. Therefore, we could expect that machine learning for this second problem is somewhat easier than for Minimum Dominating Set and that the performance improvement is higher. Indeed these assumptions are confirmed in this case when we compare the figures for the two problems. Again, we observe a remarkable difference between the minimum and maximum solving

time for the very same problem instance, especially when we look at the maximum runtime variation depicted in the header of Figure 3.4. In our experiments with 3-Colorability, selecting a "good" tree decomposition can make the difference between solving an instance within seconds or having to wait more than twenty minutes.

Apart from this small side remark, we have for this problem domain the situation that each of the computed models has a good selection quality (compared to selecting a decomposition randomly) in most of the cases, as we can see in both figures for the problem at hand. An interesting fact visualized in the figures is that many models select in average a rank less than 10 out of 40 available tree decompositions for a problem instance while Models 7 and 13 (PLSClassifier and Bagging) are still performing good but significantly worse than the others.

Perfect Dominating Set

An extension to the problem of finding minimum dominating sets in a given input graph is the problem of finding minimum perfect dominating sets in a graph. The only difference between the two problems is the fact that in the latter, a dominated vertex must have exactly one dominator. This allows for much fewer solutions and so we expect a higher impact of the tree decomposition features on the solving time and therefore a better predicted rank than in the case of Minimum Dominating Set.

Perfect Dominating Set			
Minimum Runtime Variation:		2.3 s – 4.3 s	
Maximum Runtime Variation:		47.3 s – 6470.8 s	
Minimum Improvement:	46.22 %	Average Improvement:	58.91 %
Maximum Improvement:	64.72 %	Median Improvement:	60.69 %
Statistical Significance:		$\geq$ 99.95 %	

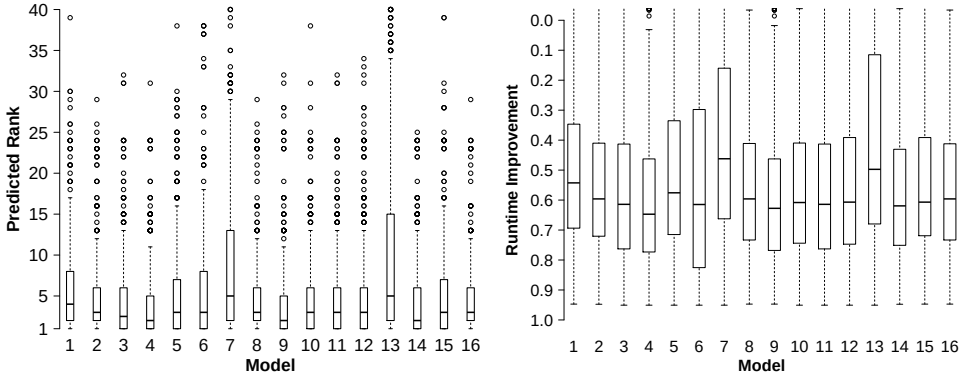


Figure 3.5: Performance Characteristics for Perfect Dominating Set

Figure 3.5 shows that the predicted rank is almost perfect for most of the models and also the runtime is cut in half in almost any of the investigated cases. Interestingly, most of the models predict rank 5 or better for the majority of the input instances. Again, we

observe that Models 7 and 13 (PLSClassifier and Bagging) show a worse outcome than the remaining models, but they still lead to an optimized runtime behavior.

Also in this case, the extremely diverging runtimes become apparent when interpreting the minimum and maximum runtime depicted in Figure 3.5. Solving a problem instance in less than a minute or waiting for the same result more than 1.5 hours can make a huge difference and also for the easiest instances the runtime needed to solve the respective instance is approximately cut in half.

Connected Vertex Cover

The next problem we want to focus on is Connected Vertex Cover. In practical situations, the connectedness of a solution is often a crucial requirement and so it is important to show that our proposed approach also works in these scenarios. The demand for connectedness makes the runtime prediction even harder because before solving the problem at hand there is no chance to maintain the solution's property of connectedness and to keep track of it by looking solely at tree decomposition features.

Connected Vertex Cover			
Minimum Runtime Variation:		4.0 s – 10.0 s	
Maximum Runtime Variation:		111.7 s – 2417.6 s	
Minimum Improvement:	22.38 %	Average Improvement:	32.91 %
Maximum Improvement:	42.78 %	Median Improvement:	34.49 %
Statistical Significance:		$\geq$ 99.95 %	

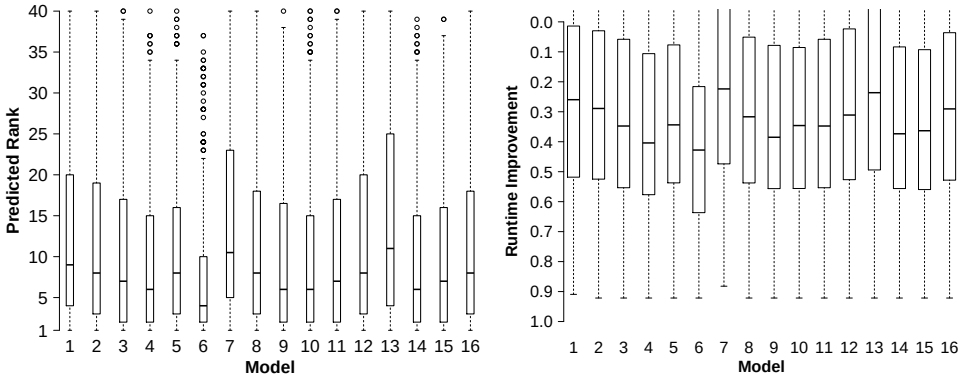


Figure 3.6: Performance Characteristics for Connected Vertex Cover

Figure 3.6 shows that also scenarios of this kind can be handled by our approach. Although the prediction is less accurate than in the case of Perfect Dominating Set – a fact that was expected, as mentioned above – we save one third of the overall runtime in the median case. Even the Models 7 and 13 (PLSClassifier and Bagging) which again are performing worst allow us to save a significant portion of the runtime in most of the cases. This time, we also have with model number 6 (PaceRegression) a dedicated

"winner" of the comparison as it is able to predict a rank between 1 and 10 in 75% of the cases and it selects rank 4 out of 40 in the majority of the cases.

Steiner Tree

The final problem domain we investigate in this chapter is the problem of Steiner Tree. Given an undirected graph with positive edge weights and a subset of the graph's vertices – the so-called *terminals* – the goal is to determine a minimum-weight, cycle-free subgraph of the input graph which connects all terminals.

We will have a look at the performance characteristics on real-world instances in the subsequent section. At this point, we first want to analyze the impact of our approach based on randomly generated instances. We fix the number of terminals for each of the instances to ten randomly chosen ones and we use the same terminal vertices in each tree decomposition generated for an instance.

Steiner Tree - 10 Terminals			
Minimum Runtime Variation:		1.8 s – 8.1 s	
Maximum Runtime Variation:		191.3 s – 10618.6 s	
Minimum Improvement:	31.56 %	Average Improvement:	59.05 %
Maximum Improvement:	66.33 %	Median Improvement:	62.02 %
Statistical Significance:		$\geq$ 99.95 %	

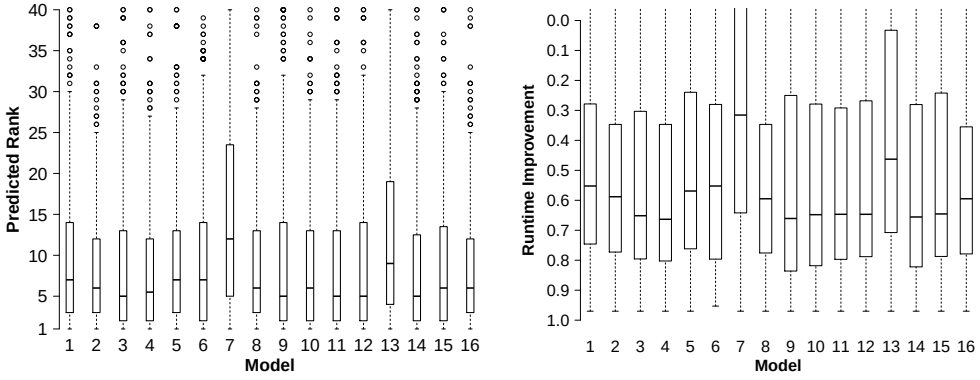


Figure 3.7: Performance Characteristics for Steiner Tree

The predicted rank and the runtime savings achieved during our experiments indicate that our approach works well also for "hidden" information like the actual terminals which are in no way distinguished from the other vertices in the generated tree decompositions. Hence, the tree decomposition features are completely unaffected by this information. Still, as shown in Figure 3.7, even the worst models – again Model 7 (PLSClassifier) holds the red lantern, while Model 13 (Bagging) is only slightly better – lead to runtime savings of about a third. The fact that even the models performing worst achieve significant savings of around a third of the total runtime becomes even more important when we look at the fact that these savings can be in the magnitude of hours when considering the maximum runtime variation depicted in Figure 3.7.

3.2.3 Experiments on Real-World Instances

Until now we only considered random instances in our experiments. With the goal of strengthening our findings, we additionally conducted a series of tests for Steiner Tree on real-world graphs.

The problem has many real-world applications like minimizing the effort (distance, time, etc.) to connect different terminals. While in some contexts the terminals are fixed for an input graph – one could for instance think of train stations – there are also situations where the set of terminal vertices changes frequently for the same input graph. One such example can be found in the area of predictive policing: In many cases the regions where crime is occurring more frequently are known but, depending on the daytime and events taking place in the city, these problematic regions can change rapidly. This is no problem as long as we can send officers to all the places, but often not enough personnel is available to do so. Therefore, there is a tendency to split the available officers into groups. One of the crucial problems to prepare for the case of an emergency is to maintain the ability to combine the forces with the least possible effort and this is exactly the point where the Steiner Tree problem comes into play.

The graphs chosen for these experiments are shown in Table 3.2 and represent the metro systems of some cities around the world. Compared to the random instances we have seen before, metro systems are much more structured. Often they have a more or less complex central region (covering the city center) and the remainder of the network is formed by simple paths (which are of width 1).

City	# Vertices	# Edges	Tree Decomposition Width			
			Min.	Max.	Avg.	Med.
Tokyo (JPN)	143	162	4	5	4.010	4
Osaka (JPN)*	145	160	4	5	4.334	4
Singapore (SGP)	101	114	4	5	4.487	4
Santiago (CHL)	127	138	4	6	4.218	4
Vienna (AUT)*	138	160	5	6	5.073	5

Table 3.2: Investigated Metro Systems (* ... Metro and Interurban Train)

Apart from the name of the selected cities, Table 3.2 also shows the size of the corresponding network in terms of vertices and edges. Furthermore, the last four columns contain the minimum, maximum, mean and median value for the width over the 2800 benchmark runs for each of the cities. Note that the metro networks contain a higher number of vertices than the random graphs investigated in the previous section. Therefore a different configuration of *D-FLAT* is used at this point in order to avoid problems due to main memory limitations. We will see in Section 3.2.5 that this modified configuration does no harm to the generality of our proposed approach.

Finally, note that Table 3.2 also highlights the fact that most tree decompositions for the cities are of the same width and hence, the huge runtime differences we observe on the different metro systems cannot be explained by considering the width only.

Steiner Tree - 10 Terminals			
Minimum Runtime Variation:		3.7 s – 8.4 s	
Maximum Runtime Variation:		35.9 s – 21528.0 s	
Minimum Improvement:	1.55 %	Average Improvement:	27.33 %
Maximum Improvement:	35.61 %	Median Improvement:	29.66 %
Statistical Significance:		≥ 99.95 %	

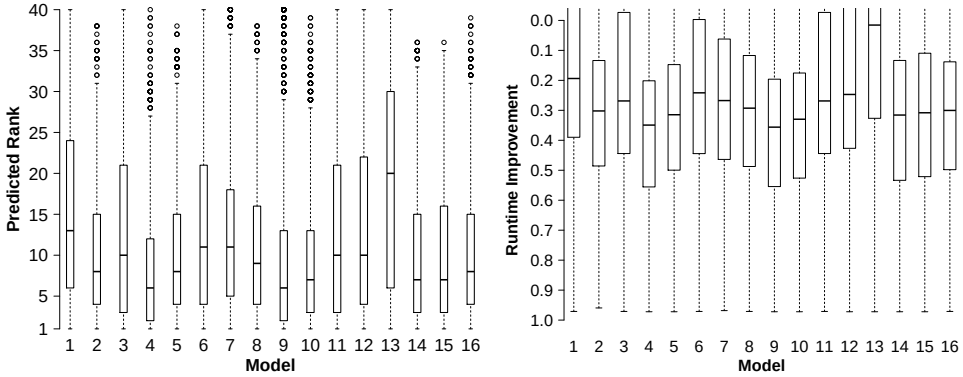


Figure 3.8: Performance Characteristics for Steiner Tree (Real-World)

Figure 3.8 shows the aggregated outcome for our experiments on the metro systems, hence each box-plot is constructed from 14000 benchmark runs. A separate discussion for each of the cities is given in [AMW16c]. In the figure we can see that each model leads to runtime savings and that the majority of the models helps us saving more than a quarter of the total runtime in average. For the metro system of Tokyo this saves us "only" a few seconds while in the case of Vienna we sometimes save hours using our approach.

3.2.4 Model Evaluation

After we presented the thorough investigation of our approach on random and real-world instances, we also want to present the results of our performance analysis separately for each model. Table 3.3 summarizes these outcomes. The table shows for each of the 16 models under investigation the median predicted rank over all evaluation runs for each of the problems. The column "ST (Real)" contains the median predicted rank over 2500 evaluations as we merge the results from the five cities under investigation. For all other problems, the cell content is computed by taking the median over 500 evaluations. The numbers in boldface highlight the best-performing models.

The last two rows (columns) then provide the mean and median of all preceding rows (columns). Note that the three cells in the bottom right corner are left empty because in these cases, the results will differ depending on whether one computes the median of means or the mean of medians. The same applies for the median of the medians, which also differs depending on whether we start with the column-wise or the row-wise median. What we can compute easily is the average predicted rank over the average model and

Model	Col	MDS	PDS	CVC	ST	ST (Real)	Avg.	Med.
1 (Gauss.-Proc.)	9	15	4	9	7	13	9.50	9.00
2 (IsotonicReg.)	7	12	3	8	6	8	7.33	7.50
3 (LeastMedSq)	6	7	2.5	7	**5**	10	6.25	6.50
4 (LinearReg.)	**5**	**6**	**2**	6	5.5	**6**	**5.08**	**5.75**
5 (ML-Perc.)	11	14	3	8	7	8	8.50	8.00
6 (PaceReg.)	9	10	3	**4**	7	11	7.33	8.00
7 (PLSClassifier)	14	16	5	10.5	12	11	11.42	11.50
8 (SMOreg)	7	11	3	8	6	9	7.33	7.50
9 (IBk)	7	7	**2**	6	**5**	**6**	5.50	6.00
10 (KStar)	7	7	3	6	6	7	6.00	6.50
11 (LWL)	6	7	3	7	**5**	10	6.33	6.50
12 (AdditiveReg.)	6	8	3	8	**5**	10	6.67	7.00
13 (Bagging)	12	11	5	11	9	20	11.33	11.00
14 (CVPSel.)	6	7	**2**	6	**5**	7	5.50	6.00
15 (M5Rules)	6	8	3	7	6	7	6.17	6.50
16 (M5P)	7	11	3	8	6	8	7.17	7.50
Average	7.81	9.81	3.09	7.47	6.41	9.44	7.34	—
Median	7.00	9.00	3.00	7.50	6.00	8.50	—	—

Table 3.3: Predicted Ranks (Median) for Computed Models

all problems and this value is shown in the highlighted cell in the bottom right corner.

We can see that in the average case we predict rank 7 out of 40, which is much better than the median rank of 20.5 and hence we can expect an important gain in terms of performance. Even the machine learning algorithms holding the red lantern, Models 7 and 13 (PLSClassifier and Bagging), predict well in most cases. In fact, the only case in our whole experimental evaluation where our approach does not lead to an improvement in the average case is Model 13 (Bagging) on the real-world Steiner Tree Problem. In all other cases we actually achieve quite impressive prediction results. Especially noteworthy is Model 4 (LinearRegression) which is able to select a Top-5 rank in the average case of our experiments.

3.2.5 Inter-Domain Evaluation

Until this point of the chapter it was the case that we investigated the applicability of our approach for each problem domain separately. For practical application scenarios it might be of importance (or at least of interest) to be able to adapt algorithms or to change the application domain without having to re-train the models one has already computed because this can be a time-consuming task.

Therefore we now want to have a deeper look at the inter-domain applicability of our approach. All the results are summarized in Table 3.4. The rows refer to the problem

that was used to generate the training data for the models and the columns stand for the evaluation dataset. The respective datasets are the same as for the domain-dependent experiments. The cells of the table then show the median value of the predicted rank over all 16 models. Cases in which we do not observe improvements are enclosed by brackets. The rightmost two columns and the last two rows illustrate the mean and median over all problem domains, analogous to Table 3.3. The only difference is the fact that this time we give two results: The number on the top of the cells is the respective outcome over all domains while the second number represents the outcome computed with the diagonal excluded. This means that the first number gives the overall performance over all domains while the second one is the performance we observe on average in our setting when we switch the problem domain.

	Col	MDS	PDS	CVC	ST	ST (Real)	Avg.	Med.
Col	7	11.75	12	11	19	11.5	12.04 13.05	11.63 11.75
MDS	12.5	9	7	8	14	9.5	10.00 10.20	9.25 9.50
PDS	(20)	10.75	3	6	6	12	9.63 10.95	8.38 10.75
CVC	16	8	5	7.5	8	11	9.25 9.60	8.00 8.00
ST	(23.75)	17.25	5	9.5	6	16	12.92 14.30	12.75 16.00
ST (Real)	16.5	15.5	10	13.5	16	8.5	13.33 14.30	14.25 15.5
Average	15.96 17.75	12.04 12.65	7.00 7.80	9.25 9.60	11.50 12.60	11.42 12.00	11.19 12.07	—
Median	16.25 16.50	11.25 11.75	6.00 7.00	8.75 9.50	11.00 14.00	11.25 11.50	—	—

Table 3.4: Predicted Ranks (Median) for Computed Models (Inter-Domain Evaluation)

We can see that in almost all cases (34 out of 36) we observe improved results and that there is only one problematic situation, namely the case where we use the dataset obtained from solving Steiner Tree on random instances to predict the outcomes for 3-Colorability. In this case we observed a slight deterioration of the predicted rank – on average, our models predicted rank 23.75 compared to the median rank 20.5. – compared to a random selection of the tree decomposition. As mentioned before, in all other cases we observed improvements which are statistically highly significant with a confidence level of over 99.95% and we have to keep in mind that these are the values for the average model, not for the best one. In the complete picture, summarizing all the positive impact of our approach, we have the fact that we were able to select rank 11 out of 40 on average (result shown in the highlighted cell in the bottom right corner of

the table) which gives us important performance improvements compared to a random selection of the tree decomposition.

3.2.6 Discussion of Experiments

The provided evaluation underlines that our machine learning approach shows great potential for improving the performance of dynamic programming algorithms. Because, in our experiments, the width of the tree decompositions is the same for almost all decompositions of a given instance, just minimizing the width of the tree decompositions is obviously not always sufficient and one needs a better way to select and/or to customize tree decompositions in order to improve the overall performance and especially the robustness of dynamic programming algorithms.

We can see that in general there is no "perfect" model which performs best in every case and that there exist differences between the problems. In our experimental evaluation it was the case that Model 4 (Linear Regression) performed best and that the Models 7 (PLSClassifier) and 13 (Bagging) showed the worst – but in many cases still relatively good – performance characteristics. We assume that the rather poor performance of PLSClassifier originates from an overly restrictive filter being used. In the case of Bagging, the underlying regression algorithm (the algorithm which is used in our experiments employs support-vector machines for regression) may be too strong and choosing a weaker base classifier may help to improve the prediction quality, as suggested in [Bre96].

Our experiments show that it is hard to predict the very best rank, but in many cases this is not needed. In general, every rank better than the median rank will increase the performance and the advantage grows with the runtime variance. Furthermore, in Section 3.2.5 we showed that one does not need to train models for each new problem and that one can achieve good results also by applying models trained in a completely different setting. This makes our approach even more applicable in real-world application scenarios as one does not necessarily have to re-train the model(s) when the problem domain changes or new constraints are added.

3.3 Discussion

In this chapter we studied the applicability of machine learning techniques to improve the runtime behavior of dynamic programming algorithms which rely on tree decompositions. To this end, we identified a variety of tree decomposition features, beside the width, that strongly influence the runtime behavior of DP algorithms. Machine learning models using those features for the selection of the optimal decomposition have been validated by means of an extensive experimental analysis including real-world instances. In our experiments, we considered five different problem domains and our approach showed a remarkable, positive effect on the performance with a high statistical significance. We thus conclude that turning the huge body of theoretical work on tree decompositions and dynamic programing into efficient systems highly depends on the quality of the

chosen tree decomposition and that advanced selection mechanisms for finding good decompositions are crucial.

The presented work, however, is only a first step of a larger research project. In a next step, we have to investigate whether the models we obtained will give further insights about those features of tree decompositions that are most influential in order to reduce the runtime of DP algorithms. Such insights then will be used to design and implement new heuristics for constructing tree decompositions that optimize the relevant features. Therefore, the ultimate goal of this research perspective is to achieve the potential speed-up we have observed in our experiments by directly obtaining tree decompositions of higher quality and thus *without* the initial training step.

Indeed, due to the fact that algorithms based on tree decompositions are an area of intensive research, there also arise performance improvements for specialized algorithms, as studied, e.g., by Fafianie et al. [FBN15], who showed that solving the Steiner Tree problem can be significantly accelerated by combining tree decompositions with methods from linear algebra. It may be an interesting task in future work to investigate the impact of tree decomposition selection also in this context. Furthermore, we expect that problem-specific features are a promising enhancement of our approach. In the chapter at hand, we only used features of the tree decomposition in order to establish the problem-independent, general applicability of our approach. In practical situations where efficiency is crucial, it may be worth trying to find some kind of problem-specific tuning. For instance, one could aim to develop a good approximation function that estimates the time of filling the dynamic programming tables, similar to the idea of the f-cost in [BF05], and then use this function to obtain a new feature.

CHAPTER 4

A Framework for (Customized) Tree Decompositions and Beyond

In the previous chapter we have seen that there is often more to consider than just the plain width of tree decompositions in order to be able to reliably predict their influence on the efficiency of given dynamic programming algorithms.

However, the approach presented in the previous chapter uses a post-processing phase based on machine learning to determine "good" tree decompositions and training the required models can be a time-consuming task. Therefore we see a strong need to offer a specialized decomposition framework that allows for directly constructing customized decompositions, i.e., decompositions which reflect certain preferences of the developer, in order to optimally fit to the dynamic programming algorithm in which they are used.

For this reason, in this chapter we present a free, open-source solution which supports a vast amount of graphs and different types of decompositions. Our framework can be easily extended as we provide programming interfaces for (almost) all classes and so one does not need to re-invent the wheel at any place. In detail, the design goals for our framework are as follows:

- Clean, easy-to-use interfaces
- Runtime and memory efficiency
- Utmost flexibility and extensibility
- Support for a variety of graph and decomposition types
- Support for a wide range of convenience features like various normalization strategies and automated modifications of decompositions directly in the context of the library in order to keep the code clean and structured

The software library is called *htd*. We consider *htd* as a potential starting point for researchers to contribute their algorithms in order to provide a new framework for all different types of (customized) graph decompositions.

Currently, *htd* is used successfully in several projects, like the *D-FLAT* framework for dynamic programming on tree decompositions (see Section 2.3), *dynASP* [MPRW10, FHMW16], an answer-set programming solver based on dynamic programming on tree decompositions, or *dynQBF* [CW16b], a solver for quantified boolean formulae based on dynamic programming and binary decision diagrams.

In the remainder of this chapter, we provide a detailed description of the features of *htd*, shed some light at crucial algorithm decisions, illustrate its usage in some example scenarios, and we also give an experimental evaluation comparing the tree decomposition heuristics currently offered by the *htd* framework to other state-of-the-art implementations of tree decomposition algorithms.

To find out how *htd* compares to other approaches for computing tree decompositions, *htd* is one of the participants of the "First Parameterized Algorithms and Computational Experiments Challenge" (PACE 2016)[1] and it was ranked at the third place in the sequential heuristics track. The results of *htd* are very close to those of the heuristic approaches ranked at the first two places. This underlines not only that *htd* is rich in features helping to make the development of dynamic programming algorithms more comfortable, but also that it is very competitive when compared to other approaches.

4.1 A General Framework for Custom Decompositions

In this section we want to have an in-depth look at some important properties of the new framework. During our work with *D-FLAT* [ABC+14b], a framework for easy prototyping of dynamic programming on tree decompositions, we faced the problem that existing implementations for graph decomposition algorithms often only minimize the width. That is, they deliver a tree decomposition without possibility to transparently customize the result. Hence, post-processing outside the decomposition library is needed in order to obtain a decomposition which reflects certain preferences of a developer and which fits well to given dynamic programming algorithms (e.g., by following the approach presented in Chapter 3). Furthermore, existing algorithms are often hard to adapt because, at design time, certain capabilities and mechanisms (like assigning arbitrary labels to the resulting decompositions automatically) were not considered and so extensions of functionality often require rewriting huge portions of the old code.

To circumvent all these problems, the proposed library, called *htd*, is free, open-source software and it is available at `http://dbai.tuwien.ac.at/research/project/decodyn/htd/`. The software was developed with the goal to serve the needs of virtually any algorithm related to graph decompositions. In the following we will highlight the library's main characteristics.

[1] See `https://pacechallenge.wordpress.com/track-a-treewidth/` for more information.

4.1.1 Support for a Variety of Input Graph Types

Since the library shall be able to decompose any given graph type, *htd* supports by default a variety of them to fit like a glove to the actual application domain where our library is applied. Indeed, all input graphs can be stored in a data structure which is able to handle multi-hypergraphs, i.e., hypergraphs with potentially duplicated hyperedges. E.g., we could also store a directed graph in a data structure for multi-hypergraphs, but multi-hypergraphs are, for instance, not aware of the concept of incoming or outgoing neighbors and also reachability is defined differently for hypergraphs than in the case of directed graphs. In order to enhance functionality, to enforce semantic coherence and to shift programming effort from the developer using the library to our framework, *htd* offers separate data types and programming interfaces for storing (multi-)hypergraphs, directed and undirected (multi-)graphs, trees and paths.

For each graph type, *htd* also offers an implementation which is able to deal with custom names so that, instead of working with plain vertex and edge identifiers in terms of numeric integers, one can additionally address a specific vertex or edge by its name. To fit the needs of dynamic programming algorithms in a general and convenient way, one can use any equality-comparable data type which provides functionality for returning its hash code (like character strings) as an alias for the name of a vertex or an edge.

Furthermore, *htd* allows to add custom labels of any data type to the vertices as well as the (hyper-)edges of a given graph by providing appropriate wrappers for each graph type. These labels can, for instance, be used to store polarities for the endpoints of an hyperedge in case that hyperedges represent clauses of the SAT problem (see Figure 4.3 for an example).

One big advantage of having a built-in support for labeled graph types is the fact that with this functionality one can keep the productive code clean and simple because one only needs a single graph representation in memory and no conversion or mapping of the graph structure between internal library code and the developed algorithm is necessary.

4.1.2 Support for a Variety of Decomposition Types

Clearly, from a graph decomposition library we expect the ability to decompose graphs. To serve this purpose, *htd* offers several decomposition methods by default and a wide range of interfaces allows to extend *htd*'s functionality easily and without even having to re-compile the library. Each decomposition algorithm in the context of *htd* takes an (potentially disconnected) input (hyper-)graph $\mathcal{G}$ in any supported representation and constructs a decomposition of the requested type. The library distinguishes between four types of decomposition algorithms:

- **Graph Decomposition Algorithms**

  ... return a new, labeled multi-hypergraph GD where the bag of each vertex in GD is a subset of the vertices in $\mathcal{G}$. This is the most general decomposition type currently supported by *htd*.

- **Tree Decomposition Algorithms**

  ... return a new, labeled tree TD where the bag of each vertex in TD is a subset of the vertices in $\mathcal{G}$ such that all criteria for tree decompositions are satisfied. In the first step, the default implementation of a tree decomposition algorithm in *htd* uses Bucket Elimination [Dec99, DGG$^+$08] based on a generated eliminating ordering to obtain a tree decomposition for each connected component of the input graph. Afterwards, it connects the trees in the forest to a single tree by adding additional edges where appropriate. The vertex ordering, required for Bucket Elimination, is obtained via the library's default ordering algorithm which the developer can select before the decomposition algorithm is invoked.[2]

- **Path Decomposition Algorithms**

  ... return a new path decomposition, i.e., a tree decomposition without join nodes, where the bag of each node is a subset of the vertices in $\mathcal{G}$ such that all criteria for tree decompositions are satisfied. The default implementation constructs a tree decomposition of the input graph and then manipulates it by rearranging the join node's children in order to obtain a path structure.

- **Hypertree Decomposition Algorithms**

  ... return a new, labeled tree TD where the bag of each vertex in TD is a subset of the vertices in $\mathcal{G}$. Additionally, each vertex of TD has assigned a second label, consisting of a subset of hyperedges of the input graph such that the corresponding bag content is a subset of the set union of the endpoints of these hyperedges. In other words, *htd*'s default implementation computes a generalized hypertree decomposition [GMS09].

  The current implementation starts by first generating a tree decomposition and then we solve the Set Cover problem for each of its bags. That is, we compute for each bag the minimum-cardinality set of all hyperedges such that each vertex in the bag has at least one hyperedge in which it is contained.

We can see that basically every algorithm is constructed in a very modular way, i.e., we only require the input graph which shall be decomposed and afterwards everything is up to the concrete implementations. This leads to light-weight interfaces and high flexibility for both developers "just" using the library and those who want to contribute to the library. For instance, although the default implementations of the algorithms rely on the simple Bucket Elimination procedure, there is absolutely no need for a developer to use Bucket Elimination or vertex elimination orderings at all.

Finally, before we have a deeper look at flexibility and extensibility of *htd*, we should have a glance at an additional feature concerning decomposition algorithms which can be very

[2] Apart from employing a custom algorithm given by the developer, *htd* currently provides default implementations for Min-Fill, Minimum Degree and Maximum Cardinality Search vertex elimination orderings. If the developer does not specify the ordering algorithm to use, the default setting is Min-Fill.

helpful in practice: For reduced post-processing effort on the developer-side, *htd* offers the concept of manipulations which can be applied to a computed decomposition. The term "manipulation" in the context of *htd* refers to operations which manipulate the structure of the decomposition, like making a tree decomposition nice, or adding/removing/changing certain labels.

Manipulations can always be applied to a given decomposition but one can also specify the list of desired manipulations in the call of the decomposition algorithm. In the latter case, the computed decomposition will be returned with the desired operations automatically applied. This helps to keep the code at developer-side clean because one does not have to do any post-processing in order to get the manipulations applied.

4.1.3 Automated Optimization of Decompositions

One of the main novelties of *htd* compared to existing implementations of decomposition algorithms is the support for automated optimization.[3] In order to support the special needs of certain dynamic programming algorithms, like minimizing the number of join nodes, or even to allow complex optimizations, like the prioritization of selected vertices with respect to their average position in the decomposition, *htd* supports two optimization strategies.

The first strategy implemented by the library is an iterative approach which computes a (user-definable) number of decompositions and finally returns the decomposition for which the fitness evaluation (the result of the provided fitness function), is maximal. This strategy allows to mimic the approach which proved successfully in [ADMW15, AMW16c, AMW17b], namely computing a pool of tree decompositions and then selecting the optimal one among them via machine learning, by only writing a few lines of code.

The second strategy relies on the fact that one can select any vertex of a tree as its root and the outcome of this reallocation will still be a tree. Based on this observation, the second optimization strategy allows to automatically select the node as root of the tree decomposition for which the fitness evaluation for the tree decomposition rooted at the respective node is maximal. In *htd*, one can select from different (custom) criteria which aim at narrowing down the subset of vertices of a decomposition which shall be considered as new root node. This is important especially for large graphs as exhaustively trying out all possible choices might be expensive.

Note that the two approaches can, of course, be combined in order to improve the result of the optimization step. Additionally, one can assign also priorities to each level of a fitness evaluation in case that multi-criteria optimization is needed.

[3]Note that in the context of *htd*, the term "optimization" is used to refer to all operations which improve certain properties of a decomposition. Via optimization the decomposition returned by an algorithm adheres to certain preferences, but in general the implemented algorithms do not guarantee global optimality of the resulting decomposition with respect to the quality criterion used.

4.1.4 High Level of Flexibility and Extensibility

One of the main limitations of many software libraries is the fact that they are often developed as a by-product of some application and so the complete functionality is, in many cases, tailored towards a specialized application domain and extensions or adaptations are hard to implement. *htd* is different in that sense as it is developed independently from a concrete application domain[4]. Moreover, it is designed to be easily extensible and highly flexible. Currently, the library provides around 80 interfaces for almost all parts of the library, allowing to easily replace, improve and extend functionality. Furthermore, *htd* provides explicit factory classes for most interfaces so that one can set new default implementations without even having to re-compile the library.

4.1.5 Working with the Library

After sketching some important characteristics of *htd*, in this section we now want to give an overview of the general workflow of how to compute decompositions via *htd* and we also provide an example of an application scenario to illustrate how the library can be used in practice.

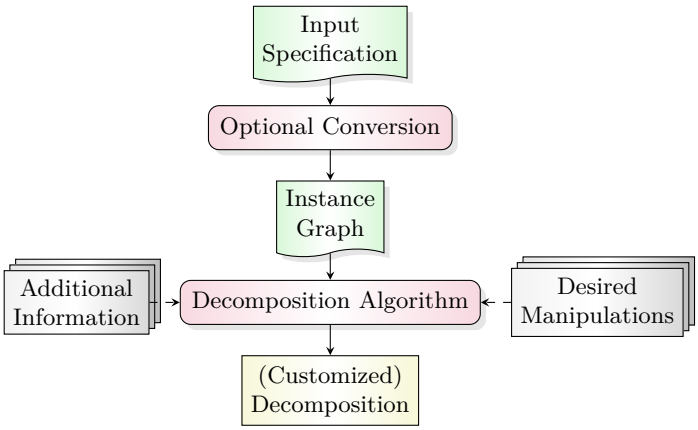


Figure 4.1: Workflow for Computing Customized Decompositions using *htd*

Figure 4.1 depicts the concept behind *htd*. At first, one clearly needs to parse the input and maybe do some conversion or preprocessing in order to obtain a graph representation of the input instance which we can directly feed into the decomposition algorithm. The decomposition algorithm may be either one of the built-ins of *htd* or a custom one specified by the developer. Optionally, the developer can provide additional information to the decomposition algorithm, like a custom vertex ordering as needed by Bucket Elimination. Further, it is possible to request different manipulations of the resulting decomposition, like computing additional labels or making a tree decomposition nice.

[4]Note that *htd* also provides a command-line application, named *htd_main* which is a small, light-weight front-end for the library that allows to trigger the main functions of *htd*. The front-end is fully configurable in order to easily investigate the effects of modifications to *htd*.

Note that the library supports a developer in any of the depicted steps, that is, it defines interfaces allowing the parsing of input files and it provides implementations for various graph types, decomposition algorithms and manipulation operations. Furthermore, the large collection of algorithms in *htd* contains lots of convenience functions.

For instance, probably a very helpful "bonus" functionality of our library is the following: All built-in decomposition algorithms of *htd* automatically provide for each bag the set of induced edges, i.e., the set of all edges which form a subset of the bag content. This means that in a dynamic programming algorithm there is no need anymore to filter the full set of edges of the input graph for each bag in order to find out which of them are affected by the respective bag. This can be a very time-consuming task for large graphs. *htd* provides this important information (almost) for free, significantly accelerating the given applications.

```
1  // Create a new management instance (with ID 1) for the current thread.
2  htd::LibraryInstance * manager = htd::createManagementInstance(1);
3  // Import some graph in the format of the PACE challenge from stdin.
4  htd_main::GrFormatImporter importer(manager);
5  htd::IMultiGraph * graph = importer.import(std::cin);
6  // Get default decomposition algorithm.
7  htd::ITreeDecompositionAlgorithm * algorithm =
8      manager->treeDecompositionAlgorithmFactory().createInstance();
9  // Set desired manipulations for algorithm.
10 algorithm.addManipulationOperation
11     (new htd::NormalizationOperation(manager));
12 // Decompose the provided graph.
13 htd::ITreeDecomposition * td =
14     algorithm->computeDecomposition(*graph);
15 // Output the width of the obtained tree decomposition.
16 std::cout << "Width: " << (td->maximumBagSize() - 1) << std::endl;
```

Listing 4.1: Example Source Code (C++) How to Use *htd* in Practice

To underline how easy *htd* can be applied in practice, in Listing 4.1 we give a short working example in terms of the only six lines long C++ source code sufficient to compute a nice tree decomposition of a given input graph in the format of the PACE challenge 2016[5]. The first line initializes a central management instance of the *htd* library which can be used to set and access the default implementations for all algorithms. The next two lines of the provided source code take care of importing the input graph. The fourth line then gets the default tree decomposition algorithm of *htd*. All what remains is to set the desired manipulation operation for normalized tree decompositions and to decompose the graph. The last line of our example then outputs the width of the decomposition, but one may also proceed with a dynamic programming algorithm.

[5]The specification of the graph format is provided at `https://pacechallenge.wordpress.com/track-a-treewidth/`

Note that the above source code is more or less a minimal working example, but one is free to implement any algorithm one can think of and it will work with the library as long as it implements the interfaces of *htd* properly. In the following list we give some examples for important interfaces of *htd*:

- `htd::IMultiHypergraph`

  All (custom) graph classes must implement the interface for hypergraphs with potentially duplicated edges. It provides the functionality which is common to all graph types, like accessing its vertices and edges or, for instance, to get the neighbors of a vertex.

- `htd::ITreeDecomposition`

  A tree decomposition in *htd* is a special case of a `htd::IMultiHypergraph` in which each vertex has a bag label and where the underlying graph is a tree. Like mentioned before, apart from the basic functionality one would expect from a tree decomposition data structure, the `htd::ITreeDecomposition` interface additionally defines a function to retrieve the hyperedges of the input graph which are induced by a bag.

- `htd::ITreeDecompositionAlgorithm`

  This interface must be implemented by all tree decomposition algorithms in the context of *htd*. It defines functionality to compute a `htd::ITreeDecomposition` of some `htd::IMultiHypergraph`. Apart from decomposing a given input graph, *htd*'s tree decomposition algorithms automatically apply provided manipulation operations.

- `htd::IDecompositionManipulationOperation`

  As mentioned in Section 4.1.2, manipulation operations are an important part of the library and cover all modifications which are applied to a decomposition's structure or its labels. For each type of decomposition, *htd* provides a separate interface in order to ensure that only compatible operations are performed during the computation of a decomposition.

4.2 Developer Documentation

htd is a relatively small piece of software for efficiently computing decomposition of large graphs and hypergraphs. The library is developed with the goal to optimally fit the needs of a wide range of dynamic programming algorithms. Although its code base is small, the library is able to efficiently decompose graphs containing millions of vertices and it offers various interfaces to provide the developers of dynamic programming algorithms exactly with those features they need without having to pay the price for functionality they don't need.

To employ *htd* for one's purposes and to fully exploit its potential it is important to know about its interface structure, the functionality each of the interfaces offers and how the underlying algorithms interact. In this section we therefore want to give a detailed introduction to *htd*'s application programming interface.

4.2.1 Introduction

Before we dig into the details of the *htd* software library, in this section we want to give an introduction to its macrostructure in order to make the remainder of the documentation easier to follow. *htd* is structured roughly as follows:

- Input graphs
- Graph decompositions
- Decomposition algorithms
- Manipulation operations
- Normalization operations
- Utility functions

Before using any of these features, the first thing to start with when developing a new application based on *htd* is creating a so-called *library instance* of *htd*. A library instance acts as a central point of management allowing to configure the default settings for any algorithm within the library. Therefore we will use the term "manager" as a synonym for the term "library instance" for the remainder of this work.

An example of how to properly initialize the manager is given in Listing 4.2. Note that one can use more than one manager per application, for example, in situations where it is desired to have different configurations per thread. After creating the new manager, it may be used by algorithms to incorporate the developer's preferences. This is ensured by the fact that the manager contains a collection of factory classes (one for each interface available in *htd*) and so each algorithm can use exactly the desired settings.

```
1 // Create a new management instance (with ID 1) for the current thread.
2 htd::LibraryInstance * manager = htd::createManagementInstance(1);
```

Listing 4.2: Example Source Code (C++) How to Initialize a Library Instance of *htd*

For a minimal working example, the code shown in Listing 4.2 suffices due to the fact that *htd* provides efficient default implementations for each of the interface classes. Nevertheless, one may decide to use a different algorithm provided by *htd* or one can even use a custom implementation. A toy example how to set and retrieve the default implementation of the graph class is given in Listing 4.3.

```
1 // Create a new management instance (with ID 1) for the current thread.
2 htd::LibraryInstance * manager = htd::createManagementInstance(1);
3 // Change the default implementation for graphs.
4 manager->graphFactory().setConstructionTemplate(new MyFancyGraphClass());
5 // Get a new instance of the graph class.
6 htd::IMutableGraph * g = manager->graphFactory().createInstance();
```

Listing 4.3: Example Source Code (C++) How to Change the Default Graph Type

Note that, in *htd*, factory classes always take control over the memory region pointed to by the argument of the function `setConstructionTemplate` in order to avoid copying the instance. Therefore, one must not free the pointed-to memory manually because this is done automatically by the factory class.

In the remainder of this chapter we present the functionality for each group of data structures and each algorithm category. All subsequent sections will rely on the fact that a properly initialized and configured library instance named "manager" already exists.

4.2.2 Graph Types

htd distinguishes five different graph types, namely hypergraphs (graphs with hyperedges), (undirected) graphs, directed graphs, trees and paths. According to this distinction, *htd* provides the following interface classes:

- `htd::IHypergraph`, `htd::IMultiHypergraph`

  Hypergraphs are the most general graph type in *htd* as they allow using hyperedges, i.e., edges with an arbitrary number of endpoints. Self-loops, induced by hyperedges containing the same vertex multiple times, are allowed.

  Inheritance:

  Each `htd::IHypergraph` is a `htd::IMultiHypergraph`.

- `htd::IGraph`, `htd::IMultiGraph`

  Graphs in the context of *htd* are hypergraphs where each hyperedge has exactly two endpoints. Again, self-loops are allowed.

  Inheritance:

  Each `htd::IGraph` is a `htd::IHypergraph`.
  Each `htd::IMultiGraph` is a `htd::IMultiHypergraph`.

- `htd::IDirectedGraph`, `htd::IDirectedMultiGraph`

  Directed graphs in the context of *htd* are graphs where the order of endpoints matters. In addition to the functionality of graphs, the corresponding interface classes for directed graphs allow to easily retrieve the incoming and outgoing neighbors of a vertex. Self-loops are allowed.

Inheritance:

Each `htd::IDirectedGraph` is a `htd::IGraph`.
Each `htd::IDirectedMultiGraph` is a `htd::IMultiGraph`.

- `htd::ITree`

  This interface is implemented by all tree classes and it allows to access the tree.

  Inheritance:

  Each `htd::ITree` is a `htd::IGraph`.

- `htd::IPath`

  This interface is implemented by all path classes and it allows to access the path.

  Inheritance:

  Each `htd::IPath` is a `htd::ITree`.

The graph classes above only provide read-only methods in order to maintain a proper inheritance hierarchy. To illustrate the problem, think about the common statement that any tree is a graph. On the one hand, when looking at the aforementioned sentence from the read-only perspective, there is no doubt that it is true. On the other hand, when looking at it from the write-perspective, i.e. when we want to modify the graph, the statement is no longer valid as we cannot add arbitrary edges to a tree without potentially violating the acyclicity requirement. To circumvent this problem, each read-only graph class has its mutable counterpart which is denoted by adding the character sequence "Mutable" between the capital letter 'I' and the remainder of the graph class name. For instance, the mutable interface for the interface class `htd::IHypergraph` is `htd::IMutableHypergraph`.

The read-only graph classes support the aforementioned inheritance hierarchy (each path is a tree, each tree is a graph ...) while the mutable graph classes, which extend the immutable ones by adding the specialized functionality to modify the underlying graph, are not subject to inheritance.

All graph types which contain the character sequence "Multi" in their names, that are `htd::IMultiHypergraph`, `htd::IMultiGraph` and `htd::IDirectedMultiGraph`, allow edge duplicates while all other graph types assume that edges with exactly the same endpoints (in identical order) refer to the very same edge instance.

Labeled Graph Types

Furthermore, for each graph type there is also a labeled counterpart which allows to assign custom labels to the vertices and edges of the graph. One example for a labeled graph type is for instance the interface class `htd::ILabeledTree` (with its mutable version `htd::IMutableLabeledTree`). Using these two interfaces one can, in addition to the basic functionality provided by the tree classes, assign arbitrary, customizable

labels to the vertices and edges of the tree. The labeled versions of the other graph types provide analogous functionality for the respective basic type.

Named Graph Types

While standard graph types implemented in *htd* take integers as identifiers for vertices and edges in order to save resources, input graphs often use different data types for referencing vertices and edges. To provide the developer with enough flexibility to use arbitrary data types as identifiers, *htd* offers for each graph type a template class which automatically takes care of the efficient one-to-one mapping between the identifier used in the context of the input graph and the integer identifiers used by *htd*. We will subsequently refer to those template wrappers for graph by the term "named graph types".

One example instantiation of such a C++ template class representing a named graph type is the class `htd::NamedGraph<std::string,std::string>` which wraps an instance of `htd::MutableLabeledGraph` in such a way that instead of using an automatically assigned integer to reference vertices and edges, one can now use a (unique) `std::string`. Similarly, the class `htd::NamedGraph<int, std::string>` uses integers to address vertices and strings to identify edges. Generally, one can use any data type as identifier which provides a hash code and which is equality-comparable. The named versions of the other graph types work in an identical manner for the respective basic type.

4.2.3 Decomposition Types

Initially, the focus of *htd* was the efficient computation of tree decompositions only. Nevertheless, in order to perfectly fit the special needs of developers of dynamic programming algorithms, *htd* at the current stage of development offers support for four basic types of decompositions:

- `htd::IGraphDecomposition`

  Graph decompositions are the most general type of decompositions in the context of *htd*. The interface offers all possibilities of `htd::IMultiHypergraph` and extends it by providing functionality to add a bag information as well as assigning vertex and edge labels. The bag information in the context of *htd* is always a sorted vector of vertices from the original graph from which the decomposition was computed. Graph decompositions allow for duplicate edges as well as for disconnected graphs.

  Inheritance:

  Each `htd::IGraphDecomposition` is a `htd::ILabeledMultiHypergraph`.

- `htd::ITreeDecomposition`

  While plain graph decompositions represented by the decomposition interface `htd::IGraphDecomposition` are basically nothing more than arbitrary graphs which can take the information about the bag content assigned to a vertex as well as

custom additional labels for all vertices and edges, tree decompositions represented by the interface `htd::ITreeDecomposition` offer much more functionality.

For instance, each tree decomposition offers functions to efficiently access its join, introduce and forget nodes. Furthermore, one can use the write-capable extension interface `htd::IMutableTreeDecomposition` to manipulate the tree not only by adding children to nodes but also by adding parents (useful for creating intermediate nodes), deleting whole subtrees as well as by re-attaching nodes to totally different parents, thus allowing almost any tree manipulation one can think of.

Inheritance:

Each `htd::ITreeDecomposition` is a `htd::IGraphDecomposition`.

- `htd::IPathDecomposition`

  Path decompositions are tree decompositions without join nodes. In order to support this type of decompositions optimally, *htd* provides a special interface for path decompositions. By checking inheritance from this interface, algorithms sometimes can take shortcuts as they can rely on the fact that each vertex has at most one child.

  Inheritance:

  Each `htd::IPathDecomposition` is a `htd::ITreeDecomposition`.

- `htd::IHypertreeDecomposition`

  The most involved type of graph decompositions currently supported by *htd* are hypertree decompositions [GLS02]. Hypertree decompositions extend the properties of tree decompositions by additionally adding functionality to retrieve the subsets of hyperedges from the input graph which are needed to cover the bag content of the node under focus.

  Note that, generally, one could also implement this functionality by adding a custom label to each vertex of a tree decomposition, but by implementing this interface, efficiency can be increased.

  Inheritance:

  Each `htd::IHypertreeDecomposition` is a `htd::ITreeDecomposition`.

Analogous to the interfaces specific to the graph types, we again distinguish between read-only and write-capable decomposition interfaces. In the same way as before, the name of the write-capable interface for the four read-only decomposition interfaces is constructed by adding the character sequence “Mutable” between the letter ‘I’ and the remainder of the graph class name. As one example, `htd::IMutableTreeDecomposition` is the mutable counterpart of the interface class `htd::ITreeDecomposition`.

4.2.4 Decomposition Algorithms

In order to support the aforementioned decomposition types it is necessary for a good code design to distinguish different types of decomposition algorithms. The following algorithms take an input graph of type `htd::IMultiHypergraph`, that is, they can be fed with any graph type *htd* supports, and they return a pointer to a decomposition of the corresponding type.

- `htd::IGraphDecompositionAlgorithm`

  This type of algorithm provides the base interface for all decomposition algorithms in the context of *htd*. Algorithms implementing this interface partition the input graph and return a labeled multi-hypergraph of type `htd::IGraphDecomposition`.

  Note that there is, in general, neither a guarantee nor a need that the vertices in the decomposition, representing the partitions of the input graph, form a single connected component. Therefore, one can implement any decomposition algorithm one can think of using this basic interface class.

- `htd::ITreeDecompositionAlgorithm`

  The purpose of algorithms implementing this specialized interface is to optimally serve the needs of dynamic programming algorithms which rely on the use of tree decompositions. In contrast to `htd::IGraphDecompositionAlgorithm`, the basic type for all decomposition algorithms, algorithms which implement the interface `htd::ITreeDecompositionAlgorithm` are somewhat more involved. This is because they return labeled trees of type `htd::ITreeDecomposition`, thus requiring that the output graph to consist of a single connected, but cycle-free, component.

  Extends:

  `htd::IGraphDecompositionAlgorithm`.

- `htd::IPathDecompositionAlgorithm`

  When we request that the resulting decomposition is cycle-free and that it must not contain vertices with more than two neighbors, one is perfectly served using decomposition algorithms of type `htd::IPathDecompositionAlgorithm` as they return labeled paths of type `htd::IPathDecomposition`.

  Extends:

  `htd::ITreeDecompositionAlgorithm`.

- `htd::IHypertreeDecompositionAlgorithm`

  In cases where the output of the decomposition algorithm shall be a labeled tree of type `htd::IHypertreeDecomposition`, one can use the designated algorithms of type `htd::IHypertreeDecompositionAlgorithm`.

  Extends:

  `htd::ITreeDecompositionAlgorithm`

Note that each interface mentioned above can also be called with additional parameters dedicated to desired manipulations. In cases where these additional parameters are provided, the decomposition algorithm is required to return a decomposition which fulfills all criteria requested via the provided manipulation operations. This feature often dramatically reduces implementation effort and it significantly improves readability and maintainability of the code to be written by the developer. More details about the built-in manipulation operations follow subsequently and examples how to use (custom) manipulation operations can be found in Section 4.4.

4.2.5 Decomposition Manipulation Algorithms

During the design stage of *htd*, one of the main goals was the ability to customize the resulting decomposition without requiring tedious post-processing work at developer side. For this reason, *htd* provides various, built-in manipulation operations and also allows to extend their functionality easily by implementing the corresponding interface classes which are of type `htd::IDecompositionManipulationOperation`.

For each of the decomposition types which *htd* distinguishes there exists a corresponding, specialized interface for tailored manipulation operations. The distinction into these specialized interface classes allows to take advantage of shortcuts potentially originating from the features of the graph types on which the operations can be applied, e.g., there are no join nodes in path decompositions, hence we do not have to handle them.

The manipulation operations can be applied directly by the decomposition algorithms or also in a post-processing step. In the latter case it is required to cast the read-only decomposition returned by the decomposition algorithm to the corresponding write-capable one as the manipulation clearly involves updating information of the decomposition. Supporting the development process and maintainability of the code, *htd* currently provides the following built-in manipulation operations:

- `htd::AddEmptyLeavesOperation`

  It often reduces the effort needed to implement dynamic programming algorithms when it is guaranteed that leaf nodes always have an empty bag. This is primarily because, then, one does not need to treat leaves as a special case.

  To ensure that a tree or path decomposition has only leaves with empty bags, *htd* provides the manipulation operation `htd::AddEmptyLeavesOperation` which simply adds a new child with empty bag to each leaf node which does not already have an empty bag.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::AddEmptyRootOperation`

  This operation ensures that the root node of a tree or path decomposition has an empty bag. Similar to the operation `htd::AddEmptyLeavesOperation`, the manipulation operation `htd::AddEmptyRootOperation` often helps to reduce the amount of special cases which one has to think about during development of dynamic programming algorithms.

  This becomes apparent when we look at the fact that without empty root one still has to check correctness of partial solutions by taking into account that the vertices in the root still have to be forgotten. With empty root, this final check can often be done exactly by the same procedure as for all other forget nodes.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::AddIdenticalJoinNodeParentOperation`

  Sometimes it is needed to do some complex post-processing of join nodes which cannot be done directly in the dynamic programming step belonging to the respective join node, like it was needed in [AMW16a]. For this purpose, *htd* offers the class `htd::AddIdenticalJoinNodeParentOperation` which ensures that each join node has a parent node with identical bag content. With this guarantee, one can easily implement the desired post-processing functionality without having to handle various special cases.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`

- `htd::CompressionOperation`

  All built-in implementations of decomposition algorithm in *htd* guarantee that the resulting decomposition is minimal in the sense that all bags in the decomposition are subset-maximal, hence no subsumed bags are contained. Clearly, this guarantee only holds as long as one does not apply manipulation operations which add nodes with bags subsumed by other nodes' bags or which manipulate existing bag contents.

  Especially for undoing manipulations applied beforehand or in order to compress tree and path decompositions computed by custom decomposition algorithms, *htd* provides the class `htd::CompressionOperation` which efficiently removes all vertices from the given decomposition which are not subset-maximal.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::ExchangeNodeReplacementOperation`

  Exchange nodes are inner nodes of decompositions where some vertices are forgotten and, at the same time, some vertices are introduced. Sometimes one wants to avoid these situations and wants to deal with introduce and forget nodes separately in the dynamic programming algorithm. For this purpose, one can employ the operation `htd::ExchangeNodeReplacementOperation` which replaces each exchange node with one forget node and one introduce node. In order to keep the decomposition width unchanged, the forget node will be the child node of the introduce node.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::InducedSubgraphLabelingOperation`

  Although the bag content of a vertex is very important, dynamic programming algorithms also need the information about the subgraph of the original graph which is induced by the vertices contained in a bag as this information is the actual part of input data on which the algorithm operates. Especially for large graphs finding the (hyper-)edges which are induced by the bag content can be extremely expensive.

  As we will observe later (see Section 4.2.7), all built-in decomposition algorithms of *htd* already compute this information very efficiently and store the outcome directly in the resulting decomposition from which it can be accessed easily. Nevertheless, the class `htd::InducedSubgraphLabelingOperation` allows to automatically add an additional label to each decomposition node containing the set of induced edges. This is especially useful to efficiently compute the set of induced edges when a custom decomposition algorithm does not expose this information directly.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::JoinNodeNormalizationOperation`

  Join nodes can be very complex to handle in dynamic programming algorithms when the children's bags differ from the respective parent node's bag. Especially in the early stages of the development of dynamic programming algorithms, but also later on, it can be very useful to have the guarantee that join nodes and their children share the same bag content. For this purpose, *htd* offers the class `htd::JoinNodeNormalizationOperation`.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`

- `htd::JoinNodeReplacementOperation`

  It is assumed that path decompositions, i.e., tree decompositions without join nodes, are to be preferred over tree decompositions with join nodes if the width is equal, but also when it is only slightly worse, it can pay off to avoid join nodes. For this purpose, *htd* offers `htd::JoinNodeReplacementOperation`. This operation takes a tree decomposition as input and replaces join nodes by re-arranging their children and combining their bags so that the result is a path decomposition.

  Note that, currently, this operation is experimental. It proceeds recursively starting from the root node of the given tree decomposition. As long as the visited nodes do not have more than one child, the algorithm simply moves on to the next unvisited node and it returns when it reaches a leaf. Whenever the algorithm observes a join node, it takes one of the join node's children as the so-called *attachment point*. Then, one of the remaining children is re-attached as intermediate node between the attachment point and the join node currently under focus. In this way, the join node looses a child, namely exactly the one which is moved in order to act as new parent of the attachment point. To avoid breaking the connectedness criterion, the bag content of this moved child must be expanded by the vertices which are in the set intersection of the current attachment point's bag and the current join node's bag. If the current child is a join node itself, we enter the recursion. After a child is fully processed, it is set as the new attachment point and the procedure continues until all children of the current join node are processed. In this way, the connectedness condition is never violated and we finally obtain a tree decomposition without join nodes because all children of former join nodes are re-arranged in such a way that each node of the decomposition has at most one child.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`

- `htd::LimitChildCountOperation`

  Due to the fact that joins, in many cases, can be carried out more efficiently when the number of children per join node is bounded, *htd* offers the manipulation operation `htd::LimitChildCountOperation` which limits the number of children per node to a pre-definable upper bound. This is achieved by adding intermediate nodes with bag content identical to the join node under focus and distributing the supernumerous children properly among these additional intermediate nodes.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`

- `htd::LimitMaximumForgottenVertexCountOperation`

  Depending on the actual implementation of the dynamic programming algorithm, it can be beneficial to have a known upper bound for forgotten vertices in order to carry

out some algorithmic steps more efficiently. Also for debugging purposes it is often useful to have only small changes between neighboring nodes of the decomposition and so it would be nice to have a built-in manipulation to actually enforce such a limit efficiently. The class `htd::LimitMaximumForgottenVertexCountOperation` is developed exactly for this purpose.

Implements the following interfaces:

- `htd::ITreeDecompositionManipulationOperation`
- `htd::IPathDecompositionManipulationOperation`

- `htd::LimitMaximumIntroducedVertexCountOperation`

  Analogous to the aforementioned manipulation operation which allows to limit the maximum number of forgotten vertices for each decomposition node, the class `htd::LimitMaximumIntroducedVertexCountOperation` can be used to limit the maximum number of introduced vertices for each node of the decomposition.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::TreeDecompositionOptimizationOperation`

  This is probably the most involved manipulation operation currently implemented in *htd*. It allows to automatically change the root of a given tree so that the outcome of a provided fitness function is maximized. Using this operation, one can easily get a customized decomposition in a few lines of code without having to touch other portions of the code.

  Although the operation at hand is a rather complex and powerful one, it is geared towards performance and it is fully compatible with other manipulations, that is, the algorithms behind `htd::TreeDecompositionOptimizationOperation` will optimize the tree decomposition considering all desired manipulation operations.

  For even better controllability of the optimization operation one can use different vertex selection strategies to filter the vertices which shall be considered as new root node. This allows to improve performance especially for large decompositions as there is no need to exhaustively check for all vertices in the tree decomposition if they are the optimal root node. Clearly, the built-in collection of vertex selection strategies can be easily extended by developers by implementing the corresponding, simple interface. A detailed explanation of this operation is given in Section 4.4.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`

- `htd::ILabelingFunction`

  This interface is used by decomposition manipulation operations which generate labels for a corresponding bag. Labeling functions take an input graph of type `htd::IMultiHypergraph` and a sorted set of vertices, representing the bag content of a decomposition node, and they return a new label of type `htd::ILabel`. The actual data type of the value of the returned label is dependent on the implementation of the respective labeling function class. Generally, one can use any data type supported in C++.

  For convenience, one can access the concrete label value via the template function `htd::accessLabel` which takes a template argument representing the data type of the label value as well as a reference to the label and it returns the concrete label value in the given data type.

  Note that, if a labeling function is provided to a decomposition algorithm, the labeling function will be applied automatically to each vertex of the resulting decomposition. This makes using custom labels on the one hand very easy and on the other hand it ensures highest efficiency and maintainability of the code.

  Implements the following interfaces:

  - `htd::IDecompositionManipulationOperation`

Also here, the list can be extended easily with own algorithms by implementing the desired interface(s) mentioned above. One only has to take care that the manipulation operation does not violate the properties of the decomposition on which it is applied, i.e., the decomposition must stay valid after the manipulation is applied, otherwise it will break the functionality of the algorithms which use the modified decomposition.

4.2.6 Decomposition Normalization Algorithms

While the manipulation operations described in Section 4.2.5 are dedicated to exactly one task per operation class, one sometimes requires more complex manipulations. One example is making a given tree decomposition nice. This would involve the combination of many of the aforementioned basic manipulation operations. We refer to such complex manipulations which are generated by combining simpler manipulation operations by the term "normalizations". For convenience, *htd* offers the following built-in normalizations:

- `htd::WeakNormalizationOperation`

  When each join node of a tree decomposition only has children with a bag content identical to the bag content of the respective join node under focus, we call such decompositions weakly normalized.

  When we want to obtain a decomposition which fulfills this criterion, one can use the manipulation operation of type `htd::WeakNormalizationOperation`.

Additionally, one can specify that the root node and/or all leaf nodes of the decomposition shall be empty. The manipulation operation at hand efficiently combines the manipulation operations `htd::JoinNodeNormalizationOperation`, `htd::AddEmptyRootOperation` and `htd::AddEmptyLeavesOperation`, the last two of them being optional.

Implements the following interfaces:

- `htd::ITreeDecompositionManipulationOperation`
- `htd::IPathDecompositionManipulationOperation`

- `htd::SemiNormalizationOperation`

  Semi-normalized tree decompositions in the context of *htd* are tree decompositions where each join node has exactly two children with the same bag content as the respective join node. Hence, the operation `htd::SemiNormalizationOperation` extends the class `htd::WeakNormalizationOperation` by combining it with the manipulation operation `htd::LimitChildCountOperation` where the child limit is set to 2.

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

- `htd::NormalizationOperation`

  Sometimes one wants to work in the dynamic programming algorithm with a fully normalized (nice) decomposition, i.e., a tree (or path) decomposition where join nodes and their children have the same bag content and the bag contents between adjacent non-join nodes differ in at most one element.

  Specifically for this purpose, *htd* offers the class `htd::NormalizationOperation` which extends the class `htd::SemiNormalizationOperation` by combining it with the manipulation operations `htd::ExchangeNodeReplacementOperation`, `htd::LimitMaximumForgottenVertexCountOperation` (with a vertex limit of 1) as well as `htd::LimitMaximumIntroducedVertexCountOperation` (with a vertex limit of 1).

  Implements the following interfaces:

  - `htd::ITreeDecompositionManipulationOperation`
  - `htd::IPathDecompositionManipulationOperation`

4.2.7 Useful Additional Functionality

An efficient way to compute tree decompositions is a very important ingredient of an application based on dynamic programming. While this is a widely accepted statement, a tree decomposition alone often is rather worthless as long as there is no way to efficiently retrieve the information stored in it. For this reason, *htd* offers a wide range of utility functions. A small selection of them is given in the following list:

- Easy Retrieval of Induced Subgraph Information

  Maybe one of the most unique features of *htd* in contrast to other tree decomposition frameworks is the fact that all implementations of graph decomposition algorithms implemented in *htd* automatically compute for each bag of the decomposition the subgraph of the input graph which is induced by the respective bag content. By using the function called "inducedHyperedges" one can easily and with almost no cost obtain the hyperedges which are induced by the bag with the given ID. This can lead to a significant performance boost as input graphs can have millions of edges and doing a subset check for each of them in each bag in the context of a dynamic programming algorithm can be expensive.

- Tree and Graph Traversal Algorithms

  Due to the fact that they are in many cases easier to develop, describe and to implement, graph traversal algorithms often use recursion. A big issue with recursion is the fact that as soon as the graphs reach a certain size, the proper operation of practical implementations is no longer guaranteed as the program stack is no longer capable of holding the information needed by the recursive function calls. Therefore, we provide the following interfaces and their non-recursive implementations in *htd*:

  Built-In Implementations of `htd::IGraphTraversal`:

  The following two algorithms, the first one implementing breadth-first and the second one depth-first traversal, traverse a graph beginning from a custom starting vertex and take a lambda expression which is called for each visited vertex with the information about the vertex at hand, its predecessor during the traversal and its distance from the starting vertex. In this way, one can easily determine even complex characteristic numbers (like the diameter) of the graph which is traversed.

  - `htd::BreadthFirstGraphTraversal`
  - `htd::DepthFirstGraphTraversal`

  Built-In Implementations of `htd::ITreeTraversal`:

  The interface `htd::ITreeTraversal` extends `htd::IGraphTraversal` and, as the name suggests, it is dedicated to trees. Here, the predecessor of a vertex is always identical to its parent. Hence, there is no need to spend time for looking up the parent of the vertex currently visited as it is provided for free to the lambda

expression. Note at this point that the parent of the root node in the context of *htd* is the "undefined vertex" referenced by the ID 0. The following three tree traversal algorithms are currently implemented in *htd*:

  - `htd::InOrderTreeTraversal`
  - `htd::PreOrderTreeTraversal`
  - `htd::PostOrderTreeTraversal`

- Connected Component Algorithms

  Sometimes it can be beneficial to preprocess a given input graph before decomposing it. Depending on the actual application scenario, one possibility of preprocessing a graph is by investigation of its (strongly) connected components. The following interfaces and their implementations help to implement such a preprocessing step effectively and efficiently:

  Built-In Implementations of `htd::IConnectedComponentAlgorithm`:

  - `htd::DepthFirstConnectedComponentAlgorithm`
    This algorithm for determining connected components is based on depth-first search and internally uses the class `htd::DepthFirstGraphTraversal`.

  Built-In Implementations of `htd::IStronglyConnectedComponentAlgorithm`:

  - `htd::TarjanStronglyConnectedComponentAlgorithm`
    This class implements Tarjan's algorithm for determining strongly connected components of direct graphs which is described in [Tar72].

4.2.8 Implementation Guidelines

The following section is a short guide to the most important design rules to which the interfaces and algorithms in *htd* conform. Especially before developing own algorithms one should read and follow the provided guidelines in order to avoid breaking functionality.

Class Names Due to the fact that C++ does not use a special keyword which allow to discriminate interfaces from (abstract) classes we define that each class name starting with the the capital letter "I" followed by another capital letter is considered an interface, i.e., a class with pure virtual functions only.

Function Arguments and Return Types Whenever a function receives a reference which is not modified this reference is marked as "const". Conversely, when a reference is not marked as "const" one can assume that the object will be modified inside the function and one should be careful in cases where it is desired to keep an unmodified variant of the input object. Similarly, when a function returns a non-const reference, one is free to modify the underlying object.

A crucial point is the memory management in cases where functions receive and/or return pointers to objects. In cases where the function argument is a non-const pointer, the function is required to take over control over the memory region. That is, the function must take care that the memory region is properly freed. This also applies to functions which take collections of non-const pointers. When a function returns a non-const pointer, one must free the resources after using them. Note that functions receiving a const-pointer will not free the pointer, so one must take care of freeing the resources.

WARNING: You must not free the memory of const-pointers returned by functions, you must not provide the same non-const pointer multiple times as a function argument and you must not free objects which were given to a function via a non-const pointer outside the respective function boundaries as this will probably lead to memory corruptions.

To be on the safe side, don't access or modify to object to which the non-const pointer points after it was used as a function argument. In order to re-use it, one can easily create a deep copy of almost all classes in *htd* via their appropriate `clone` method.

Type Conversions and Casts Note that its a safe operation to up-cast from a read-only graph or decomposition type to the corresponding write-capable one. That is, one can use the `dynamic_cast` function provided in the C++ language specification to convert, for instance, a pointer or reference to `htd::IGraph` to a valid pointer or reference to `htd::IMutableGraph`. This convertibility must also be guaranteed by custom implementations in order to avoid breaking the functionality of algorithms.

WARNING: It is important to note that casts of the aforementioned kind are only a safe and permitted operation for directly related interfaces. For instance, although each object of type `htd::ITreeDecomposition` is also an object of type `htd::IMultiHypergraph` one cannot cast `htd::ITreeDecomposition` to `htd::IMutableMultiHypergraph`. A type cast between these two interfaces is neither possible nor permitted as hypergraphs allow cycles and trees do not.

4.3 Algorithm Engineering

During the development of *htd*, a lot of time was spent on implementing, extending and optimizing the algorithms which contribute to the library. The extension of (existing) algorithms was needed to be able to incorporate all customization capabilities of *htd* and the optimization is needed to be able to efficiently deal with large input graphs.

Subsequently, we will discuss in detail our approach how to accelerate the most-crucial parts for computing tree decompositions in the context of *htd*, namely the algorithm for computing the Min-Fill vertex elimination ordering and the algorithm for computing the actual tree decomposition via Bucket Elimination [Dec99, McM04, Sch06]. We chose these two algorithms for presentation here as they represent well-established techniques and we want to share our findings in order to speed up development of implementations of Min-Fill and Bucket Elimination in future projects.

4.3.1 Accelerating Min-Fill

Min-Fill is a prominent heuristic for computing vertex elimination orderings which often produces good results, i.e., tree decompositions of low width, in practice [KBvH01]. Like any greedy triangulation algorithm, Min-Fill follows the schema that, given an input graph $\mathcal{G} = (V, E)$, in each iteration a vertex $v \in V$ is chosen based on a given criterion and then a clique incorporating all of v's neighbors is formed in $\mathcal{G}$ by adding the so-called *fill edges*. The vertex v is then removed from $\mathcal{G}$ and stored in the next position of the resulting ordering. These simple steps are repeated until $\mathcal{G}$ finally is empty.

In the case of Min-Fill, the criterion for selecting the vertex to be removed is the following: In each iteration, we select and eliminate the vertex which requires the least amount of fill edges to be added in order to form a clique of all its neighbors. As, in general, there are multiple vertices with the same amount of required fill edges, ties are broken randomly.

Figure 4.2 shows an example input graph for the Min-Fill graph triangulation algorithm. The vertex labels denote the *fill value* of the corresponding vertex. For instance, Vertex a has a fill value of 2 as we need two additional edges, namely (b, d) and (b, e), in order to create a clique containing all of a's neighbors which are given by the vertices named b, d and e.

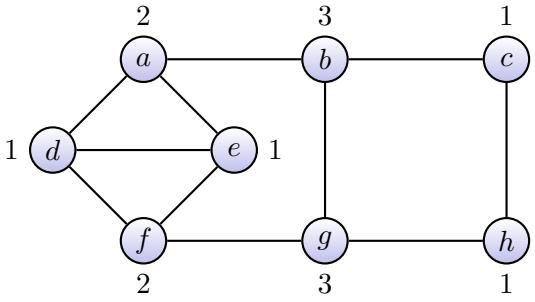


Figure 4.2: Example Input Graph for Min-Fill

The pseudo-code of the Min-Fill heuristic is shown in Algorithm 4.1. Note that, although our definition of triangulation algorithms given before refers to graphs, the pseudo-code in Algorithm 4.1 takes an arbitrary (constraint) hypergraph as input. This is a valid move because we can easily transform any (constraint) hypergraph to its corresponding primal graph (containing the same vertices as the input hypergraph) by introducing an edge between each pair of vertices in a hyperedge.

Basically, the pseudo-code does not tell us how to efficiently determine the fill value of a vertex. By the term "fill value" we refer to the amount of fill edges that need to be added in order to form a clique incorporating all neighbors of the respective vertex. Subsequently, we will give detailed insights on how to accelerate the computation of a Min-Fill vertex elimination ordering.

Algorithm 4.1: Min-Fill (Simple Pseudo-Code)

Input: A (constraint) hypergraph $\mathcal{H} = (V, H)$
Result: A vertex elimination ordering $\sigma = \{\sigma_1 \dots \sigma_n\}$ of the vertices in V

```
1  Let G = (V, E) be the primal graph of H.
2  σ ← [];                                   // List σ is initially empty.
3  while G is not empty do
       /* Ties are broken randomly!                                    */
4      Select a vertex v_x ∈ V whose elimination requires the least amount of edges to
        be added;
       /* Add necessary fill-in edges.                                 */
5      neighbors ← ({v | (v_x, v) ∈ E} ∪ {v | (v, v_x) ∈ E}) \ {v_x};
6      for v_1 ∈ neighbors do
7          for v_2 ∈ neighbors do
8              if v_1 ≠ v_2 and (v_1, v_2) ∉ E then
9                  E ← E ∪ {(v_1, v_2)};
10             end
11         end
12     end
       /* Remove v_x from G.                                           */
13     V ← V \ {v_x};
14     E ← E \ {(v_x, v) | (v_x, v) ∈ E} ;
15     E ← E \ {(v, v_x) | (v, v_x) ∈ E} ;
16     Append v_x to σ;
17 end
18 return σ;
```

First, let us define the data structures which the algorithm will rely on and fix the notation we will use throughout the following explanation: As underlying data structure for storing the graph we use a simple adjacency list. Additionally we use a dictionary which holds the current fill value for each vertex which is not yet removed and we maintain a set of vertices which we refer to as the *pool* and which contains all vertices with minimum fill value. We define that, given a graph $\mathcal{G} = (V, E)$, the (one-hop) neighborhood $N_1(v)$ of a vertex $v \in V$ is represented by the set $\{v_x|(v_x, v) \in E\} \cup \{v_x|(v, v_x) \in E\} \cup \{v\}$ and the two-hop neighborhood $N_2(v)$ of a vertex $v \in V$ is given by the set $N_1(v) \cup (\bigcup_{v_x \in N_1(v)}(v_y|(v_y, v_x) \in E\} \cup \{v_y|(v_x, v_y) \in E\}))$. That is, $N_1(v)$ contains all vertices which are reachable from v in at most one hop and $N_2(v)$ contains all vertices which are reachable from v within not more than two hops. Based on these definitions, we define the following notions with respect to a removed vertex $v_r \in V$:

Definition 12. *For a vertex $v \in N_2(v_r)$, we call the set $N_e(v) = (N_1(v) \cap N_1(v_r)) \setminus \{v_r\}$ the* existing neighbors *of v. Furthermore, we call the set $N_u(v) = N_1(v) \setminus N_1(v_r)$ the* unaffected neighbors *of v and we refer to $N_a(v) = (N_1(v_r) \setminus N_1(v)) \setminus \{v_r\}$ as the* additional neighbors *of v. Note that $N_e(v) \cup N_u(v) \cup N_a(v) \equiv N(v) \setminus \{v_r\}$ holds and that the sets are disjoint, i.e., existing, unaffected and additional neighbors of v form a partition of v's neighborhood with v_r removed.*

The intuition behind the terms *existing, unaffected* and *additional vertex* is the following: The existing vertices of a vertex v are those vertices which are directly manipulated, i.e., the vertices whose neighborhood will be updated. The additional vertices are those vertices which are to be added to the neighborhood of v in order to create a clique between the neighbors of v_r and the unaffected vertices are those neighbors of v which are not adjacent to v_r. Note that $N_a(v) = \emptyset$ holds for all vertices $v \in N_2(v_r) \setminus N_1(v_r)$ where v_r is the vertex eliminated in the current iteration. This is because of the fact that only vertices which are direct neighbors of the eliminated vertex v_r will potentially get additional neighbors due to the creation of the clique.

As mentioned before, in order to perform the selection step in Line 4 of Algorithm 4.1 efficiently, we decide to keep track of the fill value of each vertex by storing it in a dictionary data structure which allows for fast lookup. That is, instead of computing the fill value of a vertex from scratch in each iteration, we just update the fill value accordingly. In this way, we can easily manage a pool of vertices which currently have the lowest fill values without having to re-compute the same information over and over for vertices which were not affected in a previous iteration. From this pool of vertices with minimum fill value we can then simply select a vertex at random in order to perform the next iteration.

To illustrate how to efficiently compute the necessary fill value changes, we refine the pseudo-code provided in Algorithm 4.1. The enhanced version of the pseudo-code is given in Algorithm 4.2. While the procedures of eliminating a vertex and adding it to the ordering stay unchanged, the extended algorithm now shows how one can easily update the fill value of those vertices (and only of those vertices) which are affected by a vertex elimination step.

In the first iteration of the algorithm, we need to compute the actual fill value for each vertex. With this information we can initialize the variables *fill*, *minfill* and *pool* (see Line 6 of Algorithm 4.2). After doing so we never need to compute the full fill value of a vertex again. We completely rely on updating the fill values as this significantly improves performance in practice. This is based on the fact that for updating the fill value it often suffices to consider at most two out of the three partitions – $N_e(v)$, $N_u(v)$ and $N_a(v)$ – of the neighborhood relation.

In Lines 8–10 of Algorithm 4.2 we update the pool if it is empty. This can occur whenever the last vertex with fill value equal to *minfill* was eliminated in the iteration before or when all vertices which were in the pool in the iteration before got updated to a fill value

Algorithm 4.2: Min-Fill (Verbose Pseudo-Code)

Input: A (constraint) hypergraph $\mathcal{H} = (V, H)$
Result: A vertex elimination ordering $\sigma = \{\sigma_1 \dots \sigma_n\}$ of the vertices in V

```
1  Let G = (V, E) be the primal graph of H.
2  σ ← [];                                  // List σ is initially empty.
3  fill ← [];  // The dictionary for the fill value of each vertex.
4  pool ← ∅;       // The set of vertices with minimum fill value.
5  minfill ← ∞;                             // The minimum fill value.
6  Initialize fill, minfill and pool based on G.
7  while G is not empty do
8      if pool == ∅ then
9          Initialize minfill and pool based on fill.
10     end
11     Select randomly a vertex v_x ∈ pool;
12     if fill[v_x] == 0 then
13         for v ∈ N_1(v_x) \ {v_x} where fill[v] > 0 do
14             fill[v] ← fill[v] − |N_1(v) \ N_1(v_x)|;
15         end
16     else
17         for v ∈ N_1(v_x) \ {v_x} do
18             fill[v] ← updateNeighbor(G, v_x, v, fill[v]);
19         end
20         for v ∈ N_2(v_x) \ ({v_x} ∪ {u|u ∈ N_1(v_x), N_u(u) == ∅ || N_a(u) == ∅}) do
21             for u ∈ N_e(v) do
22                 fill[v] ← fill[v] − |{t|t ∈ N_e(v), t > u} ∩ N_a(u)|;
23             end
24         end
25     end
26     Add necessary fill-in edges;
27     Remove v_x from G;
28     Append v_x to σ;
29 end
30 return σ;
```

which is higher than *minfill*. Note that we omit in Algorithm 4.2 the code for updating the pool content and the value of *minfill* for better readability. Basically, whenever the fill value of a vertex is updated, also the pool and, potentially, also the value *minfill* need to be updated. For instance, if the new fill value of a vertex v is less than *minfill*, *minfill* is set to the new lower bound for the fill value and the pool of vertices with minimal fill value is reset to contain only v. Similarly, if the new fill value of a vertex v is equal to *minfill* and v is not already contained in *pool*, v is added to *pool*. If the new fill value of a vertex v is greater than *minfill*, v has to be removed from the pool of vertices with minimal fill value.

In Lines 12–15 of Algorithm 4.2 the case is handled in which the eliminated vertex v_x has a fill value of 0, that is, all its neighbors already form a clique. In this case, we can simply subtract the amount of unaffected neighbors $N_u(v)$ from the fill value of each neighbor v of v_x. This update is sufficient because v does not have any additional neighbors (because the neighbors of v_x already form a clique) and the existing neighbors of v do not contribute to its fill value (for the same reason). Hence, a change of the fill value of v can be only be caused by the fact that now all missing edges between v_x and v's neighbors are no longer relevant after the elimination of v_x.

Perhaps more interestingly is the case in which the fill value of the eliminated vertex is greater than 0. In those cases we first enter the loop at Lines 17–19 of Algorithm 4.2 for each neighbor of the eliminated vertex v_x. To update the fill value of these vertices, we use Algorithm 4.3. This helper algorithm takes as input an undirected graph, the eliminated vertex v_x, the vertex v whose fill value shall be updated and the old fill value of v. The result of the algorithm is the new fill value of v after eliminating v_x.

In the case where there are no unaffected neighbors of v, Algorithm 4.3 will simply return 0 as the new fill value because after the elimination of v_x all of v's neighbors will be together in a clique. When there exists at least one unaffected neighbor of v, we distinguish two cases: If there exist additional neighbors of v, we update the fill value of v according to the formula shown in Line 3 of Algorithm 4.3. That is, we first increase its fill value by the maximum amount of edges that can exist between the additional and the unaffected vertices and then we subtract the amount of existing edges between the additional and unaffected vertices.

In the second case, i.e., when v has some unaffected but no additional vertices, we first subtract the amount of unaffected neighbors from v's fill value because v_x is a neighbor of v and the missing edges between v_x and the unaffected neighbors no longer matter. Then we iterate over all $u \in N_e(v)$ and subtract the size of the set intersection of all additional vertices of u and of all existing neighbors of v greater than u. By considering only vertices greater than u we do not count any edge twice. In this way, we can efficiently compute the amount of edges between the existing neighbors of v which were missing but which are added during the construction of the clique between the neighbors of v_x.

This brings us back to Lines 20–23 of Algorithm 4.2. In this loop, we iterate over all two-hop neighbors of the eliminated vertex v_x (excluding v_x) which are either not directly

Algorithm 4.3: Procedure `updateNeighbor` for Algorithm 4.2

Input: An undirected graph $\mathcal{G} = (V, E)$
A vertex $v_r \in V$ which is eliminated in the current iteration
A vertex $v \in V$ whose fill value shall be updated
The old fill value f of v
Result: The new fill value of v

```
1  if |N_u(v)| > 0 then
2  |  if |N_a(v)| > 0 then
3  |  |  f ← f + (|N_a(v)| − 1) ∗ |N_u(v)| − Σ_{u∈N_a(v)} |N_u(u) ∩ N_u(v)|;
4  |  else
5  |  |  f ← f − |N_u(v)|;
6  |  |  for u ∈ N_e(v) do
7  |  |  |  f ← f − |{t|t ∈ N_e(v), t > u} ∩ N_a(u)|;
8  |  |  end
9  |  end
10 else
11 |  f ← 0;
12 end
13 return f;
```

adjacent to v_x or which have both unaffected and additional neighbors. That is, we consider again those two-hop neighbors v_x which are also direct neighbors and which were handled by Line 3 of Algorithm 4.3. Line 22 of Algorithm 4.2 follows the same idea as Line 7 of Algorithm 4.3. All what remains is to add the necessary fill-in edges, remove v_x from the graph and append it to the ordering.

Note that one can achieve additional improvements by keeping track of the total fill value – the sum of all fill values – and abort the main loop of the algorithm earlier when this counter reaches 0. This is possible because a total fill value of 0 implies that the remaining graph is a clique. The ordering can then be completed by appending the vertices in the remaining graph in arbitrary order.

4.3.2 Extending Bucket Elimination

Bucket Elimination [Dec99] was originally presented as a unifying framework for reasoning. Based on this concept, we can easily compute a tree decomposition of a given input hypergraph as shown in Algorithm 4.4. The pseudo-code in Algorithm 4.4 was presented in [McM04, Sch06] and it illustrates the process of computing a tree decomposition when given an input hypergraph and a corresponding vertex elimination ordering. The algorithm first creates an empty bucket for each vertex of the input graph (Line 2) and then it proceeds by initializing the bags based on the (hyper-)edges which are contained in the graph (Lines 3–6). This happens by assigning the vertices of a given hyperedge h

to the bucket which corresponds to the lowest ranked vertex in h with respect to the given elimination ordering σ. Afterwards we have to iterate over the vertex elimination ordering σ in order to obtain the complete content of each bucket and the edges between the buckets by which the final tree is constructed (Lines 7–12).

Algorithm 4.4: Bucket Elimination

Input: A (constraint) hypergraph $\mathcal{H} = (V, H)$
A vertex elimination ordering $\sigma = \{\sigma_1 \dots \sigma_n\}$ of the vertices in V
Result: A tree decomposition $(\mathcal{T}, \chi)$ of $\mathcal{H}$

1 $E = \emptyset$;
2 Create an empty bucket B_{σ_i} for each vertex $\sigma_i \in \sigma$: $\chi(B_{\sigma_i}) \leftarrow \emptyset$.
3 **for** $h \in H$ **do**
4 Let v be the vertex in h which is ranked at lowest position in σ among all vertices in h;
5 $\chi(B_v) \leftarrow \chi(B_v) \cup \mathit{vertices}(h)$;
6 **end**
7 **for** $i \in \{1..n\}$ **do**
/* Create temporary vertex set A based on bucket B_{v_i}. */
8 Let $A = \chi(B_{v_i}) \setminus \{v_i\}$;
/* Determine lowest ranked vertex in A based on σ. */
9 Let $v_j \in A$ be the next in A following v_i in σ;
10 $\chi(B_{v_j}) = \chi(B_{v_j}) \cup A$;
11 $E = E \cup \{(B_{v_i}, B_{v_j})\}$;
12 **end**
13 **return** $((B, E), \chi)$ where $B = \{B_1 \dots B_n\}$;

Subsequently, we propose an extension of Algorithm 4.4 which can be extremely helpful for the development of dynamic programming algorithms. For dynamic programming on tree decompositions we always need the information about the (hyper-)edges which are induced by a bag. A naive approach where we check in each bag of the tree decomposition the induced edges soon becomes a bottleneck when the input graph contains thousands of edges. To overcome this issue, we use the following trick in *htd*, so that all the work is done directly by the Bucket Elimination procedure:

In fact, at Line 5 of Algorithm 4.4 we can store the target bucket of a (hyper-)edge, that is, we can use a dictionary which for each (hyper-)edge holds the bucket in which it is fully contained. The speedup comes from the fact that while finding the first bucket in a given tree decomposition in which an edge is contained can be tedious, internally, during execution of the Bucket Elimination algorithm, we get this valuable information for free.

After implementing this small change, all that remains in order to find the induced edges for each bag is to start, for each edge in the input graph, a kind of depth limited search

in the tree decomposition starting from the target bag of the respective edge. The limit in this case is given by the bucket where the edge is no longer fully contained. That is, we follow a branch until we reach a bag which does not contain the edge as a whole anymore and then we backtrack.

4.4 *htd* at Work

In this section we will have an in-depth look at how to use *htd* optimally in different situations and how to adapt it according to the actual needs. The focus of this explanation is not only to introduce the developer to the concrete classes and interfaces which can be useful for implementing dynamic programming algorithms but also to shed some light at the interplay between them. This section will first cover details on loading input data and then we will have a look at how to customize tree decompositions. Finally we will make use of *htd*'s utility functions to show how the information stored in the resulting decomposition can be incorporated in a given dynamic programming algorithm.

4.4.1 The Dynamic Programming Algorithm

For the remainder of this section, let's assume that we want to solve the Minimum Dominating Set problem via dynamic programming on tree decompositions. That is, we try to find cardinality-minimal subsets $S \subseteq V$ of a graph $\mathcal{G} = (V, E)$ such that each vertex $v \in V$ is either contained in S or adjacent to at least one vertex in S. Furthermore, assume that we are interested in determining the complete set of all such solutions and that the dynamic programming algorithm we want to implement follows exactly the schema we presented in Section 2.3.

4.4.2 Loading the Input Data

In general, there are many ways to parse input data and even the origin of the data stream from which to read may vary between different scenarios. For the following example code we assume that the input is stored in files conforming to the following variant of the DIMACS graph format which is also used as the official input format of Track A of the "First Parameterized Algorithms and Computational Experiments Challenge" (PACE 2016, see `https://pacechallenge.wordpress.com/track-a-treewidth/`):

- The file starts with a header matching the pattern "`p tw` *Vertices Edges*", where the placeholder *Vertices* represents the number of vertices of the input graph and *Edges* is a placeholder for the number of edges which are contained in the input graph. The vertices are numbered between 1 and *Vertices*.
- All lines starting with the letter 'c' are treated as comments.
- Each remaining line (there must be exactly *Edges* such lines) represents an edge. An edge information consists of exactly two vertex identifiers (numbers between 1 and *Vertices*) separated by a single space.

Subsequently we will sketch how one can simply and effectively create a new instance of `htd::IMultiGraph` which contains all relevant information of a given input graph. Note that the following C++ code snippets are not optimized and that we omit error handling here for brevity. An efficient implementation for parsing input files of the aforementioned format, also containing error handling routines, is provided via the class `htd_main::GrFormatImporter` which can be found in the folder "src/htd_main" located in the main folder of the *htd* project.

The project *htd_main*, which is providing a command-line executable for the *htd* project, also contains a collection of other useful classes allowing to import additional, often-used graph types. These classes can act also as a starting point for the development of custom importers.

To come back to our example, let's assume that there exists a pointer to an object of type `htd::LibraryInstance` called "manager" and assume furthermore that we already parsed the first non-comment line of an input file in the aforementioned format. At this point, we already know the total amount of vertices the new graph will contain and we can store it in the variable *Vertices*. With this information we can then create a new instance of the desired graph type using the code shown in Listing 4.4.

```
1 // Get a new instance of the default multi-graph implementation.
2 htd::IMutableMultiGraph * g =
3     manager->multiGraphFactory().createInstance(Vertices);
```

Listing 4.4: Creating a New Instance of `htd::IMutableMultiGraph` (Variant 1)

Another way of achieving the same result, i.e., creating a new graph instance of non-zero size, is shown in Listing 4.5. While the code in Listing 4.4 automatically initializes the graph to the requested size, the second variant first creates an empty graph and afterwards adds the desired number of vertices to it via a bulk operation. In both cases, the created vertices are numbered between 1 and *Vertices*. When the method `addVertices` is called repeatedly, the range of the newly inserted vertices starts from the identifier of the vertex added last, incremented by 1. Alternatively, if it is needed to add a single vertex to a graph, one can use the method `addVertex`. To add a vertex to a tree or path, *htd* offers the methods `insertRoot` (for creating the first vertex of the tree or path), `addChild` and `addParent`. Note that the creation and initialization of all other (labeled) graph types and decomposition types works analogously.

```
1 // Get a new, empty instance of the default multi-graph implementation.
2 htd::IMutableMultiGraph * g =
3     manager->multiGraphFactory().createInstance();
4 // Add all vertices to the graph
5 g->addVertices(Vertices)
```

Listing 4.5: Creating a New Instance of `htd::IMutableMultiGraph` (Variant 2)

In our example, after the graph is properly initialized to the right size, one can simply go ahead and for each edge with endpoints *V1* and *V2* which is read from the input file

we call the code shown in Listing 4.6.

```
1 // Add a new edge with the two endpoints V1 and V2.
2 htd::id_t edgeId = g->addEdge(V1, V2);
```

Listing 4.6: Creating a New Edge

Note that for adding hyperedges, one can also use the method `addEdge` and call it with a vector (or any other collection) of vertices. The function always returns the ID of the corresponding edge. For graph types which do not allow duplicate edges, the function returns the ID of the unique edge with the same endpoints.

4.4.3 Decomposing the Input Graph

After the input file is parsed successfully, we can now decompose the graph. As already mentioned earlier, *htd* supports a variety of decomposition types together with their corresponding decomposition algorithms. Because the general schema for obtaining a decomposition of a given graph in *htd* is very similar for all decomposition types, we pick here the example of a "plain" tree decomposition of type `htd::ITreeDecomposition`.

If we just need a simple tree decomposition, we can use the code from Listing 4.7 and we will obtain a non-normalized tree decomposition where each bag is subset-maximal with respect to the other bags in the same decomposition. The algorithms in *htd* are required to be able to deal with empty and disconnected input graphs. This means that one can feed them any graph and the result will be a valid decomposition of the requested type.

```
1 // Obtain a new instance of the default tree decomposition algorithm.
2 htd::ITreeDecompositionAlgorithm * algorithm =
3     manager->treeDecompositionAlgorithmFactory().createInstance();
4 // Compute a new tree decomposition of the given graph.
5 htd::ITreeDecomposition * decomposition =
6     algorithm.computeDecomposition(*g);
```

Listing 4.7: Computing a Tree Decomposition

We can see that obtaining a plain tree decomposition of a graph for the use in a dynamic programming algorithm is a very simple task using *htd*. This is a nice observation, but often an arbitrary tree decomposition is not good enough as the absence of structural guarantees makes the design of the dynamic programming algorithms much more complex and often this increased code complexity has negative effects on the performance.

To illustrate that *htd* also makes applying modifications to tree decompositions very convenient, let's have a look at Listing 4.8. The code snippet is to be placed before `computeDecomposition` is called. In the given example, we ensure that each join node only has children for which the bag content is equal to the bag content of the respective join node. Having this guarantee is often very useful as it makes join operations much easier to implement. For instance, when we recall the dynamic programming tables from Figure 2.4, we can see that, in the context of join nodes of non-normalized tree

decompositions, we have to check for introduced and removed vertices and handle them appropriately. This is not necessary in those cases where it is guaranteed that the bags of join nodes and their children coincide.

```
1 // Ensure that each child bag of a join node matches the join node's bag.
2 algorithm->addManipulationOperation
3     (new htd::JoinNodeNormalizationOperation(manager));
```

Listing 4.8: Applying a Manipulation Operation

Clearly, one can also request multiple manipulation operations to be applied automatically by a decomposition algorithm. In this case, the manipulations are applied in the order they were provided. For convenience, in addition to the possibility to apply manipulations globally to all decompositions computed by an algorithm instance, one can also request manipulations directly in the call to `computeDecomposition`. In this case, all manipulation operations provided in the call to `computeDecomposition` are applied after all operations provided directly to the decomposition algorithm.

4.4.4 Decomposing the Input Graph with Optimization

Sometimes, the manipulation of a tree decomposition like shown before is not enough and we want to obtain an optimized decomposition with respect to a custom fitness function. Also for scenarios of this kind, *htd* offers an easy-to-use workflow.

To illustrate this workflow by means of our working example, let's assume that we want to obtain a tree decomposition which is of low width and whose height is minimized. Furthermore, let's assume that minimizing the width is more important than minimizing the height of the tree decomposition. To achieve this goal, we first have to define the simple fitness function shown in Listing 4.9.

```
1  class FitnessFunction : public htd::ITreeDecompositionFitnessFunction
2  {
3     public:
4        htd::FitnessEvaluation * fitness
5            (const htd::IMultiHypergraph &,
6             const htd::ITreeDecomposition & decomposition) const
7            {
8               return new htd::FitnessEvaluation(2,
9                  -(double)(decomposition.maximumBagSize()),
10                 -(double)(decomposition.height()));
11           }
12           FitnessFunction * clone(void) const
13           {
14              return new FitnessFunction();
15           }
16 };
```

Listing 4.9: Fitness Function for Minimizing Width and Height of a Tree Decomposition

Basically, a fitness function is a class with a method `fitness` which takes two parameters, the input graph and a tree decomposition, and which returns a fitness evaluation consisting of an arbitrary number of levels (corresponding to the priorities) each represented by a value of type `double`.

The constructor of `htd::FitnessEvaluation` takes a (non-empty) parameter list where the first argument of the constructor is an integer value determining the number of levels the evaluation will contain. A decomposition A is considered better than a decomposition B if the fitness evaluation of A is lexicographically greater than B's fitness evaluation. In our example, as we want to minimize the width and height of the tree decomposition, we have to negate the corresponding values before we create the fitness evaluation object. In Listing 4.9 we can also see that there is a method `clone`. This function is required by almost all classes in *htd* as we often need deep copies of an object and this is exactly the purpose of the `clone` function.

```
1  // Obtain a new instance of the default tree decomposition algorithm.
2  htd::ITreeDecompositionAlgorithm * baseAlgorithm =
3      manager->treeDecompositionAlgorithmFactory().createInstance();
4  // Create a new fitness function object.
5  FitnessFunction function;
6  // Create a new optimization operation for selecting the optimal root.
7  htd::TreeDecompositionOptimizationOperation * operation =
8      new htd::TreeDecompositionOptimizationOperation
9          (manager.get(), function);
10 // Consider at most ten randomly selected vertices as new root node.
11 operation->setVertexSelectionStrategy
12     (new htd::RandomVertexSelectionStrategy(10));
13 // Ensure that each child bag of a join node matches the join node's bag.
14 operation->addManipulationOperation
15     (new htd::JoinNodeNormalizationOperation(manager));
16 // Compute a new tree decomposition of the given graph.
17 htd::ITreeDecomposition * td = algorithm.computeDecomposition(*g);
18 // Apply the optimization operation to the tree decomposition algorithm.
19 baseAlgorithm->addManipulationOperation(operation);
20 // Create a new instance of a tree decomposition algorithm
21 // which iteratively calls the base algorithm and returns
22 // the decomposition with optimal fitness.
23 htd::IterativeImprovementTreeDecompositionAlgorithm algorithm
24     (manager.get(), baseAlgorithm, function);
25 // Compute at most ten decomposition of the input graph.
26 algorithm.setIterationCount(10);
27 // Abort the optimization process before the iteration limit is
28 // reached if no improvement was found in the last five iterations.
29 algorithm.setNonImprovementLimit(5);
```

Listing 4.10: Computing a Customized Tree Decomposition

After implementing the fitness function, the few lines of code presented in Listing 4.10 suffice to enhance the code from Listing 4.7 in such a way that the fitness function will be maximized. This happens in two steps: The tree decomposition optimization operation takes a tree decomposition and tries to change its root in such a way that the fitness function is maximized and the iterative improvement algorithm calls the base algorithm (which automatically applies the optimization operation) repeatedly and returns the decomposition having optimal fitness. Indeed, one can also use different fitness functions in the two algorithms or nest the algorithms.

For the use in own implementations, one can freely customize the set of vertices which shall be considered in the search for the optimal root by choosing a different vertex selection strategy which is used by the tree decomposition optimization operation. *htd* offers a set of built-in selection strategies but one can also define custom strategies depending on the actual needs of the dynamic programming algorithm.

Note that in the listing above, the optimization algorithm takes care of applying the join node normalization operation which is also shown in Listing 4.8. Moving the responsibility for applying the manipulations from the base algorithm towards the optimization operation is necessary because we want the fitness function to be evaluated based on the resulting decomposition. Applying the normalization operation after the optimization operation likely would destroy the property of (local) optimality of the decomposition as the height could change when normalizing the join nodes.

To track the optimization progress, an interesting aspect of the iterative improvement algorithm is the fact that we can call the function `computeDecomposition` with an additional lambda expression, like shown in Listing 4.11. In this example, the code outputs the width and height of each new decomposition computed by the algorithm.

```
1  htd::ITreeDecomposition * decomposition =
2      algorithm.computeDecomposition(*g,
3          [&](const htd::IMultiHypergraph &,
4              const htd::ITreeDecomposition &,
5              const htd::FitnessEvaluation & fitness)
6          {
7              std::size_t w = -fitness.at(0) - 1;
8              std::size_t h = -fitness.at(1);
9              std::cout << "Width: " << w << "    Height: " << h << std::endl;
10         }
11     });
```

Listing 4.11: Tracking the Optimization Progress

Finally, it is important to notice that the iterative improvement tree decomposition algorithm is safely interruptible. That is, one can call the `terminate` method of the corresponding manager and the iterative improvement algorithm will immediately return the best tree decomposition found so far or a `nullptr` in case that no decomposition was computed so far.

During the experiments we made in the context of the publication [ADMW15] it turned out that *D-FLAT* often benefits from a reduction of the bag size of join nodes. This is most probably caused by the fact that then the number of possible solution candidates which have to be joined is potentially reduced. Due to the fact that the bag size of join nodes is not affected by changing the root of a normalized tree decomposition, in this case we can even omit the class `htd::TreeDecompositionOptimizationOperation` or leave away the fitness function in its constructor in order to make it a transparent "non-operation". Because the join node bag sizes in a decomposition with normalized join nodes are potentially not influenced when selecting a different vertex as root node, we decided to present the reduction of the decomposition height in the example code in Listing 4.10. The height of a tree decomposition is highly dependent on the actual node which acts as its root and so it is a better suited application of the decomposition optimization operation of *htd* for the illustrative scenario at hand.

4.4.5 Working with the Decomposition

After the tree decomposition is computed, we still have to execute the DP algorithm on it. Due to the fact that there are various ways to implement a dynamic programming algorithm for a given problem, we do not go into details of the concrete algorithm for Minimum Dominating Set at this point. Instead, we provide in Listing 4.12 a short example how a computed tree decomposition together with the induced edges for the bags can be printed in a human-readable form as this code can act as a general starting point to get an idea how *htd* supports the implementation of dynamic programming algorithms also beyond the plain decomposition of input graphs.

```
1  htd::PreOrderTreeTraversal traversal;
2  traversal.traverse(*decomposition,
3     [&](htd::vertex_t vertex, htd::vertex_t parent, std::size_t depth)
4  {
5     for (htd::index_t index = 0; index < distanceToRoot; ++index)
6     {
7        outputStream << "    ";
8     }
9     std::cout << "NODE " << vertex << ": " <<
10       decomposition.bagContent(vertex) << std::endl;
11    for (const htd::Hyperedge & e :
          ↪ decomposition.inducedHyperedges(vertex))
12    {
13       for (htd::index_t index = 0; index < distanceToRoot + 1; ++index)
14       {
15          outputStream << "    ";
16       }
17       std::cout << e.id() << ": " << e.elements() << std::endl;
18    }
19 });
```

Listing 4.12: Printing a Tree Decomposition

The code above traverses the tree decomposition in preorder and outputs for each node its ID together with its corresponding bag content. The very helpful and powerful feature for easily deriving the (hyper-)edges induced by a bag is also illustrated in the example source code shown in Listing 4.12. With this feature one automatically has direct access to all hyperedges whose endpoints are a subset of the current bag content. In this way, one can save significant portions of the total runtime compared to tree decomposition libraries which do not provide this feature as in the latter case one has to do a subset check for each (hyper-)edge in each node's bag in the worst case.

4.4.6 Upgrading the Dynamic Programming Algorithm

At this point, we presented all aspects of *htd* which are needed to solve the Minimum Dominating Set problem. Subsequently, we will have a glance at how to utilize *htd*'s support for custom vertex labels in order to solve the problem of Minimum Weighted Dominating Set. Afterwards we will also illustrate how exploiting the full potential of *htd* makes developing dynamic programming algorithms much more convenient.

First, let's recall the definition of Minimum Weighted Dominating Set: The problem is defined on a graph $\mathcal{G} = (V, E)$ with vertex weights $W : V \to \mathbb{R}$ and we want to find all sets $S \subseteq V$ of minimum cost such that each vertex in V is either contained in S or adjacent to at least one vertex inside S. The cost of a dominating set S is defined as the sum of all vertex weights of the vertices in S.

To adapt the basic algorithm for Minimum Dominating Set so that it is able to deal with vertex weights, we just have to sum up the proper weight instead of the value 1 for each selected vertex. Clearly, we can achieve this modification of the dynamic programming algorithm presented previously by maintaining a mapping between the vertices and the corresponding weights in the algorithm, but this would involve using a global mapping variable or passing the mapping variable as a parameter to the function which computes the dynamic programming tables, probably causing high maintenance effort. As a workaround and in order to keep the code clean and highly maintainable, one can use *htd*'s functionality to add, remove and access vertex and edge labels in the context of any available decomposition and labeled graph type. An example is given in Listing 4.13 where we add a vertex label to Vertex 1 of type `double`, representing its weight.

```
1  // Get a new instance of the default multi-graph implementation.
2  htd::IMutableLabeledMultiGraph * g =
3      manager->labeledMultiGraphFactory().createInstance(Vertices);
4  // Add a vertex label "Weight" to the vertex with ID 1.
5  g->setVertexLabel("Weight", 1, new htd::Label<double>(3.14159));
6  // Access the vertex label "Weight" assigned to the vertex with ID 1.
7  double weight = htd::accessLabel<double>(g->vertexLabel("Weight", 1));
8  // Remove the vertex label "Weight" from the vertex with ID 1.
9  g->removeVertexLabel("Weight", 1);
```

Listing 4.13: Using *htd*'s Support for Custom Labels

Following this example, all what remains to do in the context of the dynamic programming algorithm is to cast the reference to `htd::IMultiHypergraph` which is used by the algorithms of *htd* as the basic graph interface to the labeled graph type at hand (which is clearly a valid and permitted up-cast) and call the desired access methods of the graph for the vertex and edge labels that are needed by the dynamic programming algorithm. We can see that the extension of the algorithm just involves changing a few lines of code while fully maintaining the readability and maintainability of the original code.

Actually, for solving the problem of Minimum Weighted Dominating Set, there is no requirement for using custom edge labels. To illustrate how a customized tree decomposition may look like, we now want to recall the SAT problem (see Section 2.1.1) as dynamic programming algorithms for this problem can benefit from such domain-specific edge labels.

Consider the following propositional formula ϕ.

$$\phi = \underbrace{(\neg a)}_{\text{Clause } c_1} \wedge \underbrace{(\neg b \vee c)}_{\text{Clause } c_2} \wedge \underbrace{(a \vee b \vee \neg c)}_{\text{Clause } c_3}$$

From Section 2.2 we know that the given formula ϕ has exactly two solutions, namely $\{a = \textbf{False}, b = \textbf{False}, c = \textbf{False}\}$ and $\{a = \textbf{False}, b = \textbf{True}, c = \textbf{True}\}$. In Section 2.3 it is shown how to obtain them based on the concept of dynamic programming on tree decompositions. For human beings, following the steps of the dynamic programming algorithm is relatively easy because we, in general, do not decompose the formula in our heads exactly like a fixed-parameter tractable algorithm. When we strictly follow the idea of such algorithms, we are in principle only allowed to access those edges in a dynamic programming step whose endpoints are a subset of the current node's bag. If this information is not provided directly by the tree decomposition, in the worst case, the whole edge set of the input graph would have to be checked to find the relevant edges. Furthermore, for the SAT problem it is also required to know the polarity of the atoms within a clause.

When using *htd*'s customized tree decompositions which always carry the information of the edges induced by a given bag we obtain the decomposition depicted at the bottom of Figure 4.3. The variable-clause incidence graph from which the given tree decomposition is obtained is shown at the top of Figure 4.3. We can see that tree decompositions computed by *htd* can be utilized to carry, apart from the induced edges, also the information about polarities.[6] Indeed, this observation also applies to other kinds of vertex and edge labels. In this way, the requested information is directly accessible within the dynamic programming algorithm and in many cases even the complete supplemental information required to solve a given problem instance can be handed over to the *htd* framework in order to enjoy automated propagation.

[6]Note that the clauses in our example do not have polarities. Therefore, the labels for the endpoints of the corresponding edges are set to N/A in Figure 4.3. This is done solely for illustrative purposes. In practice, one will probably use labels for the polarities of atoms only.

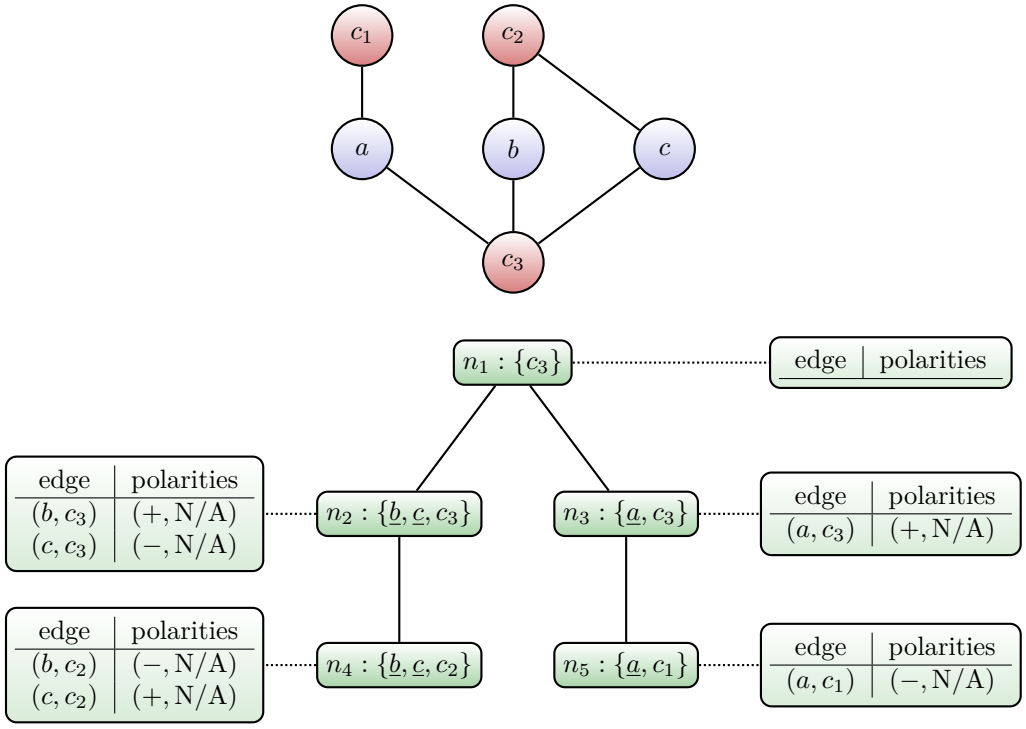


Figure 4.3: Example Graph and a Possible Customized Tree Decomposition.

4.5 Performance Characteristics

In this section we give first results regarding the performance characteristics of our decomposition framework. In order to have an indication for the actual efficiency of *htd*, we compare *htd* 1.0.1 [Abs16] to the other participants of Track A ("Treewidth") of the "First Parameterized Algorithms and Computational Experiments Challenge" (PACE 2016, see `https://pacechallenge.wordpress.com/track-a-treewidth/`).

The experiments are based on the following algorithms submitted to PACE 2016:

- 1: "tw-heuristic"

  Available at `https://github.com/mrprajesh/pacechallenge`.

  GitHub-Commit-ID: `6c29c143d72856f649de99846e91de185f78c15f`

- 5 (*htd*): "htd_gr2td_exhaustive.sh"

  Available at `https://github.com/mabseher/htd`

  GitHub-Commit-ID: `fc8df04fc433f22f49d15900ac139b22754458fe`

- 6: "tw-heuristic"
  Available at `https://github.com/maxbannach/Jdrasil`
  GitHub-Commit-ID: `fa7855e4c9f33163606a0677485a9e51d26d7b0a`
- 9: "tw-heuristic"
  Available at `https://github.com/elitheeli/2016-pace-challenge`
  GitHub-Commit-ID: `2f4acb30b5c48608859ff27b5f4e217ee8346ca5`
- 10: "tw-heuristic" [GGJ+16]
  Available at `https://github.com/mfjones/pace2016`
  GitHub-Commit-ID: `2b7f289e4d182799803a014d0ee1d76a4de70c1f`
- 12: "flow_cutter_pace16" [HS16]
  Available at `https://github.com/ben-strasser/flow-cutter-pace16`
  GitHub-Commit-ID: `73df7b545f694922dcb873609ae2759568b36f9f`

The list of algorithms contains all participants of the sequential heuristics track of the PACE treewidth challenge in the variant in which they were submitted to the challenge plus the most recent release of *htd*. For each of the algorithms we provide its ID that was used in the challenge (and also in our experiments), the name of the binary, the location of its source code and the exact identifier of the program version in the GitHub repository. Note that for *htd* (ID 5), four different configurations exist. In our experiments here we only consider the best-performing variant, namely "htd_gr2td_exhaustive.sh". Its implementation first computes an upper bound for the width using the Minimum Degree heuristic [BHS03] and then iteratively calls the Min-Fill algorithm [Dec03] to improve the width of the obtained tree decomposition using the class `htd::WidthMinimizingTreeDecompositionAlgorithm` which is a specialization of the class `htd::IterativeImprovementTreeDecompositionAlgorithm`.

The first evaluation is done based on the public data set of the PACE challenge 2017 which is available for download at `https://people.mmci.uni-saarland.de/~hdell/pace17/he-instances-PACE2017-public-2016-12-02.tar.bz2`. It contains 100 graph instances originating from real-world scenarios.

The second evaluation uses the first data set of the QBFEval 2016 competition of solvers for quantified boolean formulae (QBF), available at `http://www.qbflib.org/TS2016/Dataset_1.tar.gz`. We note that the 825 instances of this second data set had to be converted in order to comply to the input format of the PACE competition. This was done in the following way: For each QBF we ignore the quantifier information and we introduce a clique between the vertices of each clause. That is, we consider the clauses in the QBF as hyperedges and work on the primal graph of the given QBF.

All our experiments were performed on a single core of an Intel Xeon E5-2637@3.5GHz processor running Debian GNU/Linux 8.7 and each test run was limited to a runtime

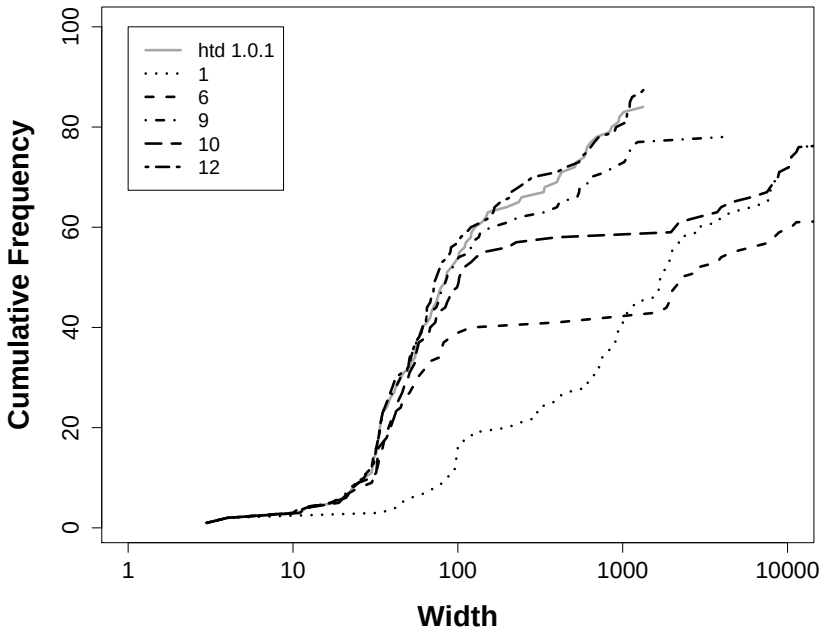


Figure 4.4: Comparison of Tree Decomposition Algorithms (Data Set "PACE 2017")

of at most 100 seconds and 32 gigabyte of main memory. For the actual evaluation we use the testbed of the PACE challenge which is available at `https://github.com/holgerdell/PACE-treewidth-testbed`. For the repeatability of the experiments we provide the complete testbed and the results for each data set and each algorithm under the following link: `http://dbai.tuwien.ac.at/research/project/decodyn/htd/evaluation_htd101.zip`

Figure 4.4 summarizes the outcome of our first evaluation. The plot is constructed by running *htd* and each of the five other algorithms of the competition on each instance contained in the public data set of the PACE challenge 2017. Afterwards, we plot the cumulative frequency of the obtained width after 100 seconds.

The solid gray line indicates the quality in terms of the width of the decompositions computed by *htd* and the dotted as well as the dashed lines illustrate the width achieved by its competitors. A point $p = (x, y)$ on a line in the figures represents the fact that y instances could be successfully decomposed within the time limit of 100 seconds and each decomposition had a width not higher than x. Hence, it is good to minimize x with respect to y, that is, the optimal algorithm reaches a certain point on the y-axis not exceeding the widths of its competitors on the x-axis.

In Figure 4.4, we can see that *htd* 1.0.1 is among the best two algorithms to decompose the graphs from the public data set of the PACE challenge 2017 when considering the width only. This indicates that *htd* is well suited for computing decompositions of small width on the given instances. We can also see that currently no algorithm manages to

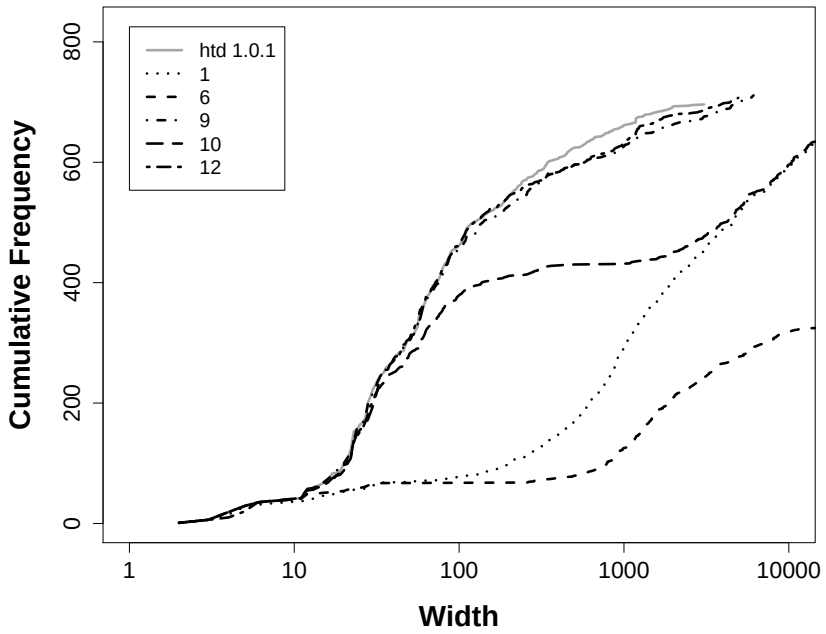


Figure 4.5: Comparison of Tree Decomposition Algorithms (Data Set "QBFEval 2016")

decompose all of the hundred instances as they are much larger than those used in the PACE challenge 2016. Obtaining a decomposition for an instance of the first PACE challenge was in almost all cases just a matter of a few seconds when *htd* is employed as the decomposition library of choice.

When we look at the QBF data set, illustrated in Figure 4.5, we can see that about 100 instances cannot be decomposed by any of the algorithms and that two algorithms (9 and 12) are actually able to decompose slightly more instances than *htd*. In the fragment of the figure around width 1000 we observe that *htd* performs significantly better than the other algorithms. When the width gets closer to 5000 and also in the case of higher widths, two competitors (9 and 12) can decompose slightly more instances.

At this point, it has to be noted that *htd* per default delivers tree decompositions in which only subset maximal bags are retained. Enforcing this constraint can deteriorate the performance in cases in which tree decompositions have a high width because this involves checking for subset inclusion and this check is clearly more expensive for large bags than for small ones. Nevertheless, performing this cleaning-up operation can help to improve the efficiency of the dynamic programming algorithm which uses the resulting decomposition. Furthermore, the system which could solve the most instances according to our benchmark results employs several preprocessing steps in order to significantly improve the efficiency of the employed algorithm (see [HS16]). In contrast to this, the *htd* framework currently does not perform any preprocessing of the input instance so that there is still room for "cheap" improvements in terms of both decomposition width and performance in future releases of the *htd* framework.

4.6 Discussion

In this chapter we presented a free, open-source C++ framework for graph decompositions. We showed the most important features, gave an introduction on how to use the library and highlighted issues we faced during the implementation phase and provided insights on how we coped with them. Furthermore, we evaluated our approach by comparing our library to the participants of the "First Parameterized Algorithms and Computational Experiments Challenge". The outcome of the evaluation indicates that the performance characteristics of the new framework are very promising.

For future work, we clearly want to further improve the built-in heuristics and algorithms in order to enhance the capabilities for the generation of decompositions of small width. Furthermore we are currently working on refining the algorithms allowing to automate the process of computing customized tree decompositions.

Last, but not least, we invite researchers and software developers to contribute to the library as we try to initiate a joint collaboration on a powerful framework for graph decompositions and any input is highly appreciated.

CHAPTER 5

Exploiting Customized Tree Decompositions

After demonstrating the usefulness of discriminating tree decompositions by additional criteria than just the plain width in Chapter 3, in this chapter we will now investigate how we can concretely improve the efficiency of dynamic programming algorithms by using customized tree decompositions as provided by *htd*. This analysis is based on two different case studies. The first one uses a state-of-the-art system specialized on the evaluation of so-called *quantified boolean formulae* (QBF) and the second one employs the general-purpose framework *D-FLAT* which is also used in the experiments presented in Section 3.2.

For repeatability of the experiments, all results, the exact configurations of the systems we used in the evaluation as well as protocols of all tests runs are provided together with a copy of the system binaries at `http://dbai.tuwien.ac.at/proj/decodyn/htd/customized_decompositions.zip`.

5.1 Case Study: *dynQBF*

The first scenario we want to have a look at is the evaluation of quantified boolean formulae using the *dynQBF* system [CW16a, CW16b]. Before we move on to the details of the evaluation, we first want to provide a formal definition of quantified boolean formulae and introduce *dynQBF*.

5.1.1 Preliminaries

Basically, QBFs are a generalization of classical propositional formulae. As the name already suggests, in the case of QBFs, one is allowed to use quantifiers in addition to variables, boolean connectives and related concepts already known from classical

propositional formulae. Most often, QBFs are provided in Prenex-CNF, i.e., the group of quantifiers precedes the formula and the propositional part of it is in conjunctive normal form.

Definition 13 (Quantified Boolean Formulae in Prenex-CNF)**.** *A* quantified boolean formula in Prenex-CNF *(PCNF QBF) is a boolean formula of the form $Q.\phi$ where Q denotes the so-called* quantifier prefix *and ϕ is a propositional formula in CNF. The quantifier prefix Q is a sequence $Q_1V_1, \ldots, Q_nV_n$ where $Q_i \in \{\exists, \forall\}$ and the sets V_i represent a partition of the atoms in ϕ. Furthermore, the quantifiers alternate between different positions, i.e., $Q_{i+1} \neq Q_i$ for all positions $1 \leq i < n$.*

Note that Definition 13 actually refers to so-called *closed QBFs*, i.e., QBFs in which all occurring variables are bound by a quantifier. In general, it is also allowed that certain atoms do not occur in the context of a quantifier. In this case, the corresponding variables are called *free* or *unquantified*.

Given that $\mathcal{L}_{QBF}$ denotes the language of quantified boolean formulae and that $\phi[\top/v]$ as well as $\phi[\bot/v]$ stand for the replacement of the variable v by a truth constant (either $\top$ or $\bot$) in the formula ϕ, based on the valuation function $\mathcal{I} : \mathcal{L}_{QBF} \to \{\textbf{True}, \textbf{False}\}$ the semantics of QBFS are defined as follows:

- $\mathcal{I}(\top) = \textbf{True}$
- $\mathcal{I}(\bot) = \textbf{False}$
- $\mathcal{I}(\neg\phi) = \textbf{True}$ if and only if $\mathcal{I}(\phi) = \textbf{False}$
- $\mathcal{I}(\phi \vee \psi) = \textbf{True}$ if and only if $\mathcal{I}(\phi) = \textbf{True}$ or $\mathcal{I}(\psi) = \textbf{True}$
- $\mathcal{I}(\phi \wedge \psi) = \textbf{True}$ if and only if $\mathcal{I}(\phi) = \textbf{True}$ and $\mathcal{I}(\psi) = \textbf{True}$
- $\mathcal{I}(\exists v\phi) = \textbf{True}$ if and only if $\mathcal{I}(\phi[\top/v]) = \textbf{True}$ or $\mathcal{I}(\phi[\bot/v]) = \textbf{True}$
- $\mathcal{I}(\forall v\phi) = \textbf{True}$ if and only if $\mathcal{I}(\phi[\top/v]) = \textbf{True}$ and $\mathcal{I}(\phi[\bot/v]) = \textbf{True}$

A closed QBF $Q.\phi$ is said to be satisfiable if its evaluation under the semantic rules from above yields the result **True**. In the case of QBFs with free variables, the satisfiability furthermore depends on the actual truth assignment to the free variables.

The decision problem corresponding to quantified boolean formulae is called Quantified Boolean Satisfiability (QSAT). Similarly to the classical SAT problem, the QSAT problem is defined as follows:

Input: A quantified boolean formula $Q.\phi$
Question: Is $Q.\phi$ satisfiable?

A trivially satisfiable QBF in Prenex-CNF is $Q.\phi$ with $Q = \exists xy \forall z$ and $\phi = [(x \vee z) \wedge (y \vee z)]$. The satisfiability of this QBF is obvious because the formula ϕ is satisfied when we set the variables x and y to **True**, regardless of the actual truth value we assign to the variable z. Although the effort for showing the satisfiability of our example QBF is negligible, the QSAT problem is known to be PSPACE-complete [SM73]. At this point, it has to be noted that it is widely assumed that the complexity class PSPACE subsumes the complete polynomial hierarchy.

While the general QSAT problem is PSPACE-complete, the actual complexity of a given problem instance strongly depends on the quantifier prefix. This dependency is exploited in complexity theory due to the fact that it allows to define a prototypical problem for each level of the polynomial hierarchy (see, e.g., [GJ79, Pap94] for more details on the polynomial hierarchy). For instance, in those cases where the quantifier prefix consists only of a single existential quantifier, we basically deal with an instance of the SAT problem, the prototypical example of an NP-complete problem.

Quantified boolean formulae are used, for instance, in model checking [DHK05], formal verification [BM08] and synthesis [ZMMN12]. In order to stimulate the development of efficient QBF solvers, the "Quantified Boolean Formulas Satisfiability Library" [GNPT05] (QBFLIB) was initiated in 2004. To keep track of the advances in solver performance, in connection to QBFLIB, also a competition of QBF solvers named QBFEval [NPT06] is organized every few years. In the latest competition, QBFEval 2016 [Pul16], more than 20 solvers participated. Without going into the details of the other systems, among the competitors of QBFEval 2016 there is one candidate which is special in the sense that its approach is based on tree decompositions. Hence, it is a promising candidate to evaluate the impact of tree decomposition customization in a practical setting.

The name of this system is *dynQBF* [CW16a, CW16b] and it is available as free and open-source software at `http://dbai.tuwien.ac.at/research/project/decodyn/dynqbf/`. Due to the fact that *dynQBF* follows the concept of dynamic programming on tree decompositions, its general workflow is similar to that of the *D-FLAT* framework we already used in the previous chapters: At first, a QBF in Prenex-CNF is parsed and a tree decomposition of the propositional part of the formula is computed. Afterwards, to decide the satisfiability of the given QSAT instance, the decomposition is traversed in a bottom-up manner while maintaining the partial solutions according to the dynamic programming algorithm. The quantifier prefix is thereby handled internally, i.e., it is not part of the decomposition. In order to boost performance and to circumvent problems with extensive space consumption, *dynQBF* does not materialize all intermediate solutions, like *D-FLAT*. Instead, so-called *binary decision diagrams* (BDDs) [Lee59, Ake78, DS01] are used to succinctly represent partial solutions.

As a small side-remark, we want to highlight that the *dynQBF* system employs *htd* as decomposition library and that it makes intensive use of many convenience features provided by the *htd* framework, like custom labels or the efficient computation of induced edges. For instance, the information about which variables occur positively or negatively in a clause of a given QBF is stored by means of (hyper-)edge labels of the formula's

hypergraph representation (see Figure 4.3). In combination with the efficiently computed set of induced edges for each bag of the tree decomposition this allows for a rapid lookup of those pieces of information which are needed in a dynamic programming step.

5.1.2 Experimental Setup

The experiments with the *dynQBF* system are performed on a single core of an Intel® Xeon E5-2637@3.5GHz processor with Hyperthreading and Intel® Turbo Boost being disabled. The benchmark machine employed in the evaluation has access to 8x32 GB of main memory (DDR4@2133MHz, ECC, registered, CL15) and as operating system Debian GNU/Linux 8.7 (3.16.0-4-amd64) was used. Each of the test runs was limited to a runtime of at most ten minutes and 32 GB of main memory.

For the following experiments we consider the 305 instances of the 2QBF track of the QBFEval 2016 competition which is available at `http://www.qbflib.org/TS2016/Dataset_3.tar.gz`. The actual binary of *dynQBF* we use in our experiments can be downloaded at `https://github.com/gcharwat/dynqbf/releases/download/v0.4.1/dynqbf-v0.4.1-x86_64-static.zip`. This archive comprises a ready-to-run version of *dynQBF* 0.4.1[1] which internally uses *htd* 1.0.1[2].

In the QBFEval challenge, the primary goal for the participants is to solve as much QBFs as possible within a given time limit. Usually, the time allowed for deciding the satisfiability of a 2QBF instance is limited to ten minutes in the challenge. If two systems are able to solve the same number of instances, the total solving time over all solved QBFs is considered as additional performance criterion. Hence, the winner of the challenge is not only determined by the number of solved instances, but also the total solving time is important.

As we have seen that the robustness of dynamic programming algorithms using tree decompositions can be problematic, it seems to be promising to find a customization which is beneficial for the runtime behavior of those algorithms. In particular, we want to avoid negative statistical outliers, i.e., tree decompositions which lead to extremely high running times whereas the majority of tree decompositions for the very same instance allows for significantly better solving times.

For this reason, in the following experiments we compare two different optimization strategies for tree decompositions provided by *dynQBF*, namely *width* and *join-child-bag-product*. The former aims for using decompositions of minimum width and the latter tries to reduce the time the dynamic programming algorithm spends in join nodes. We quantify this effort for a given (rooted) tree decomposition $(\mathcal{T}, \chi)$ with $\mathcal{T} = (N, E_{\mathcal{T}})$ by the estimation function $f(\mathcal{T}, \chi) = \sum_{j \in \mathit{Join}(\mathcal{T})} \prod_{c \in \mathit{Children}(\mathcal{T}, j)} |\chi(c)|$ where $\mathit{Join}(\mathcal{T})$ represents the set of join nodes in $\mathcal{T}$ and $\mathit{Children}(\mathcal{T}, j)$ denotes the set of children of node j in the rooted tree $\mathcal{T}$. At this point it is important to note that *dynQBF* works

[1] Available at `https://github.com/gcharwat/dynqbf/releases/v0.4.1`
[2] Available at `https://github.com/mabseher/htd/releases/1.0.1`

on non-normalized tree decompositions, i.e., the bag contents of different children of the same join node are not identical. The intuition behind the aforementioned formula f is to capture the worst-case effort for the merging of information in join nodes as we assume that propagating and updating information in the remaining nodes can usually be done quite efficiently based on *dynQBF*'s sophisticated algorithms based on binary decision diagrams.

Using the *width* strategy can be enforced in *dynQBF* 0.4.1 by specifying the program option `--ds width`. Analogously, *dynQBF* will follow the *join-child-bag-product* strategy in presence of the program option `--ds join-child-bag-prod`. In *dynQBF* 0.4.1, this second strategy is also the default, i.e., it is used in those cases where no decomposition strategy is selected explicitly. The number of optimization iterations to be used for a strategy are provided via the program option `--dsi`. A full program call for deciding the satisfiability of an instance `instance.qdimacs` may look as follows:

```
./dynqbf --ds width --dsi 50 < instance.qdimacs
```

The above program call triggers *dynQBF* to solve the given QSAT instance based on the *width* strategy. In this example, the tree decomposition of minimal width among 50 heuristically generated ones (computed via the Min-Fill heuristic) will be used for the dynamic programming algorithm.

To rule out bias, each QBF is solved ten times using different random seeds. Based on the fact that the implementation of the Min-Fill heuristic provided by *htd* breaks ties randomly, this leads to ten different tree decompositions. Apart from ruling out bias, repeated test runs with changing tree decompositions also allow us to investigate the difference between "average" decompositions and "bad" ones.

5.1.3 Results

Before we move on to the actual performance evaluation, let us first investigate whether the two strategies make a difference with respect to the width of the tree decompositions which are used.

In Figure 5.1 we can see that this is indeed the case. This plot depicts the median of the maximum bag sizes over 10 different random seeds for the Min-Fill heuristic. The solid line illustrates the outcome for test runs where the first decomposition returned by the Min-Fill heuristic is used. The dot-and-dash line represents the outcome when we use the tree decomposition of minimum width among 50 iteratively generated ones and the dotted line is used to show the outcome when the tree decomposition is used which minimizes our estimation function f from above in a pool of 50 generated decompositions. The whole figure is based on 113 QBFs which could be solved in all of the three cases based on each of the 10 random seeds. We can see that, although the three lines are often relatively close together, aiming for a low "complexity" of join nodes may increase the width significantly (in extreme cases by up to 13 points in our experiments).

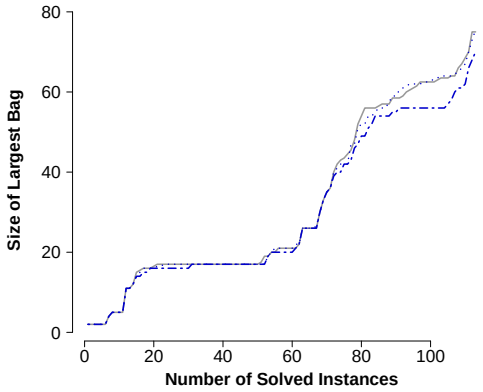


Figure 5.1: Maximum Bag Size of Tree Decompositions over 10 Random Seeds

From a theoretical point of view, we would assume that a lower width in general has a positive impact on the solving time and therefore the number of solved instances should be higher when we use a larger pool of decompositions to choose from. That this is not necessarily the case is illustrated in Figure 5.2. This figure shows the number of solved instances for the two strategies depending on the number of computed tree decompositions from which the best one (with respect to the selected strategy) is chosen for the dynamic programming algorithm. In the figure, the letter 'W' stands for the *width* strategy and the letter 'J' corresponds to the *join-child-bag-product* strategy. Each box-plot in the figure is constructed from ten different measurements about the number of solved instances, one for each random seed.

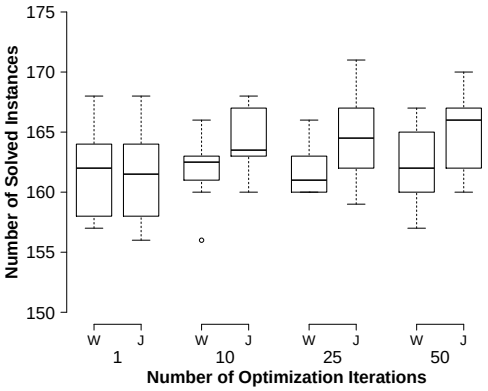
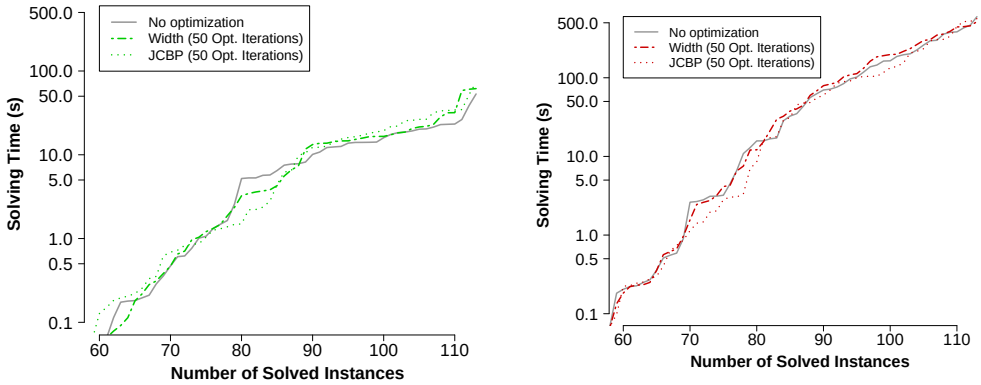


Figure 5.2: Distribution of Solved QBF Instances over 10 Random Seeds

Figure 5.2 shows that when we prefer tree decompositions of small width, the number of solved instances does not change much although the widths are indeed smaller in many cases compared to just taking the first tree decomposition returned by Min-Fill. In contrast, when we look at the box-plots which are based on tree decompositions which are customized towards low join node complexity, we observe the trend that the more

iterations we spend on finding a beneficial tree decomposition, the more instances we can solve. Without optimization, *dynQBF* can solve 162 out of 305 QBFs in the average case. With ten optimization iterations for the *join-child-bag-product* strategy this number increases to 163.5, with 25 iterations it grows to 164.5 and with 50 iterations we can solve 166 instances. At this point we want to highlight that the time needed to compute the respective number of decompositions indeed is counted for the timeout limit of ten minutes. This means that the four instances we can solve additionally using customized tree decompositions are by no means hypothetic. Instead, they can be crucial in both competition and practice.

Note that the same ten random seeds are used for each QBF. Therefore, the first two box-plots in Figure 5.2, depicting the cases in which the first decomposition returned by the Min-Fill heuristic is used regardless of the optimization strategy, are indeed based on the very same tree decompositions. The small difference in the number of solved instances is caused by border cases for which the solving time is close to the timeout limit. Due to inevitable system interrupts (like, e.g., cache misses, context switches or other kernel tasks) it can happen that the time for performing the very same operations varies enough that one time an instance is solved whereas a repeated experiment with the prerequisites exceeds the timeout limit by some seconds.



(a) Median Runtime

(b) Maximum Runtime

Figure 5.3: Solving Times for QBF Instances over 10 Random Seeds

What is left is to investigate the impact of the two customization strategies on runtime behavior of *dynQBF*. This information is provided in Figure 5.3. The chart in Figure 5.3a depicts the distribution of the median runtime over ten random seeds for the 113 QBFs which were also used to construct Figure 5.1. Analogously, Figure 5.3b illustrates the distribution of the maximum runtime over ten random seeds. For the sake of readability, instances with a solving time under a tenth of a second are omitted in the figure. This time, as we are solely interested in the effect of tree decomposition customization on the net solving time, the depicted times refer to the time spent in the dynamic programming algorithm, i.e., the time needed to compute 50 decompositions is not considered.

An interesting observation is that both strategies do not necessarily lead to lower solving times compared to the first decomposition when we consider the median case. For the *width* strategy this can be explained by the fact that the Min-Fill algorithm often delivers tree decomposition of rather low width and so the chances for finding a decomposition of smaller width are reduced. Furthermore, a low width often comes along with a higher number of nodes and this may cause increased effort in the context of the dynamic programming algorithm. One possible reason for the circumstance that the *join-child-bag-product* strategy does not show a strictly positive impact on the median runtime is the fact that the data structures used by *dynQBF* are of course geared towards efficiency and so the already relatively fast computations in the average case are not so easy to accelerate.

Instead, the strengths of the *join-child-bag-product* strategy become apparent when we look at the worst-case running times depicted in Figure 5.3b. As noted in Section 5.1.2, it is important for dynamic programming algorithms using tree decompositions to narrow down the gap between average and worst-case running time and this is exactly what we achieve with a reduction of the join node complexity. While the worst-case solving time for the *width* strategy, using the tree decomposition of minimum width from a pool of 50 iteratively generated ones, is rarely below the solving time using the first decomposition delivered by Min-Fill in our experiments, the *join-child-bag-product* strategy increases the robustness of *dynQBF* notably.

In the context of this case study we also performed additional experiments with other decomposition strategies. *dynQBF* 0.4.1 offers ten different ones to choose from, including the reduction of the total number of join nodes or minimizing the worst-case size of the data structure employed for storing the intermediate solutions. Although all strategies are based on natural assumptions regarding the impact of the shape of the given tree decomposition on the runtime of *dynQBF*, none of them performed as good as the *join-child-bag-product* strategy we present here. This underlines that finding appropriate strategies for computing customized tree decompositions requires careful engineering due to the fact that the final outcome has to capture all the specifics of the dynamic programming algorithm. In that sense, the development of strategies is closely related to the also quite complex task of feature engineering in the context of machine learning (see Section 2.4 for a short overview of this topic).

5.2 Case Study: *D-FLAT*

The second scenario we want to investigate is solving the problem of Steiner Tree using the *D-FLAT* framework. The following experiments are closely related to those presented in Section 3.2.3, but this time we no longer need to perform complex and potentially time-consuming post-processing steps. This is because we have with *htd* a dedicated decomposition framework capable of delivering customized tree decompositions directly. By providing this experimental evaluation, we want to show the effect of combining the main ideas we formulated in Chapter 3 in a practical system.

At this point, we want to highlight that the goal of the following experiments is not to outperform dedicated systems or specialized dynamic programming algorithms for the Steiner Tree problem but to increase the efficiency and robustness of solving this problem by means of the general-purpose framework *D-FLAT*.

5.2.1 Experimental Setup

The experiments with the *D-FLAT* framework are performed on the same system as those from the first case study (see Section 5.1.2 for details). Only the resource limits differ. Here, each of the test runs is limited to a runtime of at most six hours and 32 GB of main memory.

As benchmark instances we consider the five metro and interurban train networks which were also used in Section 3.2.3 (Tokyo, Singapore, Santiago, Osaka, Vienna) plus the metro system of London due to the fact that the current version of *D-FLAT* is efficient enough to handle also this rather complex metro network. For each city we consider 50 instances of the Steiner Tree problem which are constructed by randomly selecting ten of the metro (or interurban train) stations as terminal vertices. Also in this case study, each problem instance is solved ten times based on ten different tree decompositions.

The experiments contributing to this case study are performed using the software binary available for download at `https://github.com/bbliem/dflat/releases/download/v1.2.4/dflat-1.2.4-x86_64.tar.gz`. The archive comprises a ready-to-run version of *D-FLAT* 1.2.4[3] which internally uses *htd* 1.0.1[4].

Also in this second case study we compare two different optimization strategies for tree decompositions. Apart from *width* we consider this time the strategy *average-join-bag-size* whose idea is the exactly same as in the first case study. Due to the fact that the dynamic programming algorithm we use here for solving the Steiner Tree problem works on semi-normalized tree decomposition, i.e., tree decompositions where the bags of join nodes and their children's bags coincide, we can now use a simpler estimation function to capture the "complexity" of join nodes. The *average-join-bag-size* strategy aims for finding a rooted tree decomposition $(\mathcal{T}, \chi)$ which either has no join node or which minimizes the outcome of the formula $f(\mathcal{T}, \chi) = \sum_{j \in Join(\mathcal{T})} |\chi(j)| / |Join(\mathcal{T})|$, where $Join(\mathcal{T})$ represents the set of join nodes in $\mathcal{T}$. Note that the formula $f(\mathcal{T}, \chi)$ computes the mean of the bag sizes of join nodes because a simple sum of all the bag sizes might prefer a single, huge join node over several small ones although the latter case is probably more beneficial for the efficiency of dynamic programming algorithms.

For choosing the decomposition strategy for an input instance, *D-FLAT* 1.2.4 offers the program option `--fitness`. Setting the corresponding parameter value to `width` (`join-bag-avg`) allows to select the *width* (*average-join-bag-size*) strategy. The number of optimization iterations to be used for a strategy are provided via the program option

[3] Available at `https://github.com/bbliem/dflat/releases/v1.2.4`
[4] Available at `https://github.com/mabseher/htd/releases/1.0.1`

`--iterations`. Given that the file `encoding.lp` contains a proper *D-FLAT* encoding for the Steiner Tree problem, a full program call for enumerating all solutions of a given problem instance `instance.lp` may look as follows:

```
./dflat -p encoding.lp --fitness join-bag-avg --iterations 50
                        < instance.lp
```

The above program call forces *D-FLAT* to solve the given instance based on the *average-join-bag-size* strategy. In this example, the dynamic programming algorithm will utilize the tree decomposition for which the average bag size of join nodes (defaulting to 0 for tree decomposition without join nodes) is minimal among a pool of 50 heuristically generated ones based on the Min-Fill heuristic.

5.2.2 Results

The first thing we want to have a look at in the experiments belonging to this case study is the actual distribution of solving times for each of the cities. To that end, we provide in Figure 5.4 a box-plot of the solving times for each of the six cities and both strategies, *width* (W) and *average-join-bag-size* (J), over the complete set of 50 problem instances and all 10 random seeds. Hence, each box-plot is constructed from 500 measurements of the total solving time (including the time for computing the decompositions). Here, a measurement for a given problem instance, seed and strategy is obtained by running the dynamic programming algorithm on the best tree decomposition found with respect to the given strategy after 50 optimization iterations.

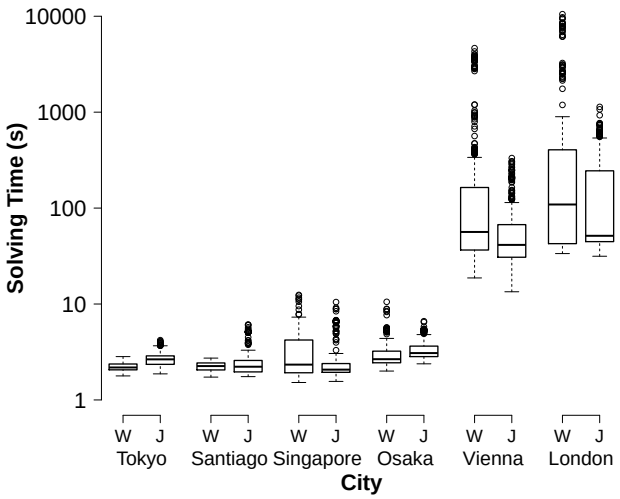


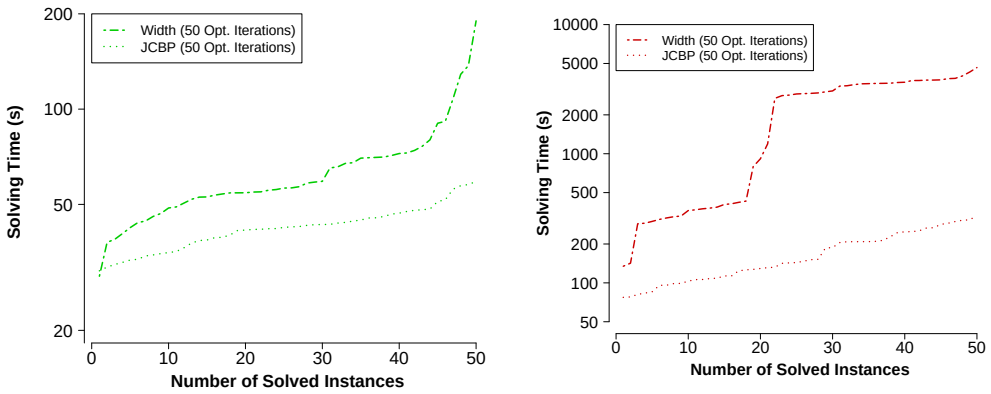
Figure 5.4: Distribution of Solving Times for Steiner Tree

Note at this point that the box-plots depicted in Figure 5.4 mix the measurements from different instances although some of them are probably harder than others. For instance,

when the terminals are far apart, then connecting these stations is likely more involved than in the case where the terminals are close together. Nevertheless, Figure 5.4 allows us to get an idea of the performance characteristics of *D-FLAT* in relation to the actual strategy used.

For the cities Tokyo, Santiago, Singapore and Osaka the picture is not significant due to the very low time consumption of the current version of *D-FLAT* in these cases. When we look at the metro and interurban train network of Vienna or the metro network of London, the positive effect of the *average-join-bag-size* strategy on the solving time becomes obvious. Especially the worst-case running times achieved on the basis of the *average-join-bag-size* strategy are by magnitudes lower than what we observe when we solely aim for a low width. But not only the worst-case solving times are significantly reduced, also the efficiency in the average case (depicted in the box-plots by the median line) improves notably.

In the rest of the evaluation we will concentrate on the two "interesting" public transport networks Vienna and London, because in these cases the difference between the two strategies in terms of solving time is significant and the savings one can achieve are practically relevant.



(a) Median Runtime

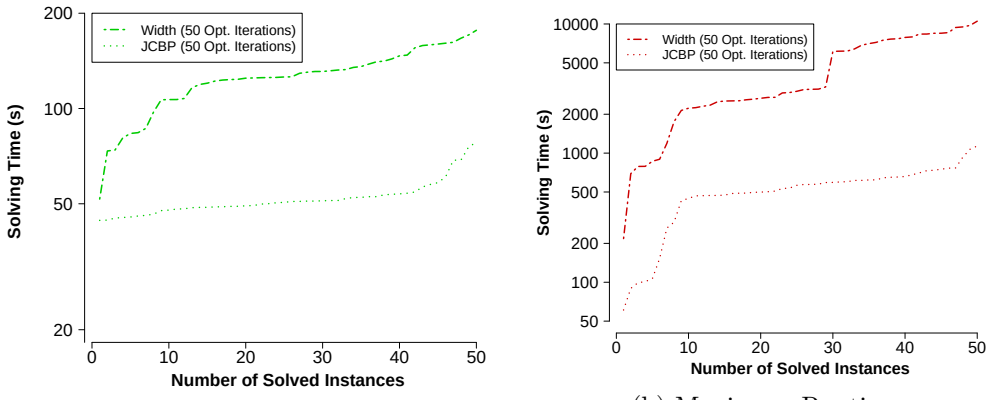
(b) Maximum Runtime

Figure 5.5: Solving Times for Steiner Tree (Vienna) over 10 Random Seeds

Figure 5.5 illustrates the detailed distribution of solving times depending on the actual strategy being used. Note that, in this case study, we omit a separate investigation of the efficiency without optimization, because, for the instances used here, the width of the first decomposition delivered by the Min-Fill heuristic is in most cases not worse than the width after 50 optimization operations with strategy *width* (see also comparison of minimum and median width in Table 3.2). Therefore, the solving times between no optimization and optimizing towards low width in almost any case coincide due to the fact that *htd* 1.0.1 sticks with the first decomposition as long as no strictly better one with respect to the selected strategy is found. In contrast to the previous case study, this time the solving times on which Figure 5.5 is based include the time needed to

compute 50 decompositions of a given instance. This is possible without influencing the interpretation because all problem instances contributing to the figure are based on the very same public transport network and so we can expect similar decomposition times for each of these instances.

Figure 5.5a confirms the observations we already made in context of Figure 5.4. We can see that there are several instances which can be solved by either strategy in less than a minute in the average case. Nevertheless, even for the "easy" instances, the strategy *average-join-bag-size* is highly favorable compared to just relying on the width. Moreover, considering the optimization towards low join node complexity becomes even more important when we look at the maximum runtime over 10 random seeds, depicted in Figure 5.5b. Here, the avoidance of join nodes with large bags can make a difference of hours. Furthermore, we can see that both the median and the maximum solving time are much better in this case and so we are approaching our goal to improve both efficiency and robustness of the *D-FLAT* system.



(a) Median Runtime

(b) Maximum Runtime

Figure 5.6: Solving Times for Steiner Tree (London) over 10 Random Seeds

Finally, let us have a look at Figure 5.6. Also here, the picture is rather clear as we observe a significant positive effect of the *average-join-bag-size* strategy on the total solving time. When comparing Figures 5.5 and 5.6, we can see that solving the problem of Steiner Tree seems to be more involved for the metro network of London than in the case of Vienna although the decompositions have the same width in almost all cases, namely 5. This observation is not only apparent for the median runtime over ten random seeds but also in the worst case over ten different decompositions. A possible explanation for the difference in the solving times is the different number of stations (Vienna: 137, London: 285). Nevertheless, the *average-join-bag-size* strategy allows us to keep the solving time around a minute in the median case over ten decompositions while when we concentrate solely on the width we have to spend much more time. Again, in the worst case, the solving times using the *average-join-bag-size* strategy are by magnitudes lower than when we just use a decomposition of low width.

Also in the context of this case study, other decomposition strategies were considered. In addition to *width* and *average-join-bag-size*, *D-FLAT* 1.2.4 also offers the following two strategies. The first one, *num-joins*, prefers decompositions in which the number of join nodes is low and the second alternative is the *median-join-bag-size* strategy which tries to minimize the median over all join bag sizes. While the *num-joins* strategy on average performs as good (or as bad) as choosing an arbitrary decomposition of low width, the picture of the *median-join-bag-size* strategy is more interesting. Here, we observed that in some situations the solving times are relatively close to those we achieve based on *average-join-bag-size*. Unfortunately, sometimes we also observe a significant deteriorations of *D-FLAT*'s performance when using the *median-join-bag-size* strategy. An in-depth investigation of these cases confirmed the assumption that the performance characteristics of the *median-join-bag-size* strategy heavily depend on how close the median of the join node bag sizes is to the mean of these bag sizes. In particular, the positive effect of using the *median-join-bag-size* strategy vanishes (or even turns into a negative effect) if the tree decomposition which is selected by the strategy contains a join node of large bag size whereas the "average" tree decomposition delivered by the Min-Fill heuristic may contain a higher number of join nodes of moderate bag size (thus potentially increasing the median) but no join node with large bags. This is by no means surprising when we recall the fact that *D-FLAT* per default always computes all solutions for a given problem instance and so it is natural that especially join nodes with large bags are often harmful for the overall performance.

5.3 Discussion

In the previous sections we presented two cases studies in which we compared the solving times we obtained following two strategies offered by *htd*. The first approach was based on the assumption that minimizing the width of a tree decomposition is enough to obtain good performance and it acted as benchmark for the other strategy. This second approach was based on our experience that join nodes with large bags or many children are often harmful for the performance of dynamic programming algorithms.

Our experiments show that using the "complexity" of join nodes as an additional criterion to judge the quality of tree decompositions can have a remarkably positive effect on the general runtime behavior of dynamic programming algorithms. This underlines that it may well pay off in practice to consider using customized tree decompositions as provided by the *htd* framework.

Nevertheless, we note that the width is still a very important criterion as it gives us an upper bound for the bag sizes of a given tree decomposition and these bag sizes indeed have a crucial impact on the efficiency of dynamic programming algorithms. Therefore, it is probably a bad idea to completely ignore the width as a feature. In our experiments, we used the Min-Fill heuristic which is known to give good results with respect to the width as base decomposition algorithm. Hence, the tree decompositions we used here always have a rather low width, independently of the selected optimization strategy.

CHAPTER 6

Related Work

As mentioned already in the introduction of this thesis, there are various scenarios in which tree decompositions are applied with great success. Apart from their extensive use in different dynamic programming algorithms for solving computationally hard problems, tree decompositions play an important role in the area of machine learning. In particular, tree decompositions are a vital ingredient of so-called *junction tree algorithms*[1] which are heavily employed in the context of probabilistic inference in Bayesian (belief) networks (see, e.g., [HD96] for an overview of this topic). The application areas of Bayesian networks are manifold and they are used in a variety of expert systems (see, e.g., [PNM08] for more details). Among the different realizations of junction tree algorithms we find, for instance, approaches by Lauritzen and Spiegelhalter [LS88], by Shenoy and Shafer [SS08] and by Jensen et al. [JOA90, JLO90]. The latter is also known in the literature as the *HUGIN* architecture due to its use in the *HUGIN* framework [AOJJ89]. A detailed comparison of these three approaches for probabilistic inference is provided in [LS98].

Apart from tree decompositions, also the closely related concept of so-called hypertree decompositions [GLS02] is used successfully in several application scenarios. Basically, a hypertree decomposition of a hypergraph $\mathcal{H} = (V, H)$ is a tree decomposition of $\mathcal{H}$ where each node of the tree has assigned a subset of the hypergraph's hyperedges in addition to the set of vertices constituting the node's bag. The hyperedges associated with a node are selected in such a way that the set union of the endpoints of these hyperedges is a superset of the node's bag. Analogously to the width of a tree decomposition (see Definition 10), the width of a hypertree decomposition is defined as the maximum cardinality across all hyperedge sets associated to the nodes of the respective decomposition. Based on this notion, the *hypertree-width* of a hypergraph is the minimum width over all its possible tree decompositions.

[1] In the machine learning literature, tree decompositions are often also called *join trees*, *clique trees* or *junction trees*.

A very recent publication which gives a detailed overview of hypertree decompositions and their applications in the context of conjunctive query answering is by Gottlob et al. [GGLS16]. Conjunctive queries represent a restricted form of first-order queries and they are a central concept in database theory because of the fact that many queries to relational database management systems can be expressed by means of this type of queries. A special case of conjunctive queries are boolean conjunctive queries. Their result is either **True** or **False**, depending on whether the relations in the given database satisfy the provided conjunction. In general, evaluating a conjunctive query is NP-complete (when considering both the query and the database as input) and this also holds for the boolean case [CM77]. Hypertree decomposition come into play in this context as it can be shown that, given a boolean conjunctive query Q, a database DB and a hypertree decomposition of Q of bounded width, deciding whether the query Q evaluates to **True** on the database DB is LOGCFL-complete [GLS02], i.e., it can be done efficiently. Moreover, Gottlob et al. [GLS02] proved the following statement which shows the relevance of hypertree decompositions also for general conjunctive query answering: The result of a non-boolean conjunctive query of bounded hypertree-width can be computed in time polynomial in the combined size of the input instance and of the output relation.

While the tractability results for conjunctive query answering strongly suggest that hypertree decompositions can be used to speed-up database queries (an assumption which is confirmed by, e.g., [GGGS07] and [SGL07]), there is also another important domain in which hypertree decompositions have beneficial effects on the performance, namely in the area of constraint satisfaction problems. Such problems arise especially in the area of operations research, a field of artificial intelligence, and they ask for concrete assignments of values to sets of variables such that all the given constraints are satisfied. Extensive surveys on the topic of constraint satisfaction are given, for instance, by Apt [Apt03] and Dechter [Dec03]. In [GLS01], Gottlob et al. highlight that constraint satisfaction is not only closely related to the evaluation problem of boolean conjunctive queries, but that the two problems are essentially the same under certain assumptions (see also [GJC94] and [KV00]). Hence, hypertree decomposition cannot only be used for the design of efficient query plans but also to speed-up the solving process of constraint satisfaction problems. At this point, we want to remind the reader of the fact that the *htd* framework has built-in support for computing hypertree decompositions.

Indeed, apart from the numerous participants of the PACE challenge (see Section 4.5) as well as `https://pacechallenge.wordpress.com`) there also exist other tools and software frameworks for decomposing graphs and hypergraphs. Prominent examples include *QuickBB* [GD04], *libTW* [vDvdHS06] or *htdecomp* [DGG$^+$08]. Unfortunately, these three projects seem to be abandoned in the meantime as the last update to these software artifacts was made years ago. A potential alternative to the aforementioned tools is the *dlib* software library (see `http://www.dlib.net`) whose focus is on machine learning algorithms and which also provides functionality to compute join trees of graphs. According to its developers, *dlib* is a "modern C++ toolkit containing machine learning algorithms and tools for creating complex software in C++ to solve real world problems".

The *dlib* software library is still maintained and updated regularly. Furthermore, probably one of the most prominent software collections for general graph partitioning is the *METIS* [KK98] family. Among this family of software tools we also find a powerful library for hypergraph partitioning called *hMETIS* [KAKS99].

What is common to all the tools listed above is that their primary goal is to deliver decompositions of small width. Customization of the input and output of the algorithms or optimization towards more involved criteria is therefore in most cases up to the developer of the algorithm which uses the computed decompositions.

Indeed, we are not the first ones who try to get a better handle on the efficiency of dynamic programming algorithms. Due to the fact that algorithms based on dynamic programming on tree decompositions are often very sensitive to the shape of the actually used tree decomposition, Bodlaender and Fomin [BF05] introduced the concept of tree decompositions of small cost. In this context, the cost associated to a node of the tree decomposition is defined based on a function f which maps the bag size of the given node to a real number according to the assumed time (or memory) complexity of the problem at hand. The total f-cost of a tree decomposition is then the sum over all evaluations of the formula f for the nodes of the given tree decomposition. Hence, the work by Bodlaender and Fomin allows to distinguish tree decompositions in a more fine-grained way than just by the width and in their article an extensive theoretical analysis of the problem of finding tree decompositions of minimum f-cost is provided. In the considerations made by Bodlaender and Fomin, the bag size is the only input for a function that estimates the costs of a given tree decomposition node in the context of a DP algorithm. Although no experimental evaluation is provided in the paper by Bodlaender and Fomin, it is assumed that considering all bags (and not just the largest one, i.e., the bag from which the width is derived) allows to better estimate the actual runtime of DP algorithms.

While large parts of the literature concerning dynamic programming algorithms focus on minimizing the solving time for a given problem instance, Betzler, Niedermeier and Uhlmann [BNU06] propose a heuristic which helps to significantly reduce the memory consumption of dynamic programming algorithms for solving optimization problems based on tree decompositions. In particular, the authors report on impressive memory savings between 60% and 98% in case of nice tree decompositions and path decompositions being used.

CHAPTER 7

Conclusion

The aim of this thesis was to provide valuable insights regarding the practical impact of customized tree decompositions on the average-case performance of dynamic programming algorithms. While a lot of (primarily theoretic) work on the worst-case complexity of computational problems exists, empirical evaluations are still relatively underrepresented.

7.1 Summary

With this thesis we significantly extended the body of empirical work regarding the performance analysis of dynamic programming algorithms which use tree decompositions. In the first part of the thesis at hand, we proposed a large set of features which allow to characterize tree decompositions in a much more fine-grained way than just by the plain width. Furthermore, we showed that based on the proposed features, given a pool of decompositions of a problem instance, techniques from the area of machine learning allow to reliably predict a decomposition which leads to a significantly reduced solving time compared to an arbitrary decomposition of the same width. This result was established on the basis of thousands of independent test runs on five different problem domains (Minimum Dominating Set, 3-Colorability, Perfect Dominating Set, Connected Vertex Cover and Steiner Tree). In these numerous test runs, we observed statistically significant improvements in terms of the overall solving time when following our approach to select a promising tree decomposition from a pool of heuristically generated ones using machine learning algorithms.

Although the investigation of the relative importance of features based on well-established feature selection techniques from the area of machine learning delivered no clear picture, a detailed inter-domain evaluation of the trained machine learning models presented in the first part of this thesis indicates that there must be features which are of general importance. This becomes apparent as the models which are trained for a specific problem domain often allow for a good prediction quality also in the context of other domains.

Independently from the attempt to unveil important tree decomposition features by means of feature selection techniques, our experience in the design of dynamic programming algorithms told us that the shape of join nodes probably has a significant impact on the performance of algorithms which internally use tree decompositions. Unfortunately, to the best of our knowledge, before initializing the work on this thesis, there was no software framework allowing to easily obtain tailored tree decompositions, i.e., tree decomposition which reflect certain preferences of the developer of dynamic programming algorithms.

For this reason, we developed the *htd* framework which is very competitive in real-world scenarios and which allows to directly obtain tree decompositions that are not only of low width but also adhere to a flexible, freely definable and potentially multi-level quality criterion. Furthermore, *htd* is designed with the goal to relieve the developers of dynamic programming algorithms from tedious tasks like designing post-processing routines for normalizing the decomposition or manually managing problem-specific labels such as vertex or edge weights. For all these tasks *htd* provides built-in functionality and, in the case that the routines supplied with the *htd* framework are not perfectly fitting the actual needs, there are well-documented interfaces for each part of the software library so that extending it is (relatively) easy. *htd* is free, open-source software and it is available at `http://dbai.tuwien.ac.at/research/project/decodyn/htd/`. All contributions to its functionality by the community are welcome and highly appreciated.

On the basis of the customization capabilities of the *htd* framework it was possible to investigate how we can take advantage of our knowledge in the design of dynamic programming algorithms in order to improve their runtime behavior. This investigation, in which we conducted experiments with two different systems for dynamic programming algorithms in which *htd* is used successfully, is presented in the last part of the thesis at hand. Our basic assumption for this empirical evaluation was that reducing the expected worst-case workload of the given dynamic programming algorithm for processing the join nodes of the computed decomposition is beneficial. It turned out that the customization of tree decompositions indeed affects the performance of the algorithms in a very positive way. Still, it has to be noted that finding a proper quality criterion to discriminate different tree decompositions is often far from easy, but once such a criterion is found the performance gain can be significant. For instance, in some of our experiments we observed that the average solving time dropped by more than 50% (see, e.g., Figure 5.6 in Section 5.2). Hence, providing a dynamic programming algorithm with a tree decomposition tailored specifically to the actual needs indeed seems to be a very promising approach to improve efficiency and robustness of the given algorithm and the *htd* framework makes the tasks of implementing and testing such customizations very convenient.

7.2 Future Work

The thesis at hand provides detailed insights into the average-case performance of dynamic programming algorithms based on tree decompositions and gives some suggestions on how to cope with the lack of robustness in the presence of different tree decompositions

of the same problem instance. Although we could show that our proposed approaches provide valuable benefits in various application scenarios, there is sufficient room for future work. For instance, in the context of features for the characterizations of tree decompositions, it is still unclear whether it is possible to optimize them efficiently. More precisely, it would be interesting for a given tree decomposition feature X to know whether is it possible to design a practical algorithm which is capable of directly computing a decomposition which minimizes or maximizes the outcome of evaluating X for a given problem instance. In order to be able to deal with any possible feature of tree decompositions in the optimization phase, the *htd* framework currently falls back to iteratively computing and/or modifying a heuristically generated tree decomposition of low width, but with a direct customization it may be possible to approach global lower or upper bounds for the value of a certain feature.

Apart from analyzing existing and proposing new tree decomposition features, we also want to further extend and improve the *htd* framework. Adding exact algorithms for computing tree decompositions of minimum width as well as providing options for the efficient preprocessing of the input graph (without negatively affecting the width of the resulting tree decomposition) are just two of the next steps we have on our agenda.

Finally, we want to invite researchers and software developers to contribute to the *htd* framework both by providing valuable theoretic input as well as by actively extending *htd*'s functionality.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ABC+14a] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. D-FLAT: Progress Report. Technical Report DBAI-TR-2014-86, DBAI, Fakultät für Informatik an der Technischen Universität Wien, 2014. Available at `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2014-86.pdf`.

[ABC+14b] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT System for Dynamic Programming on Tree Decompositions. In *Proceedings of the 14th European Conference on Logics in Artificial Intelligence (JELIA 2014)*, volume 8761 of *LNCS*, pages 558–572. Springer, 2014.

[ABC+15] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, and Stefan Woltran. Computing Secure Sets in Graphs using Answer Set Programming. *Journal of Logic and Computation*, 2015. Available at `https://doi.org/10.1093/logcom/exv060`.

[Abs16] Michael Abseher. htd 1.0.1, 2016. Available at `https://github.com/mabseher/htd/releases/tag/1.0.1`.

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a k-Tree. *Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[ADMW15] Michael Abseher, Frederico Dusberger, Nysret Musliu, and Stefan Woltran. Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 275–282. AAAI Press, 2015.

[AGM+16] Michael Abseher, Martin Gebser, Nysret Musliu, Torsten Schaub, and Stefan Woltran. Shift Design with Answer Set Programming. *Fundamenta Informaticae*, 147(1):1–25, 2016.

[AKA91] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, 1991.

[Ake78] Sheldon B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 100(6):509–516, 1978.

[AMS97] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal. Locally Weighted Learning. *Artificial Intelligence Review*, 11(1):11–73, 1997.

[AMW16a] Michael Abseher, Marius Moldovan, and Stefan Woltran. Providing Built-In Counters in a Declarative Dynamic Programming Environment. In *Proceedings of the 39th German Conference on Artificial Intelligence (KI 2016)*, volume 9904 of *LNCS*, pages 3–16. Springer, 2016.

[AMW16b] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd – A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. Technical Report DBAI-TR-2016-96, DBAI, Fakultät für Informatik an der Technischen Universität Wien, 2016. Available at `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-96.pdf`.

[AMW16c] Michael Abseher, Nysret Musliu, and Stefan Woltran. Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. Technical Report DBAI-TR-2016-94, DBAI, Fakultät für Informatik an der Technischen Universität Wien, 2016. Available at `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-94.pdf`.

[AMW17a] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd – A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. In *Proceedings of the 14th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR 2017)*, 2017. To appear.

[AMW17b] Michael Abseher, Nysret Musliu, and Stefan Woltran. Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. *Journal of Artificial Intelligence Research*, 2017. To appear.

[And73] Michael R. Anderberg. *Cluster Analysis for Applications*. Probability and Mathematical Statistics: A Series of Monographs and Textbooks. Academic Press, 1973.

[AOJJ89] Stig K. Andersen, Kristian G. Olesen, Finn V. Jensen, and Frank Jensen. HUGIN - A Shell for Building Bayesian Belief Universes for Expert Systems. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI 1989)*, pages 1080–1085. Morgan Kaufmann Publishers, 1989.

[AP89] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[Apt03] Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[BB72] Richard E. Barlow and Hugh D. Brunk. The Isotonic Regression Problem and Its Dual. *Journal of the American Statistical Association*, 67(337):140–147, 1972.

[BB73] Umberto Bertelè and Francesco Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148, 1973.

[BB06] Emgad H. Bachoore and Hans L. Bodlaender. A Branch and Bound Algorithm for Exact, Upper, and Lower Bounds on Treewidth. In *Proceedings of the 2nd International Conference on Algorithmic Aspects in Information and Management (AAIM 2006)*, volume 4041 of *LNCS*, pages 255–266. Springer, 2006.

[BET11] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer Set Programming at a Glance. *Communications of the ACM*, 54(12):92–103, 2011.

[BF05] Hans L. Bodlaender and Fedor V. Fomin. Tree decompositions with small cost. *Discrete Applied Mathematics*, 145(2):143–154, 2005.

[BHS03] Anne Berry, Pinar Heggernes, and Geneviève Simonet. The Minimum Degree Heuristic and the Minimal Triangulation Process. In *Proceedings of the 29th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2003)*, volume 2880 of *LNCS*, pages 58–70. Springer, 2003.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Information Science and Statistics. Springer, 2006.

[BJ13] Jori Bomanson and Tomi Janhunen. Normalizing Cardinality Rules Using Merging and Sorting Constructions. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)*, volume 8148 of *LNCS*, pages 187–199. Springer, 2013.

[BK08] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial Optimization on Graphs of Bounded Treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[BK10] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[Bli12] Bernhard Bliem. Decompose, Guess & Check: Declarative Problem Solving on Tree Decompositions. Master's thesis, TU Wien, 2012.

[BM08] Marco Benedetti and Hratch Mangassarian. QBF-Based Formal Verification: Experience and Perspectives. *Journal on Satisfiability, Boolean Modeling and Computation*, 5(1-4):133–191, 2008.

[BMW12] Bernhard Bliem, Michael Morak, and Stefan Woltran. D-FLAT: Declarative Problem Solving using Tree Decompositions and Answer-Set Programming. *Theory and Practice of Logic Programming*, 12:445–464, 2012.

[BNU06] Nadja Betzler, Rolf Niedermeier, and Johannes Uhlmann. Tree Decompositions of Graphs: Saving Memory in Dynamic Programming. *Discrete Optimization*, 3(3):220–229, 2006.

[Bod93] Hans L. Bodlaender. A Tourist Guide through Treewidth. *Acta Cybernetica*, 11:1–23, 1993.

[BPT92] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(1):555–581, 1992.

[BPW13] Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. Declarative Dynamic Programming as an Alternative Realization of Courcelle's Theorem. In *Proceedings of the 8th International Symposium on Parameterized and Exact Computation (IPEC 2013)*, volume 8246 of *LNCS*, pages 28–40. Springer, 2013.

[Bre96] Leo Breiman. Bagging Predictors. *Machine Learning*, 24(2):123–140, 1996.

[Chu36] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing (STOC 1977)*, pages 77–90. ACM, 1977.

[CMNC04] François Clautiaux, Aziz Moukrim, Stèfane Négre, and Jaques Carlier. Heuristic and meta-heuristic methods for computing graph treewidth. *RAIRO Operational Research*, 38(1):13–26, 2004.

[Cou90] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.

[CS14] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[CT95] John G. Cleary and Leonard E. Trigg. K*: An Instance-based Learner Using an Entropic Distance Measure. In *Proceedings of the 12th International Conference on Machine Learning*, pages 108–114. Morgan Kaufmann Publishers, 1995.

[CW16a] Günther Charwat and Stefan Woltran. BDD-based Dynamic Programming on Tree Decompositions. Technical Report DBAI-TR-2016-95, DBAI, Fakultät für Informatik an der Technischen Universität Wien, 2016. Available at `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-95.pdf`.

[CW16b] Günther Charwat and Stefan Woltran. Dynamic Programming-based QBF Solving. In *Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF 2016)*, volume 1719 of *CEUR Workshop Proceedings*, pages 27–40. CEUR-WS.org, 2016.

[Dec99] Rina Dechter. Bucket Elimination: A Unifying Framework for Reasoning. *Artificial Intelligence*, 113(1):41–85, 1999.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[DGG+08] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Ben McMahan, Nysret Musliu, and Marko Samer. Heuristic Methods for Hypertree Decomposition. In *Proceedings of the 7th Mexican International Conference on Artificial Intelligence (MICAI 2008): Advances in Artificial Intelligence*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.

[DHK05] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. Bounded Model Checking with QBF. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, volume 3569 of *LNCS*, pages 408–414. Springer, 2005.

[DS01] Rolf Drechsler and Detlef Sieling. Binary Decision Diagrams in Theory and Practice. *International Journal on Software Tools for Technology Transfer*, 3(2):112–136, 2001.

[ELLS11] Brian S. Everitt, Sabine Landau, Morven Leese, and Daniel Stahl. *Cluster Analysis*. Wiley Series in Probability and Statistics. John Wiley & Sons, Inc., 5th edition, 2011.

[FBN15] Stefan Fafianie, Hans L. Bodlaender, and Jesper Nederlof. Speeding Up Dynamic Programming with Representative Sets: An Experimental Evaluation of Algorithms for Steiner Tree on Tree Decompositions. *Algorithmica*, 71(3):636–660, 2015.

[FG65] Delbert R. Fulkerson and Oliver A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15:835–855, 1965.

[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[FHMW16] Johannes Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. DynASP 2.0: System Specification. Technical Report DBAI-TR-2016-101, DBAI, Fakultät für Informatik an der Technischen Universität Wien, 2016. Available at `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-101.pdf`.

[FHP02] Eibe Frank, Mark Hall, and Bernhard Pfahringer. Locally Weighted Naive Bayes. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI 2002)*, pages 249–256. Morgan Kaufmann Publishers, 2002.

[Fri02] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, 2002.

[Gav72] Fănică Gavril. Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[GD04] Vibhav Gogate and Rina Dechter. A Complete Anytime Algorithm for Treewidth. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence (UAI 2004)*, pages 201–208. AUAI Press, 2004.

[GE03] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[GGGS07] Lucantonio Ghionna, Luigi Granata, Gianluigi Greco, and Francesco Scarcello. Hypertree decompositions for query optimization. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007)*, pages 36–45. IEEE, 2007.

[GGJ+16] Serge Gaspers, Joachim Gudmundsson, Mitchell Jones, Julian Mestre, and Stefan Rümmele. Turbocharging Treewidth Heuristics. In *Proceedings of the 11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, 2016. To appear.

[GGLS16] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2016)*, pages 57–74. ACM, 2016.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of* NP*-Completeness*. W. H. Freeman, 1979.

[GJC94] Marc Gyssens, Peter G. Jeavons, and David A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66(1):57–89, 1994.

[GLS01] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions: A Survey. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science (MFCS 2001)*, volume 2136 of *LNCS*, pages 37–57. Springer, 2001.

[GLS02] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[GMS09] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized Hypertree Decompositions: NP-hardness and Tractable Variants. *Journal of the ACM*, 56(6):1–32, 2009.

[GNPT05] Enrico Giunchiglia, Massimo Narizzano, Luca Pulina, and Armando Tacchella. Quantified Boolean Formulas Satisfiability Library (QBFLIB). `www.qbflib.org`, 2005.

[Gut15] Gregory Gutin. Should We Care about Huge Imbalance in Parameterized Algorithmics? *The Parameterized Complexity Newsletter*, 11(2), 2015.

[HA28] David Hilbert and Wilhelm Ackermann. *Grundzüge der Theoretischen Logik*, volume 27 of *Die Grundlehren der mathematischen Wissenschaft*. Springer, 1928.

[HA38] David Hilbert and Wilhelm Ackermann. *Grundzüge der Theoretischen Logik*, volume 27 of *Die Grundlehren der mathematischen Wissenschaft*. Springer, 2nd edition, 1938.

[HA50] David Hilbert and Wilhelm Ackermann. *Principles of Mathematical Logic*, volume 69. American Mathematical Society, 1950.

[Hal76] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

[HD96] Cecil Huang and Adnan Darwiche. Inference in Belief Networks: A Procedural Guide. *International Journal of Approximate Reasoning*, 15(3):225–263, 1996.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Exploration Newsletters*, 11(1):10–18, 2009.

[HHF99] Geoffrey Holmes, Mark Hall, and Eibe Frank. Generating Rule Sets from Model Trees. In *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence: Advanced Topics in Artificial Intelligence (AI 1999)*, volume 1747 of *LNCS*, pages 1–12. Springer, 1999.

[HK04] Michael Haenlein and Andreas M. Kaplan. A Beginner's Guide to Partial Least Squares Analysis. *Understanding Statistics*, 3(4):283–297, 2004.

[HM10] Thomas Hammerl and Nysret Musliu. Ant Colony Optimization for Tree Decompositions. In *Proceedings of the 10th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2010)*, volume 6022 of *LNCS*, pages 95–106. Springer, 2010.

[HMS15] Thomas Hammerl, Nysret Musliu, and Werner Schafhauser. Metaheuristic Algorithms and Tree Decomposition. In *Handbook of Computational Intelligence*, pages 1255–1270. Springer, 2015.

[HS16] Michael Hamann and Ben Strasser. Graph Bisection with Pareto-Optimization. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX 2016)*, pages 90–102. Society for Industrial and Applied Mathematics, 2016.

[HXHL14] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[JLO90] Finn V. Jensen, Steffen L. Lauritzen, and Kristian G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly*, 4:269–282, 1990.

[JOA90] Finn V. Jensen, Kristian G. Olesen, and Stig K. Andersen. An algebra of Bayesian belief universes for knowledge-based systems. *Networks*, 20(5):637–659, 1990.

[JT14] Philippe Jégou and Cyril Terrioux. Bag-Connected Tree-Width: A New Parameter for Graph Decomposition. In *Proceedings of the 13th International Symposium on Artificial Intelligence and Mathematics (ISAIM 2014)*, pages 12–28, 2014.

[KAKS99] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[KBvH01] Arie M. C. A. Koster, Hans L. Bodlaender, and Stan P. M. van Hoesel. Treewidth: Computational Experiments. *Electronic Notes in Discrete Mathematics 8*, 8:54–57, 2001.

[KCHS11] Jorge Kanda, André Carvalho, Eduardo Hruschka, and Carlos Soares. Selection of algorithms to solve traveling salesman problems using meta-learning. *International Journal of Hybrid Intelligent Systems*, 8(3):117–128, 2011.

[Kjæ92] Uffe Kjærulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2(1):7–17, 1992.

[KK62] John F. Kenney and Ernest S. Keeping. Linear Regression and Correlation. *Mathematics of Statistics (3rd Edition)*, pages 252–285, 1962.

[KK98] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[Klo94] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *LNCS*. Springer, 1994.

[KLR11] Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's Theorem - A Game-Theoretic Approach. *Discrete Optimization*, 8(4):568–594, 2011.

[Koh96] Ron Kohavi. *Wrappers for Performance Enhancement and Oblivious Decision Graphs*. PhD thesis, Stanford University, 1996.

[Kot14] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 35(3):48–60, 2014.

[KV00] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, 61(2):302–332, 2000.

[KvHK99] Arie M. C. A. Koster, Stan P. M. van Hoesel, and Antoon W. J. Kolen. Solving Frequency Assignment Problems via Tree-Decomposition. *Electronic Notes in Discrete Mathematics*, 3:102–105, 1999.

[Lee59] C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

[Lif08] Vladimir Lifschitz. What Is Answer Set Programming? In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1594–1597. AAAI Press, 2008.

[LKPM97] Pedro Larrañaga, Cindy M. H. Kuijpers, Mikel Poza, and Roberto H. Murga. Decomposing Bayesian networks: Triangulation of the moral graph with genetic algorithms. *Statistics and Computing*, 7(1):19–34, 1997.

[LNS09] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical Hardness Models: Methodology and a Case Study on Combinatorial Auctions. *Journal of the ACM*, 56(4):1–52, 2009.

[LS88] Steffen L. Lauritzen and David J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B*, 50:157–224, 1988.

[LS98] Vasilica Lepar and Prakash P. Shenoy. A Comparison of Lauritzen-Spiegelhalter, Hugin, and Shenoy-Shafer Architectures for Computing Marginals of Probability Distributions. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI 1998)*, pages 328–337. Morgan Kaufmann Publishers, 1998.

[Mac98] David J.C. MacKay. Introduction to Gaussian processes. *NATO ASI Series F: Computer and Systems Sciences*, 168:133–166, 1998.

[MBT+13] Olaf Mersmann, Bernd Bischl, Heike Trautmann, Markus Wagner, Jakob Bossek, and Frank Neumann. A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem. *Annals of Mathematics and Artificial Intelligence*, 69(2):151–182, 2013.

[McM04] Ben McMahan. Bucket Elimination and Hypertree Decompositions. Technical report, DBAI, Fakultät für Informatik an der Technischen Universität Wien, 2004. Implementation Report.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw Hill Series in Computer Science. McGraw-Hill, 1997.

[MMP+12] Michael Morak, Nysret Musliu, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Evaluating Tree-Decomposition Based Algorithms for Answer Set Programming. In *Proceedings of the 6th International Conference on Learning and Intelligent Optimization (LION 2012)*, volume 7219 of *LNCS*, pages 130–144. Springer, 2012.

[MPRW10] Michael Morak, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. A Dynamic-Programming Based ASP-Solver. In *Proceedings of the 12th European Conference on Logics in Artificial Intelligence (JELIA 2010)*, volume 6341 of *LNCS*, pages 369–372. Springer, 2010.

[MRT12] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[MS07] Nysret Musliu and Werner Schafhauser. Genetic algorithms for generalized hypertree decompositions. *European Journal of Industrial Engineering*, 1(3):317–340, 2007.

[MS13] Nysret Musliu and Martin Schwengerer. Algorithm Selection for the Graph Coloring Problem. In *Proceedings of the 7th International Conference on Learning and Intelligent Optimization (LION 2013)*, volume 7997 of *LNCS*, pages 389–403. Springer, 2013.

[MT99] Victor W. Marek and Miroslaw Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.

[Mus08] Nysret Musliu. An Iterative Heuristic Algorithm for Tree Decomposition. In *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, volume 153 of *Studies in Computational Intelligence*, pages 133–150. Springer, 2008.

[Nie99] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3):241–273, 1999.

[Nie06] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press, 2006.

[NPT06] Massimo Narizzano, Luca Pulina, and Armando Tacchella. The QBFEVAL Web Portal. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA 2006)*, volume 4160 of *LNCS*, pages 494–497. Springer, 2006.

[Pap94] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[PBPPM09] Marius-Constantin Popescu, Valentina E. Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer Perceptron and Neural Networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.

[PM14] Josef Pihera and Nysret Musliu. Application of Machine Learning to Algorithm Selection for TSP. In *Proceedings of the 26th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2014)*, pages 47–54, 2014.

[PNM08] Olivier Pourret, Patrick Naïm, and Bruce Marcot. *Bayesian Networks: A Practical Guide to Applications*, volume 73. John Wiley & Sons, Inc., 2008.

[Pul16] Luca Pulina. QBFEval'16 – Competitive Evaluation of QBF solvers. `http://www.qbflib.org/qbfeval16.php`, 2016. Accessed: 2017-02-05.

[Qui92] John R Quinlan. Learning with Continuous Classes. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*, volume 92, pages 343–348. World Scientific, 1992.

[RL05] Peter J. Rousseeuw and Annick M. Leroy. *Robust Regression and Outlier Detection.* John Wiley & Sons, Inc., 2005.

[RS84] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[RS91] Neil Robertson and Paul D. Seymour. Graph minors. X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153 – 190, 1991.

[Sam59] Arthur L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[Sch06] Werner Schafhauser. New Heuristic Methods for Tree Decompositions and Generalized Hypertree Decompositions. Master's thesis, TU Wien, 2006.

[SG97] Kirill Shoikhet and Dan Geiger. A Practical Algorithm for Finding Optimal Triangulations. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Conference on Innovative Applications of Artificial Intelligence*, pages 185–190. AAAI Press / The MIT Press, 1997.

[SGL07] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. Weighted Hypertree Decompositions and Optimal Query Plans. *Journal of Computer and System Sciences*, 73(3):475–506, 2007.

[SKBM00] Shirish K. Shevade, Sathiya S. Keerthi, Chiranjib Bhattacharyya, and Karaturi R. K. Murthy. Improvements to the SMO algorithm for SVM regression. *IEEE Transactions on Neural Networks*, 11(5):1188–1193, 2000.

[SM73] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM, 1973.

[Smi08] Kate Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys*, 41(1):6:1–6:25, 2008.

[SMvHL10] Kate Smith-Miles, Jano I. van Hemert, and Xin Yu Lim. Understanding TSP Difficulty by Learning from Evolved Instances. In *Proceedings of the 4th International Conference on Learning and Intelligent Optimization (LION 2010)*, volume 6073 of *LNCS*, pages 266–280. Springer, 2010.

[SS04] Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.

[SS08] Prakash P. Shenoy and Glenn Shafer. Axioms for probability and belief-function propagation. In *Classic Works of the Dempster-Shafer Theory of Belief Functions*, volume 219 of *Studies in Fuzziness and Soft Computing*, pages 499–528. Springer, 2008.

[SWLI13] Kate Smith-Miles, Brendan Wreford, Leo Lopes, and Nur Insani. Predicting Metaheuristic Performance on Graph Coloring Problems Using Data Mining. In *Hybrid Metaheuristics*, volume 434 of *Studies in Computational Intelligence*, pages 417–432. Springer, 2013.

[Tar72] Robert E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tur36] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

[Tur37] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction. *Proceedings of the London Mathematical Society*, s2-43(1):544–546, 1937.

[TY84] Robert E. Tarjan and Mihalis Yannakakis. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[vDvdHS06] Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. Computing Treewidth with LibTW. Technical report, Utrecht University, 2006. Available at `http://treewidth.com/treewidth/docs/LibTW.pdf`.

[vWK17] Rim van Wersch and Steven Kelk. ToTo: An open database for computation, storage and retrieval of tree decompositions. *Discrete Applied Mathematics*, 217 (Part 3):389–393, 2017.

[Wan00] Yong Wang. *A new approach to fitting linear models in high dimensional spaces*. PhD thesis, The University of Waikato, 2000.

[Wol96] David H. Wolpert. The Lack of A Priori Distinctions Between Learning Algorithms. *Neural Computation*, 8(7):1341–1390, 1996.

[WW02] Yong Wang and Ian H. Witten. Modeling for Optimal Probability Prediction. In *Proceedings of the 19th International Conference on Machine Learning (ICML 2002)*, pages 650–657. Morgan Kaufmann Publishers, 2002.

[XHHL08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[XJB05] Jinbo Xu, Feng Jiao, and Bonnie Berger. A Tree-Decomposition Approach to Protein Structure Prediction. In *Proceedings of the 4th IEEE Computational Systems Bioinformatics Conference (CSB 2005)*, pages 247–256, 2005.

[ZMMN12] Shuyuan Zhang, Abdulrahman Mahmoud, Sharad Malik, and Sanjai Narain. Verification and Synthesis of Firewalls using SAT and QBF. In *Proceedings of the 20th IEEE International Conference on Network Protocols (ICNP 2012)*, pages 1–6, 2012.