FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Decompose, Guess & Check

Declarative Problem Solving on Tree Decompositions

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Bernhard Bliem
Matrikelnummer 0725395

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Privatdoz. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler

Wien, 17. Oktober 2012

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Decompose, Guess & Check

Declarative Problem Solving

on Tree Decompositions

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Bernhard Bliem
Registration Number 0725395

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Stefan Woltran
Assistance: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler

Erklärung zur Verfassung der Arbeit

Bernhard Bliem
Lerchengasse 31/5
1080 Wien

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.

(Ort, Datum) (Unterschrift des Verfassers)

Danksagung

Ich möchte mich zunächst herzlich bei meinem Betreuer, Stefan Woltran, bedanken, dessen Unterstützung beim Schreiben dieser Arbeit nicht den geringsten Wunsch offen ließ. Es war ein Privileg, die Hilfe eines Betreuers zu genießen, der nicht nur mit wertvollen Korrekturen und Anregungen zur Seite stand, sondern außerdem stets bereit war, für die Betreuung beträchtliche Mengen seiner Zeit zu opfern. Neben seiner kompetenten fachlichen Unterstützung möchte ich mich auch für das ausgesprochen angenehme Arbeitsklima bedanken, das das Schreiben dieser Arbeit zu einer Freude werden ließ.

Auch den anderen Kollegen am Arbeitsbereich DBAI, die mich bei vielen Gelegenheiten freundlich unterstützt haben, bin ich zu Dank verpflichtet. Insbesondere ist hier Reinhard Pichler zu nennen, dessen Rat bei einigen Teilen dieser Arbeit eine große Hilfe war.

Ebenso wichtig wie fachliche Unterstützung ist natürlich persönlicher Rückhalt. Deshalb möchte ich meinen Freunden Dank dafür aussprechen, dass ich mit ihnen viele schöne Stunden abseits des Schreibens dieser Arbeit verbringen konnte, dass sie Nachsicht für aus dieser Arbeit resultierende zeitliche Engpässe hatten und dass sie mich auch trotz meiner tagelangen Beschäftigung mit abstrakten Konzepten, die leicht dazu führen können, den Blick für das Wesentliche zu verlieren, (hoffentlich) vor dem Wahnsinn bewahren konnten.

Nicht zuletzt gebührt meiner Familie besonderer Dank. Vor allem gilt das für meine Eltern Adelheid und Josef, da sie mir nicht nur das Studium in dieser schönen Stadt ermöglicht haben, sondern mir auch erlaubt haben, mich so zu entfalten, wie es für mich am besten ist, ohne dass sie mir eigennützige Vorstellungen davon auferlegt hätten, wie ich in ihren Augen sein sollte. Ich bedaure, nicht soviel der Unterstützung zurückgeben zu können, wie ich erfahren durfte, und hoffe, dass die hier genannten Personen ein wenig an meiner Freude über die schönen Erfahrungen der letzten Jahre teilhaben können.

Bernhard

Abstract

Many practically relevant problems are infeasible for large instances. However, often they become tractable when only instances where a certain parameter is bounded by a constant are considered. Especially *treewidth* has proven to be an attractive parameter because it applies to many different problems. Bounded treewidth leading to tractability can frequently be exploited by *dynamic programming* on a *tree decomposition* of the original instance. Until now, implementing such algorithms, however, has usually been quite intricate which is due to the lack of supporting tools that offer an adequate language for conveniently specifying such algorithms.

In this thesis, we therefore present a method for problem solving called *Decompose, Guess & Check* that enables the user to specify such algorithms in a declarative way. For this, we employ *Answer Set Programming* – a logic programming formalism which supports a programming paradigm called *Guess & Check* and is recognized for its capability to express hard problems quite succinctly. Using Answer Set Programming as a language to specify the problem-specific parts of dynamic programming algorithms, *Decompose, Guess & Check* benefits from efficient solvers as well as from a rich language that allows for easily readable and maintainable code.

We conduct an analysis of the proposed approach that shows it to be powerful enough to efficiently solve a large class of problems on instances of bounded treewidth. Furthermore, we present a software framework called D-FLAT that provides this method and makes rapid prototyping of algorithms on tree decompositions possible. We finally apply *Decompose, Guess & Check* to a selection of different problems to illustrate the versatility of the approach.

Because instances in practical applications often exhibit small treewidth, our approach has great practical relevance. For many problems that are hard in general, *Decompose, Guess & Check* is thus a promising candidate for solving large instances which have so far been out of reach for existing Answer Set Programming systems.

Kurzfassung

Für große Instanzen sind viele praktisch relevante Probleme nicht effizient lösbar. Oftmals werden sie jedoch bewältigbar, wenn die Eingaben auf solche Instanzen beschränkt werden, die einen durch eine Konstante beschränkten Parameter aufweisen. Besonders die *Baumweite* hat sich als attraktiver Parameter erwiesen, da sie auf viele unterschiedliche Probleme anwendbar ist. Beschränkte Baumweite, die zu praktischer Lösbarkeit führt, kann häufig durch *dynamische Programmierung* auf einer *Baumzerlegung* der ursprünglichen Instanz ausgenutzt werden. Bisher war es jedoch üblicherweise äußerst aufwendig, solche Algorithmen zu implementieren, was auf den Mangel an unterstützenden Werkzeugen, die eine angemessene Sprache zur einfachen Spezifizierung bereitstellen, zurückzuführen ist.

In dieser Diplomarbeit stellen wir deshalb eine Problemlösungsmethode namens *Decompose, Guess & Check* vor, die es ermöglicht, solche Algorithmen deklarativ zu spezifizieren. Dafür verwenden wir *Antwortmengenprogrammierung* – einen logischen Programmierformalismus, der ein Programmierparadigma namens *Guess & Check* unterstützt und für seine Eignung geschätzt wird, schwierige Probleme sehr prägnant auszudrücken. Durch die Anwendung von Antwortmengenprogrammierung als Sprache zum Spezifizieren der problembezogenen Teile der dynamischen Programmierungsalgorithmen profitiert *Decompose, Guess & Check* von effizienten Solvern sowie von einer reichhaltigen Sprache, die leicht lesbaren und wartbaren Code ermöglicht.

Wir führen eine Analyse des vorgeschlagenen Ansatzes durch, die zeigt, dass dieser mächtig genug ist, um eine große Klasse an Problemen auf Instanzen beschränkter Baumweite effizient zu lösen. Weiters stellen wir ein Softwaregerüst namens D-FLAT vor, das diese Methode bereitstellt und Rapid Prototyping von Algorithmen auf Baumzerlegungen ermöglicht. Schließlich wenden wir D-FLAT auf eine Auswahl verschiedener Probleme an, um die Vielseitigkeit des Ansatzes zu illustrieren.

Da Instanzen in praktischen Anwendungen oft kleine Baumweite aufweisen, besitzt unser Ansatz hohe praktische Relevanz. Für viele im Allgemeinen schwierige Probleme ist *Decompose, Guess & Check* daher ein vielversprechender Kandidat, um große Instanzen zu lösen, die für bestehende Systeme der Antwortmengenprogrammierung bisher außer Reichweite lagen.

Contents

Chapter 1

Introduction

Since its beginnings, the study of computers has brought about powerful tools and methodologies that aid in engineering correct and efficient software. For problems of low complexity, this usually works quite well (even though writing correct, efficient and maintainable code can still be far from trivial in practice). There are, however, problems that are inherently difficult to solve, where the primary obstacles are not implementation details but rather difficulties arising from the inherent complexity of such problems that can make algorithms infeasible for large input. Even worse, in such cases classical imperative programming languages tend to yield complicated code that is hard to understand and to maintain. Declarative approaches have been developed to avoid these issues. However, even though they often at least permit clear and succinct specifications, computational complexity still remains a barrier. Thus, although declarative languages help with regard to specification, there are many cases where an appropriate language does not suffice for successful applications. Unfortunately, a lot of interesting problems fall under this category.

The main challenge posed by theoretical computer science and, in particular, artificial intelligence, that we are facing today is how to cope with hard problems on large amounts of data. This challenge is still open. There has been much theoretical progress toward understanding sources of complexity and practical progress toward building tools that try to make the best of the (presumably unavoidable) explosion of runtime for large inputs by building systems that make clever choices and thus try to "cheat death". However, there is still no entirely satisfactory answer to that challenge. As it is unlikely that problems known to be hard today turn out to be tractable after all, it is doubtful whether a "one size fits all" solution will eventually be found. Despite this, the present work proposes a solution that fits many.

Tackling intractability. Consider the question of how to deal with intractable problems. One possibility to do this is to accept solutions that might not be optimal but which are at least efficiently computable. This scheme includes approximation algo-

rithms and heuristic methods. Another possibility is to find subclasses of instances that are tractable. The drawback of inexact methods is obvious: We usually end up with suboptimal solutions. On the other hand, relying on exact methods for tractable subclasses also suffers from a serious issue: Often the constraints defining these subclasses are too restrictive for practical use. This severely limits the usefulness of many methods for dealing with hard problems on large data.

Recently, an interesting new approach appeared in the shape of *parameterized complexity theory* [Downey and Fellows, 1999, Flum and Grohe, 2006, Niedermeier, 2006]. This research area has attracted a lot of attention lately because it allows for a more fine-grained analysis of the complexity of computational problems than classical complexity theory, which only considers the size of the input as the quantity of interest. In contrast, parameterized complexity theory shifts the emphasis away from mere size of the input toward other properties.

This new perspective on the cause of intractability allows for problem solving methods that always produce exact solutions and are yet applicable to a broad range of practical problems. The idea is to put a bound on a parameter different from the input size. This way, we can often confine the exponential explosion to this parameter, which means that even huge inputs pose no difficulties as long as the parameter is bounded.

Taking internal structure into account. An especially useful parameter is *treewidth* [Robertson and Seymour, 1984], which, roughly speaking, measures the "tree-likeness" of a graph. Choosing treewidth as a parameter has significant advantages compared to other parameterizations. Since treewidth is a property of graphs, it can be applied to many problems – in fact not just to problems on graphs but to all problems that can be *represented* as a graph. In contrast, many other parameters, such as the maximum clause size in a propositional formula, only make sense for few problems, and often there is a threshold above which even relatively small values of the parameter do not help – for instance, the Boolean Satisfiability problem is tractable for instances having clause size 2, but it is NP-complete already for clause size 3.

When parameterizing a problem by treewidth, there is usually no such sharp threshold, as we can often write *fixed-parameter algorithms* whose runtime is exponential in the treewidth instead of the input size. This way, there is no rift between those instances an algorithm can handle efficiently and all the other instances that it cannot handle at all. Rather, a fixed-parameter algorithm can solve *all* instances. The transition between the (exponential) runtime in the general case and the (usually linear) runtime for the efficiently solvable instances becomes smooth. This way, we no longer merely tell efficiently solvable instances apart from infeasible ones in the crude way of a dichotomy. The question rather becomes "how feasible" an instance is. That is: What is the value of the instance's parameter that controls the exponential explosion; to wit, what is its treewidth?

The famous theorem by Courcelle [Courcelle, 1990] is an important criterion to identify problems for which bounded treewidth is useful. It shows that any problem that can be stated in the formalism of monadic second-order logic becomes tractable for instances of bounded treewidth.

In practical applications, the treewidth is small for many problems [Thorup, 1998, Agarwal et al., 2011, Gramm et al., 2008, Huang and Lai, 2007, Latapy and Magnien, 2006, Melançon, 2006]. An explanation for this is the observation that graphs in the real world usually do not resemble random graphs very much. They exhibit structure since the universe is not in a state of complete disorder.

Decomposed problem solving. To take advantage of the fact that the treewidth is often small for real-world instances, the concept of *dynamic programming on tree decompositions* [Niedermeier, 2006, Bodlaender, 1997] has been developed and proven itself to be an especially powerful technique with successful applications to many problems. This approach solves problems in a decomposed way by considering individual subproblems and combining partial solutions. To achieve this, first a *tree decomposition* of the instance is constructed, which breaks the instance down into smaller parts whose sizes only depend on the treewidth. Subsequently *dynamic programming* is employed on the obtained decomposition to compute, combine and extend partial solutions.

Declarative problem solving. Aside from the difficulties computers have with solving hard problems due to their complexity, such problems are also especially challenging for the human who must write an algorithm that is ideally easy to read and to maintain. Declarative approaches have proven to be particularly well suited for such tasks, as they allow the user to focus on the "what" instead of the "how".

Answer Set Programming (ASP) [Brewka et al., 2011, Lifschitz, 2008] is a logic programming language that allows for succinct specifications of computationally hard problems. In particular, ASP makes it easy to write programs that follow the *Guess & Check* principle where a *guess* is performed to open up the search space nondeterministically and a subsequent *check* phase eliminates all guessed candidates that turn out not to be solutions. Compared to implementations in imperative languages, ASP programs have the advantage of being often much easier to read and to maintain. There are sophisticated ASP solvers available that offer high efficiency and provide a rich language for modeling problems in a natural way, which is one of the reasons why ASP is by now widely acknowledged as a powerful way to solve hard problems.

The state of the art and its shortcomings. Tree decompositions, dynamic programming and declarative problem solving have been studied to a great extent. Each of these notions has established itself as an invaluable concept. Plenty of software tools have been developed to employ them in practice (cf. Section 5.1.2). However, none of the existing approaches that we know of combines these areas in an adequate way.

- On the one hand, there are frameworks that allow users to exploit decomposition but not to do so in a declarative way. This narrows their usefulness because especially for hard problems easily readable and maintainable imperative implementations are often much more difficult to come up with than declarative specifications.

- On the other hand, logic programming languages like ASP make declarative problem solving possible but do not offer decomposition. This is unfortunate because these languages thereby ignore the potential of solving problems in a decomposed way and so cannot compete with dedicated solutions that exploit bounded treewidth.

- Beyond that, attempts have been made to develop systems that make use both of decomposition and declarative specification of problems by evaluating logical formulas with the help of tree automata or tree decompositions [Klarlund et al., 2002, Kneis et al., 2011].

  At first glance, the very high level of declarativity that such logical formalizations offer appears to be a virtue. However, sometimes the excess of virtue is a vice: It has been found that implementations of such approaches usually suffer from bad practical performance. This is explained by the fact that they cannot exploit knowledge about the application domain like a tailored algorithm can.

  In spite of this, we do not believe that we must therefore give up declarativity and resort to imperative languages – no one shall expel us from the paradise that declarative problem solving has created. Rather, we conclude that the *right level* of declarativity for decomposed problem solving has not yet been found.

Main goal. Our goal is to offer a method for combining the advantages of decomposition methods with those of declarative problem solving in such a way that the intuition behind a problem-specific dynamic programming algorithm on a tree decomposition does not get lost. We wish to provide a means to specify algorithms that explicitly make use of knowledge about the domain – like an imperative implementation does – while at the same time offering a high level of declarativity in order to benefit from the known advantages of declarative problem solving.

Since especially ASP with its *Guess & Check* approach has proven to be an excellent choice for solving many hard problems, we seek a bridge between the worlds of ASP and dynamic programming on tree decompositions.

Combining decomposition methods with declarative problem solving. In this thesis, we present an approach called *Decompose, Guess & Check* that fits the bill. It proceeds by providing the user with an automatically constructed tree decomposition and,

for the actual problem-specific algorithm, executing a user-supplied ASP encoding to perform dynamic programming.

There has so far been no system that brings the advantages of ASP to bear in a decomposed setting. What makes the proposed approach novel is thus that it is now possible to specify concrete dynamic programming algorithms on tree decompositions in a declarative way.

Using ASP to compute the partial solutions of the subproblems can be motivated by the observation that ASP is well suited for a lot of problems and is thus often also well suited for parts of such problems. Therefore, we can benefit from its advantages also in a decomposed setting. Because ASP originated in part from research on databases, it can be conveniently used to specify table transitions, which are the typical operations in dynamic programming. Thus, ASP is an excellent choice for realizing dynamic programming algorithms on tree decompositions.

Challenges. Combining dynamic programming on tree decompositions with the *Guess & Check* paradigm found in ASP is not a trivial task. Several demanding challenges have to be overcome in order to achieve our goals.

We wish to use ASP for dynamic programming on tree decompositions, but at the same time we want to avoid extending the ASP syntax. This is because introducing syntactical changes would lead to yet another programming language for which existing technology cannot be used directly. Using the established language of ASP allows us to benefit from future improvements to the already very sophisticated solvers.

To accomplish this, we need a framework that offers the power of decomposition to ASP by employing a high-performance solver to execute user-supplied ASP encodings that specify the table computations required for dynamic programming. To make this possible, appropriate data structures for the objects that are being manipulated during the run of an algorithm must be chosen and, most importantly, an interface between the user's declarative encodings and the framework must be specified. This is required so that the problem-specific encodings receive all necessary information from the framework to perform the dynamic programming task, and that the framework can interpret the results of the encodings to populate the data structures accordingly. For the latter task, we need to define how the contents of the dynamic programming tables can be extracted from the answer sets produced by the user's encoding. It is moreover vital that the interface be designed so as to meet the requirements for solving all kinds of different problems, and care must be taken to find a balance between flexibility and simplicity.

Contributions. Our main contribution is the development of a declarative problem solving technique called *Decompose, Guess & Check* that makes use of dynamic programming on tree decompositions using ASP as a language for the problem-specific encodings. This endeavor comprises the following most notable items.

- We introduce the features of the *Decompose, Guess & Check* approach, describe its theoretical underpinnings and analyze its properties. The main innovation is that it provides an instrument to use the efficiency of dynamic programming on tree decompositions combined with the power of ASP for succinctly modeling hard problems.

  The benefits of *Decompose, Guess & Check* can be seen from multiple aspects. From the perspective of the practitioner, *Decompose, Guess & Check* can be seen as a valuable method for developing rapid prototypes of dynamic programming algorithms that operate on tree decompositions. It can also be useful for the researcher or student because it allows for quickly implementing such algorithms. Therefore, *Decompose, Guess & Check* is also attractive for educational purposes because it lowers the barrier for getting acquainted with dynamic programming on tree decompositions by allowing the user to readily experiment with this problem solving technique. This promotes understanding of the underlying concepts, which many consider to be a bit difficult to grasp at first. Finally, from the aspect of a member of the ASP community, *Decompose, Guess & Check* is a vehicle that opens up the possibility of solving problems in a decomposed way without giving up the advantages of ASP.

- *Decompose, Guess & Check* allows the programmer to focus on the actual problem-specific algorithm by writing a declarative encoding. It therefore makes it easier to develop and maintain dynamic programming algorithms on tree decompositions. To effectively use it in practice, we give guidelines for designing algorithms with *Decompose, Guess & Check*.

- We present our implementation of a software framework for developing algorithms by means of *Decompose, Guess & Check*. The user can therefore rely on tedious non-problem-specific tasks being handled by a ready-to-use framework that allows the actual problem-specific computations to be specified by means of ASP.

  So far, dynamic programming on tree decompositions has often been an intricate task – not only because dynamic programming takes some getting used to, but also since it involves steps that have little to do with the actual problem and are perceived as a burden to the programmer, such as providing suitable data structures, parsing input, constructing a "good" tree decomposition (which is in itself a quite difficult task) and implementing the data flow. Until now, these unrelated issues often stood in the way of quickly implementing an idea for such an algorithm.

  Our framework eliminates these obstacles by dismissing the requirement to reinvent the wheel for each problem. We thus relieve the user from having to deal with issues that distract from the essentials of the algorithm.

- We show that *Decompose, Guess & Check* and our framework are powerful enough to solve all problems that are definable in the formalism of monadic second-order logic and to do so efficiently for graphs of bounded treewidth. We thus provide evidence that *Decompose, Guess & Check* is well equipped for solving a multitude of hard problems efficiently if the treewidth of instances is bounded.

  Furthermore, this result amounts to an alternative approach to proving Courcelle's theorem. Because it is conceptually simpler than the original proof, it might even be of theoretical use in the future by allowing us to prove extensions of Courcelle's theorem that are difficult to prove otherwise.

- We list encodings for various kinds of problems for illustration. These case studies comprise problems from graph theory (e.g., Graph Coloring, Minimum Vertex Cover and Hamiltonian Cycle), logic (e.g., Boolean Satisfiability, Disjunctive Answer Set Programming and Quantified Boolean Formulas) and other fields (e.g., Cyclic Ordering). This shows that *Decompose, Guess & Check* is indeed well suited for solving a wide range of problems by means of succinct declarative specifications. Some of our encodings implement novel algorithms, and we even consider problems for which we are not aware of any published fixed-parameter algorithm exploiting treewidth.

- We report preliminary experiments, which confirm that *Decompose, Guess & Check* can be successfully employed in practice. We manage to outperform traditional ASP systems on some hard problems for large instances of bounded treewidth.

Organization. This thesis is structured in the following way. Chapter 2 covers the necessary background – in particular it gives brief introductions to computational complexity theory, dynamic programming, ASP and tree decompositions. Our main contribution is presented in Chapter 3 where we introduce the *Decompose, Guess & Check* approach to problem solving. After giving an overview, we state a methodology to implement concrete algorithms and study the applicability of the approach. This is complemented by Chapter 4 where we present a framework that allows putting the *Decompose, Guess & Check* approach into action. After describing this system in a general way, we give an extensive collection of case studies for various problems from different domains to illustrate the versatility of the presented approach. The last section of that chapter is devoted to an experimental evaluation. Finally, Chapter 5 concludes this work with a discussion of the obtained results, including an investigation of related work as well as a summary and an outlook.

Achievements. Our framework for implementing algorithms with *Decompose, Guess & Check* is released as free software at `http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/`. In [Bliem et al., 2012], we have presented a prototypical

implementation of this framework. The article was accepted as a full paper for the *28th International Conference on Logic Programming* (ICLP 2012), a prestigious conference for research concerned with logic programming in general and ASP in particular. The paper was then published in the renowned journal *Theory and Practice of Logic Programming* (TPLP). In the current work, we significantly generalize and extend the concepts from that preceding article.

Chapter 2

Background

This chapter describes the preliminary concepts and underlying theory of the approach presented in this work. Section 2.1 discusses the most relevant topics of computational complexity theory, Section 2.2 is about dynamic programming, Section 2.3 is devoted to Answer Set Programming, and Section 2.4 covers tree decompositions.

2.1 Computational Complexity

Perhaps the most important contribution to the foundation of computer science as an area of theoretical research was Alan Turing's seminal paper [Turing, 1936] on computable numbers in which he introduced a model of computation that came to be known as the concept of *Turing machines*. By means of such machines, Turing provided a formal description of what it means to compute something and thus of precisely what can be called an algorithm. His motivation was to solve one of the most important questions concerned with the foundation of mathematics at the time, the so-called *Entscheidungsproblem*, which is, roughly speaking, the question of whether the truth or falsity of any statement in predicate logic can be determined by an algorithm. The Entscheidungsproblem was an important part of Hilbert's program – an ambitious attempt, formulated by David Hilbert, to provide a reliable foundation of mathematics. Turing proved that the answer to the Entscheidungsproblem is negative. This means that some mathematical problems cannot possibly be solved by a computer. Thereby, Turing showed that one of the core elements of Hilbert's program was impossible to carry out.

Thus a sharp boundary between two fundamental classes of computational problems has been established that divides decidable from undecidable problems. The research area of computational complexity builds upon this result and goes one step further: Given that some problems are decidable, while others are not, it is natural to ask about the decidable problems *how difficult* it is to solve them. By "difficulty" of a problem, we usually mean *time complexity*, i.e., the asymptotic worst-case running

time of any possible algorithm for the problem. Other resources, such as memory consumption, can also be considered, but unless explicitly noted otherwise, we always refer to time complexity.

Computational complexity is highly relevant also in practice: When asked to implement a program that solves a particular problem, there is often the question of whether the resulting algorithm is unnecessarily cumbersome or whether it solves the problem as efficiently as possible (save for optimizations that are negligible from a theoretical point of view). For instance, when it has been established that there cannot be a tractable algorithm for a particular problem, vain efforts to produce an efficient implementation – one whose runtime is polynomial in the input size – can be directed to more fruitful tasks.

A basic distinction within the class of decidable problems is thus between *tractable* and *intractable* problems. Informally, a problem is called intractable if each algorithm for it must have the property that "small" growth of the size of the problem instance leads to an "excessive" increase in running time. By such an "excessive increase" we mean what is often referred to as "exponential explosion" which occurs when the running time is exponential in the size of the input. On the other hand, we call a problem tractable if there is an algorithm that runs in polynomial time. The remainder of this section is devoted to formalizing basic complexity-theoretic notions. For thorough introductions to complexity theory, we refer to [Papadimitriou, 1994] and [Garey and Johnson, 1979].

2.1.1 Basic Complexity Theory

The objects of study in complexity theory are computational problems.

Definition 2.1. Let Σ denote a finite alphabet and Σ^* the set of all finite strings over Σ. A *decision problem* is a language $L \subseteq \Sigma^*$. We call any $x \in \Sigma^*$ an *instance* (or *input*) of the problem. x is called a *positive instance* if $x \in L$, and a *negative instance* otherwise. □

In order to make specifications more readable, we usually state problems by describing what we allow as instances and posing a question that can be asked about an instance. For example, the 3-Colorability problem for graphs (3-Col for short) is defined as follows:

Input:	A graph $G = (V, E)$
Question:	Is there a proper 3-coloring of G, i.e., a mapping $f : V \to \{\text{red}, \text{green}, \text{blue}\}$ such that for each edge $(a, b) \in E$ it holds that $f(a) \neq f(b)$?

Given a positive instance of a problem, we denote by *solution* an object that is a *witness* (or *certificate*) for the instance being positive. For example, the solutions of a 3-Col instance are its 3-colorings.

The most widely studied kind of problems in complexity theory are decision problems, but there are others, too. For decision problems, the question we ask is always a yes-no question. For other types of problems we may also specify a computation task that is to be done with an instance instead of formulating a question. For example, other important problem types include *counting problems* ("How many solutions exist?"), *search problems* ("Print any solution.") and *enumeration problems* ("Print all solutions."), as well as optimization variants where solutions are additionally required to have a minimum associated cost. In the following, we will always consider decision problems unless stated otherwise.

A *complexity class* subsumes all problems that are solvable with a certain bound on the resources. The most prominent classes are P and NP, which we now define.

Definition 2.2. A problem is in the class P if it can be decided by a deterministic Turing machine requiring time polynomial in the input size. □

Analogously, the class NP is defined using non-deterministic Turing machines as the model of computation. When a non-deterministic Turing machine is in a certain state and reads a symbol, there can be more than one successor configuration. The computation (i.e., the sequence of configurations) thus no longer proceeds linearly but can be depicted as a computation tree. A non-deterministic Turing machine is said to *accept* the input if *at least one* computation path in that tree leads to an accepting state.

Definition 2.3. A problem is in the class NP if it can be decided by a non-deterministic Turing machine requiring time polynomial in the input size. □

We call a problem tractable if the runtime of a deterministic algorithm is polynomial in the input size. Clearly, the problems in P are tractable by definition. Obviously, $\mathsf{P} \subseteq \mathsf{NP}$, but it is unknown whether this inclusion is proper. It is believed that $\mathsf{P} \neq \mathsf{NP}$, which would imply that many problems in NP are intractable, i.e., that exponential time is required to solve them with a deterministic algorithm.

For each decision problem there is also a corresponding complement problem which can be obtained by just inverting the question. Correspondingly, for each complexity class there is a complement class (denoted by the prefix "co-"); for instance, the complement class of NP is called co-NP. Here, it is not just required that a non-deterministic Turing machine accepts on *any* computation path but rather that it accepts on *all* of them. Whether $\mathsf{NP} = \mathsf{co\text{-}NP}$ is another prominent open question. It is believed that $\mathsf{NP} \neq \mathsf{co\text{-}NP}$.

A most important technique in complexity theory are *reductions*. Informally, a reduction is a procedure that transforms an instance of one problem to an instance of another problem such that the answer to the question stays the same.

Definition 2.4. We say that a function $R : \Sigma^* \to \Sigma^*$ is a polynomial-time many-one reduction (in the following only called *reduction*) from a problem A to a problem B if for any input string $x \in \Sigma^*$ two conditions are satisfied: $x \in A$ if and only if $R(x) \in B$, and $R(x)$ can be computed in polynomial time by a deterministic algorithm. □

Definition 2.5. If a reduction from A to B exists, we call A *reducible* to B, denoted by $A \leq_m B$. □

From $A \leq_m B$ we can conclude that A is at most as hard as B. The hardest problems of a complexity class are called *complete* for this class.

Definition 2.6. Given a complexity class C, a problem Π is called C-*hard* if for any $\Pi' \in \mathsf{C}$ it holds that $\Pi' \leq_m \Pi$. Π is called C-*complete* if additionally $\Pi \in \mathsf{C}$. □

If we can prove a problem Π to be complete for a ("natural") class C, we have gained a good understanding about the difficulty of Π. To do this, we commonly show membership (usually by giving an algorithm appropriate to C) and hardness (usually by reducing a C-complete problem to Π).

The prototypical example of an NP-complete problem is the Boolean Satisfiability problem (for short Sat):

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

A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses. The problem of deciding satisfiability of a CNF formula with at most three literals per clause is known as 3-Sat and is also NP-complete, whereas 2-Sat – where each clause has at most two literals – is known to be in P.

2.1.2 Complexity and "Guess & Check"

An alternative characterization of NP is by means of *succinct certificates* (also called *witnesses*). Given a solution to an instance of a problem in NP, it can be efficiently verified whether this solution is indeed a witness to the instance being positive, whereas it is believed that actually *finding* a solution is difficult.

We formalize this relationship between instances and witnesses that is characteristic for NP by means of certificate relations which must fulfill certain conditions.

Definition 2.7. An $(n+1)$-ary relation R over Σ^* is called *polynomially balanced* if for any $(x, y_1, \ldots, y_n) \in R$ and $1 \leq i \leq n$ it holds that $|x| \leq |y_i|$. □

Definition 2.8. An $(n+1)$-ary relation R over Σ^* is called *polynomial-time decidable* if there is a deterministic algorithm that, given an $(n+1)$-tuple t over Σ^* as input, decides in polynomial time whether $t \in R$. □

Theorem 2.9. *Let $L \subseteq \Sigma^*$ be a language. $L \in$ NP if and only if there is a polynomial-time decidable and polynomially balanced relation $R \subseteq \Sigma^* \times \Sigma^*$ such that*

$$L = \{x \mid \exists y : (x, y) \in R\}.$$ □

This characterization is especially interesting for the purpose of this work, since it suggests a methodology of problem solving that is known as *Guess & Check* and will

play a substantial role when we present our own extension of this approach. When confronted with an instance of a problem in NP, a non-deterministic Turing machine can first guess a candidate for a solution (thus yielding multiple computation paths) and then check whether this candidate is indeed a solution. For non-deterministic Turing machines, all this is possible in polynomial time because it suffices for *any* computation path to succeed in order to yield the answer "yes". As we have just seen, verifying whether some candidate is a witness (i.e., the "check" part) can be done efficiently.

Note that for a problem in co-NP we require a Turing machine to accept on *all* computation paths, which appears not to be solvable by *Guess & Check* in general (otherwise we would have a proof for $\mathsf{NP} = \text{co-}\mathsf{NP}$).

Since in reality we are unfortunately not in possession of a non-deterministic Turing machine, it is probably not possible to solve NP-complete problems efficiently. However, the *Guess & Check* approach is nevertheless valuable in practice, since it allows us, if not to *solve* such problems efficiently, at least to *succinctly specify* programs to solve them. Especially Answer Set Programming (cf. Section 2.3) is a prime example for a means to achieve this [Leone et al., 2006].

2.1.3 The Polynomial Hierarchy

Of course, some problems are even more difficult than NP-complete problems. A notable example are problems in the *polynomial hierarchy* which we will now define.

First, a brief introduction to oracle machines is in order. A Turing machine with an oracle for the class O may always call this oracle which can decide any problem in O in just one step. For any time complexity class C, we denote by C^{O} the class of all problems decidable by a Turing machine of the same kind (e.g., deterministic or non-deterministic) as in C, having access to an oracle for O, with the same resource bound (e.g., polynomial time).

Definition 2.10. The basis of the polynomial hierarchy is defined as $\Sigma_0^{\mathsf{p}} = \mathsf{P}$, and the higher levels are recursively defined as $\Sigma_i^{\mathsf{p}} = \mathsf{NP}^{\Sigma_{i-1}^{\mathsf{p}}}$ for $i > 0$. We also define the related classes $\Delta_i^{\mathsf{p}} = \mathsf{P}^{\Sigma_i^{\mathsf{p}}}$ and $\Pi_i^{\mathsf{p}} = \text{co-}\Sigma_i^{\mathsf{p}}$ for $i \geq 0$. The *polynomial hierarchy* is defined as $\mathsf{PH} = \bigcup_{i \geq 0} \Sigma_i^{\mathsf{p}}$. □

Note that any problem in the polynomial hierarchy can be decided in polynomial space.

Definition 2.11. PSPACE is the class of all problems decidable in polynomial space. □

Theorem 2.12. $\mathsf{PH} \subseteq \mathsf{PSPACE}$ □

Analogous to Theorem 2.9, the levels of the polynomial hierarchy can also be characterized by means of certificate relations and quantifier alternation:

Theorem 2.13. *Let $L \subseteq \Sigma^*$ be a language and $i \geq 0$. $L \in \Sigma_i^p$ if and only if there is a polynomial-time decidable and polynomially balanced $(i+1)$-ary relation R over Σ^* such that*

$$L = \{x \mid \exists y_1 \forall y_2 \exists y_3 \dots Q y_i : (x, y_1, \dots, y_i) \in R\}$$

where Q is $\exists$ for odd i and $\forall$ for even i. □

Quantifier alternation also plays a role in QSAT_i which is the showcase problem that captures the complexity of the ith level of the polynomial hierarchy. QSAT_i is the following problem:

Input:	A formula of the form $\exists X_1 \forall X_2 \exists X_3 \dots Q X_i \ \phi$ where Q is $\exists$ for odd i and $\forall$ for even i, ϕ is a propositional formula and X_j (for $1 \leq j \leq i$) is a set of propositional variables
Question:	Is there a truth assignment to the variables in X_1 such that for any truth assignment to the variables in X_2 there is a truth assignment to the variables in X_3, and so on, such that ϕ is satisfied?

Theorem 2.14. QSAT_i *is Σ_i^p-complete.* □

A generalization of QSAT_i, where we drop the requirement of a bounded number of quantifier alternations, is the QSAT problem:

Input:	A formula of the form $\exists x_1 \forall x_2 \exists x_3 \dots Q x_i \ \phi$ where Q is $\exists$ for odd i and $\forall$ for even i, ϕ is a propositional formula and x_j (for $1 \leq j \leq i$) is a propositional variable
Question:	Is there a truth assignment to x_1 such that for any truth assignment to x_2 there is a truth assignment to x_3, and so on, such that ϕ is satisfied?

Theorem 2.15. QSAT *is* PSPACE*-complete.* □

2.1.4 Parameterized Complexity Theory

Parameterized complexity [Downey and Fellows, 1999, Flum and Grohe, 2006, Niedermeier, 2006] is a recent development in complexity theory that quickly gained a lot of attention. Here, one not only studies the behavior of algorithms for problems in the face of growing instance sizes but also considers other parameters than mere input size. The motivation is that, given an intractable problem, we wish to gain a deeper insight into the question of what properties of instances are responsible for the intractability. For many problems it can be seen that restricting ourselves to instances without some "evil" property makes the problem easy even for huge inputs.

Definition 2.16. Let Σ denote a finite alphabet. A *parameterized decision problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$. We call any $(x, k) \in \Sigma^* \times \mathbb{N}$ an *instance*; x is the *main part* and k is the *parameter*. □

Given an element of $\Sigma^* \times \mathbb{N}$, the main part is analogous to a traditional problem instance, whereas the parameter expresses some property of the main part.

Definition 2.17. A parameterized problem L is called *fixed-parameter tractable* if, for any $(x, k) \in \Sigma^* \times \mathbb{N}$, a deterministic Turing machine can decide whether $(x, k) \in L$ in time $f(k) \cdot |x|^{\mathcal{O}(1)}$, where f is some computable function only depending on the instance's parameter k. □

Of course, there are various ways to parameterize a problem. The Graph Colorability problem (a generalization of 3-Col from Section 2.1 where the number of colors is part of the input) could, for example, be parameterized by the number of colors or by the graph's treewidth (which we will introduce in Section 2.4). When declaring that a problem is fixed-parameter tractable, one therefore always has to state which parameterization is considered. For instance, we will see that Graph Colorability is fixed-parameter tractable w.r.t. treewidth, but it is doubtful whether it is so w.r.t. the number of colors, because then we would have proven $\mathsf{P} = \mathsf{NP}$ since we know that 3-Col is NP-complete.

It is possible that two problems are both classified as, say, NP-complete in traditional complexity theory, although one is fixed-parameter tractable w.r.t. some parameter while the other is not. For instance, the problems Vertex Cover and Clique (which we do not define at this point) are both NP-complete; however, the former is fixed-parameter tractable w.r.t. solution size whereas it is believed that the second is not [Niedermeier, 2006]. Hence, parameterized complexity allows for a more sophisticated classification of problems.

2.2 Dynamic Programming

Dynamic programming [Larson, 1967, Cormen et al., 2009, Dasgupta et al., 2006] is a problem solving strategy that tries to compute solutions for a problem by recursively dividing it into subproblems[1] such that the solutions of the basic subproblems can be used to construct the solutions of the larger subproblems. The reason this is often favorable to a naive recursive implementation is that dynamic programming avoids solving the same subproblems over and over again by storing already computed solutions for a subproblem in a table that is associated with it.

The approach of *divide and conquer algorithms* is conceptually related [Cormen et al., 2009] – however, in dynamic programming the subproblems need not be *substantially* smaller, as they are for instance in the merge sort algorithm which always divides sub-

[1] Actually, we are dividing *instances*, not problems. The established terminology is here somewhat sloppy but we stick with it when there is no danger of confusion.

problems in half, leading to $\mathcal{O}(n \log n)$ runtime compared to $\mathcal{O}(n^2)$. More precisely, the term "divide and conquer" is normally used when the subproblems are disjoint, whereas in dynamic programming they generally overlap.

Another related approach is *memoization* [Cormen et al., 2009]. While dynamic programming generates solutions to subproblems even if those solutions are not actually required to create a global solution, memoization only computes what is necessary on demand. It can be seen as a recursive strategy that – in contrast to a naive implementation – remembers which subproblems have already been solved. Hence, while dynamic programming proceeds *bottom-up* when computing the tables for each subproblem, memoization goes *top-down*. When using memoization, one does not have to change much of a naive recursive implementation. In particular, memoization does not require the user to explicitly figure out a reasonable order in which to process the subproblems. Drawbacks of memoization are the overhead of permanently doing table lookups, that the level of recursions can become too large and that memory problems occur when a lot has been cached that is no longer needed.

2.2.1 Properties of Dynamic Programming

Problems that are good candidates to solve with dynamic programming exhibit the following properties.

Overlapping subproblems: When computing a solution, some subproblems occur multiple times. This property is responsible for the fact that a naive recursive implementation is inefficient because it performs the same computations many times.

Optimal substructure: When dealing with an optimization problem, if a solution is optimal for the global problem, then it also contains optimal solutions to all subproblems. (For non-optimization problems, optimality coincides with validity.)

2.2.2 Example: Minimum Vertex Cover on Trees

As an example for dynamic programming, we present an algorithm that solves the Minimum Vertex Cover problem on trees.

A *vertex cover* of a graph is a subset of the vertices that contains at least one endpoint of each edge. The Minimum Vertex Cover problem is defined as follows.

Input: A graph
Task: Determine the smallest size of all vertex covers of the graph.

It is well known that Minimum Vertex Cover is NP-complete. However, if we restrict the instances to trees, the problem is easy to solve. We use this special case to illustrate the principle of dynamic programming by providing an algorithm adhering to this

method. Before we do this, however, we will present a naive recursive algorithm that suffers from the mentioned issues.

For our naive recursive algorithm, let v be the current vertex in a top-down traversal. Whether a vertex v is contained in a minimum vertex cover (v is "in") or not (v is "out") depends on whether its children are contained, because if there is an "out" child, we are forced to conclude that v is "in" to cover the edge to that child, whereas otherwise we can choose whether v is contained or not such that the total cost is minimized. As we do not know its status in advance, we compute two values for v: The cost of the minimum vertex cover of the subtree rooted at v under the condition that v is "in" ($c_i(v)$) resp. "out" ($c_o(v)$). We let $\text{ch}(v)$ denote the children of v and define:

$$c_i(v) = 1 + \sum_{w \in \text{ch}(v)} \min(c_i(w), c_o(w)) \qquad c_o(v) = \sum_{w \in \text{ch}(v)} c_i(w)$$

We compute these functions for the root, choose whichever is smaller and thus obtain the size of a minimal vertex cover.

Although this algorithm is correct, it is unnecessarily inefficient. For a vertex v, the number of times $c_i(v)$ and $c_o(v)$ are computed is exponential in the depth of v.[2] This indicates we are dealing with overlapping subproblems. Also the property of optimal substructure is fulfilled, for if a subproblem had a smaller vertex cover, we could use it to construct a smaller global vertex cover. So we give dynamic programming a try.

In this example, not much work is required to turn this exponential recursive algorithm into a dynamic programming algorithm. The following algorithm does the job. For each vertex v we store a table which will eventually contain two entries, $c_i(v)$ and $c_o(v)$. The meaning of these values is just as before. The difference is that now we are doing a bottom-up traversal: We start at the leaves and compute the respective two function values which clearly takes constant time for each leaf; the results are stored in the tables of the leaves. When we consider a vertex with the property that all its children's tables are already filled, we can use the stored entries in the child tables to compute the entries in the current table in constant time. When we have reached the root, we can read off the optimum cost from the table. This algorithm is clearly feasible in linear time. We can also construct a minimum vertex cover in linear time by a final top-down traversal.

This simple example should have served to illustrate the differences between a naive recursive approach and dynamic programming. We will get back to dynamic programming in a more interesting setting, namely when we discuss dynamic programming on tree decompositions in Section 2.4. In Section 4.2.4 we will even see such an algorithm for Minimum Vertex Cover on arbitrary graphs.

[2] In fact, the number of computations grows exactly like the Fibonacci sequence.

2.3 Answer Set Programming

Since NP-complete problems are believed not to be solvable in polynomial time, in principle we probably cannot do better than an algorithm that guesses (potentially exponentially many) candidates and then checks (each in polynomial time) if these are indeed valid solutions, because, as we have seen (cf. Theorem 2.9), problems in NP can be solved via *Guess & Check*. Logic programming under the answer set semantics is a formalism that allows us to succinctly specify programs that follow such an approach [Baral, 2003, Gelfond and Leone, 2002, Leone et al., 2006]. *Answer Set Programming* (ASP) denotes a paradigm in which one writes a logic program to solve a problem such that the answer sets of this program correspond to the solutions of the problem. Easily accessible introductions are given in [Brewka et al., 2011, Lifschitz, 2008]. [Niemelä, 1999, Marek and Truszczyński, 1999] propose ASP as a paradigm for declarative problem solving. In particular, the "disadvantage" of the answer set semantics of admitting many (or even zero) solutions, which was initially a concern among researchers, has been turned into a virtue by arguing that multiple models allow for modeling non-deterministic computations naturally. Thus, ASP allows us to easily write programs following a *Guess & Check* approach.

2.3.1 Syntax

In the following, we suppose a language with predicate symbols having a corresponding arity (possibly 0), as well as variables and constants. By convention, variables begin with upper-case letters while predicate symbols and constants begin with lower-case letters.

Definition 2.18. Each variable and each constant is a *term*. Constants are also called *ground terms*. 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 an atom with the default negation connective "not" put in front of it. A *ground atom* (resp. *ground literal*) is an atom (resp. literal) in which only ground terms occur. □

Using these building blocks, we define the following central syntactical concept.

Definition 2.19. A *logic program* (sometimes just called "program" for short) is a set of rules which have 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 the 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*, $b^-(r) = \{b_{m+1}, \ldots, b_n\}$.

We call a rule r *safe* if each variable occurring in r is also contained in $b^+(r)$. In the following, we only allow programs where all rules are safe.

If the body of a rule r is empty, r is called a *fact*, and the $\leftarrow$ symbol can be omitted. A rule (or a program) is called *ground* if it contains only ground atoms. Note that we sometimes write

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

A rule of this form (i.e., without a head) is called an *integrity constraint* and is shorthand for

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

where a is some new atom that exists nowhere else in the program. □

The intuition behind a ground rule is the following: If we consider an answer set containing each atom from the positive body but no atom from the negative body, then the head atom must be in this answer set. An integrity constraint, i.e., a rule with an empty head, therefore expresses that a set containing each atom from the positive body but none from the negative body cannot be an answer set. Of course, we still need to define the notion of answer sets in a formal way, which we will now turn to.

2.3.2 Semantics

Since the semantics of ASP, as we will see, deals only with variable-free programs, we first require the notion of grounding a program, i.e., instantiating variables with ground terms, for which the following definitions are essential.

Definition 2.20. Given a logic program Π, the *Herbrand universe* of Π, denoted by $\mathcal{U}_\Pi$, is the set of all ground terms occurring in Π, or, if no ground terms occur, the set containing an arbitrary constant as a dummy element. The *Herbrand base* of Π, denoted by $\mathcal{B}_\Pi$, is the set of all ground atoms obtainable by using the elements of $\mathcal{U}_\Pi$ with the predicate symbols occurring in Π. The *grounding of a rule* $r \in \Pi$, denoted by $gr(r)$, is the set of rules that can be obtained by substituting all elements of $\mathcal{U}_\Pi$ for the variables in r. The *grounding of a program* Π is the ground program defined as

$$gr(\Pi) = \bigcup_{r \in \Pi} gr(r). \qquad \square$$

We now define the answer set semantics that have first been proposed in [Gelfond and Lifschitz, 1988]. To this end, we first introduce the notion of answer sets for ground programs.

Definition 2.21. Let Π be a ground logic program and I be a set of ground atoms (called an *interpretation*). A rule $r \in \Pi$ is satisfied by I if $h(r) \in I$ or $b^-(r) \cap I \neq \emptyset$ or $b^+(r) \setminus I \neq \emptyset$. I is a model of Π if it satisfies each rule in Π. We call I an *answer set* of Π if it is a subset-minimal model of the *Gelfond-Lifschitz reduct* of Π w.r.t. I, which is the program defined as

$$\Pi^I = \left\{ h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset \right\}. \qquad \square$$

Having introduced the notion of answer sets for ground programs, we can now state the answer set semantics for potentially non-ground programs by means of their groundings.

Definition 2.22. Let Π be a logic program and $I \subseteq \mathcal{B}_\Pi$. I is an *answer set* of Π if I is an answer set of $gr(\Pi)$. □

Therefore, answer sets of a program with variables can be computed by first grounding it and then solving the resulting ground program. This is mirrored in ASP systems which typically distinguish a grounding step from a subsequent solving step and can thus be divided into a *grounder* and a *solver* component.

2.3.3 Complexity and Expressive Power

Naturally, questions of computational complexity and expressive power of ASP arise. A survey of results is given in [Dantsin et al., 2001]. We will now mention results that are especially important for our purposes.

Theorem 2.23 ([Marek and Truszczyński, 1991])**.** *Deciding whether a ground logic program has an answer set is* NP*-complete.* □

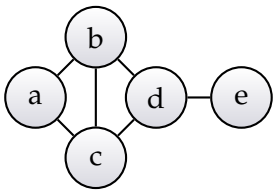
We are of course not only interested in the propositional case but also in the complexity in the presence of variables. The use of variables allows us to separate the actual program from the input data, so ASP can be seen as a query language where the (usually non-ground) program can be considered a query over a set of facts as input.

This dichotomy between the actual *program* and the *data* serving as input should be taken into account when studying the complexity of non-ground ASP. As mostly we are dealing with such situations where the program stays the same for variable data (cf. Section 2.3.4 for an example encoding for 3-Col), it is reasonable to consider the *data complexity* of ASP. By this we mean the complexity when the *program* (consisting of a set of rules with possibly non-empty bodies) is fixed whereas only the set of facts representing the *data* changes.

Theorem 2.24. *Let Π be a logic program and Δ be a set of facts. Deciding answer set existence of $\Pi \cup \Delta$ is* NP*-complete w.r.t. the size of Δ (i.e., when Π is fixed).* □

This is because when Π is fixed and only Δ varies, the size of $gr(\Pi \cup \Delta)$ is polynomial in the size of Δ.

The aforementioned complexity results give us insight into how difficult it is to solve ASP programs. A related question is which problems can actually be expressed in ASP. Informally, when we say that ASP *captures* a complexity class, it means that for any problem in that class we can write a *uniform* logic program (i.e., a single logic program that stays the same for all instances) such that this program together with a set of facts describing an instance has an answer set if and only if the instance is positive. In fact, it turned out that ASP captures NP [Schlipf, 1995], and also that a similar result holds for search problems:




```
vertex(a;b;c;d;e).
edge(a,b). edge(a,c).
edge(b,c). edge(b,d).
edge(c,d). edge(d,e).
```

Figure 2.1 A 3-Col instance and its representation in the input language of Gringo

Theorem 2.25 ([Marek and Remmel, 2003])**.** *Every search problem in* NP *can be expressed by a uniform ASP program.* □

There are various generalizations of the presented ASP syntax and semantics in the literature. For instance, allowing the use of disjunctions in rule heads (as in [Gelfond and Lifschitz, 1991]) yields higher expressiveness at the cost of Σ_2^p-completeness for the problem of deciding answer set existence for ground programs [Eiter and Gottlob, 1995, Eiter et al., 1997]. Further, the use of function symbols even leads to undecidability in general [Calimeri et al., 2008, Alviano et al., 2011]. In some of the examples in this work, we will occasionally make use of function symbols, but only in a very restricted way so that no such unpleasant effects arise. In particular, the Herbrand universe will remain finite.

2.3.4 ASP in Practice

Various systems are available [Gebser et al., 2011b, Leone et al., 2006] which proceed according to the aforementioned approach of grounding followed by solving and which offer auxiliary facilities (like aggregates and arithmetics) to make modeling easier. In this work, we use the input language of Gringo [Gebser et al., 2009, Gebser et al., 2010] in the program examples and use a `monospaced` font for typesetting rules in that language. The ← symbol corresponds to `:-` and each rule is terminated by a period. In our listings, we will perform "beautifications" such as using ← instead of `:-` and ≠ instead of `!=` for the sake of better readability.

As an introductory example, we will show how ASP can be applied to solve 3-Col (see Section 2.1 for the problem definition). Figure 2.1 depicts an instance of 3-Col together with its representation as a set of facts in the input language of Gringo. The following program solves 3-Col for instances specified in this way.

```
color(red;green;blue).
1 { map(X,C) : color(C) } 1 ← vertex(X).
← edge(X,Y), map(X,C), map(Y,C).
```

This program is to be grounded together with the facts describing the input graph using the predicates `vertex/1` and `edge/2`. The answer sets encode exactly the valid 3-colorings of the graph.

The first line uses pooling (indicated by ";") and is expanded to three facts:

```
color(red). color(green). color(blue).
```

The second line uses a cardinality constraint in the head. The ":" symbol indicates a condition on the instantiation of the variables. Conceptually, this line can be expanded as follows:

```
1 { map(X,red), map(X,green), map(X,blue) } 1 ← vertex(X).
```

The grounder will eventually expand this rule further by substituting ground terms for `X`. Roughly speaking, a cardinality constraint $l\{L_1, \ldots, L_n\}u$ is satisfied by an interpretation I iff at least l and at most u of the literals $L_1, \ldots, L_n$ are true in I. Therefore, the rule in question expresses a choice of exactly one of `map(X,red)`, `map(X,green)` and `map(X,blue)` for any vertex `X`. Finally, the integrity constraint in the third line ensures that no answer set maps the same color to adjacent vertices.

As another example, consider the SAT problem. An instance in CNF can be given by facts using the predicates `atom/1` and `clause/1`, as well as `pos`(C, A) resp. `neg`(C, A), denoting that the atom A occurs positively resp. negatively in the clause C. The following ASP program then solves the SAT problem.

```
{ true(A) : atom(A) }.
sat(C) ← pos(C,A), true(A).
sat(C) ← neg(C,A), not true(A).
← clause(C), not sat(C).
```

Note that the absence of bounds in the first rule indicates that this rule derives any subset of the atoms in the input formula as the extension of `true/1`.

2.4 Tree Decompositions

Many computationally hard problems on graphs are easy if the instance is a tree. For example, we have already seen in Section 2.2 that MINIMUM VERTEX COVER, if restricted to trees, is feasible in linear time, while the problem is intractable in general. It would of course be desirable if we could also efficiently solve instances that are "almost" trees. Fortunately, this is not entirely a pipe dream. It is indeed possible to shed some light on the seemingly so fundamental gap between trees and cyclic graphs, determine how far a graph is from being a tree and, in many cases, exploit "tree-likeness". Tree decompositions and the associated concept of treewidth provide us with powerful means to achieve this. They are also the basis for the proposed problem solving methodology – not only are tree decompositions useful for theoretical investigations, but they also serve as the structures on which the actual algorithms function.

Lately, tree decompositions and treewidth have received a great deal of attention in computer science. This interest was sparked primarily by [Robertson and Seymour, 1984]. Since then, it has been widely acknowledged that treewidth represents a very useful parameter which is applicable to a broad range of problems. There are several overviews of this topic, such as [Bodlaender, 2005, Bodlaender, 1993, Aschinger et al., 2011, Niedermeier, 2006].

2.4.1 Concepts and Complexity

Basically, a tree decomposition of a (potentially cyclic) graph is a certain kind of tree that can be obtained from the graph. From now on, to avoid ambiguity, we follow the convention that the term "vertex" refers to vertices in the original graph, whereas the term "node" refers to nodes in a tree decomposition.

To give a very rough idea, the intuition behind a tree decomposition is that each node subsumes multiple vertices, thereby isolating the parts responsible for the cyclicity. When we thus want to turn a graph into a tree, we can think of contracting vertices (ideally in a clever way) until we end up with a tree whose nodes represent subgraphs of the original graph. Our sought-for measure of a graph's cyclicity can thereby be determined as "how extensive" such contractions must be at the very least in order to get rid of all cycles. These intuitions will now be formalized.

Definition 2.26. Given a graph $G = (V, E)$, a *tree decomposition* of G is a pair (T, χ) where $T = (N, F)$ is a (rooted) tree and $\chi : N \to 2^V$ assigns to each node a set of vertices (called the node's *bag*), such that the following conditions are satisfied:

1. For every vertex $v \in V$, there exists a node $n \in N$ such that $v \in \chi(n)$.
2. For every edge $e \in E$, there exists a node $n \in N$ such that $e \subseteq \chi(n)$.
3. For every $v \in V$, the set $\{n \in N \mid v \in \chi(n)\}$ induces a connected subtree of T.

We call $\max_{n \in N} |\chi(n)| - 1$ the *width* of the decomposition. The *treewidth* of a graph is the minimum width over all its tree decompositions. □

Condition 3 is also called the *connectedness condition* and is equivalent to the requirement that if a vertex occurs in the bags of two nodes $n_0, n_1 \in N$, then it must also be contained it the bag of each node on the path between n_0 and n_1, which is uniquely determined because T is a tree.

Note that each graph admits a tree decomposition, namely at least the "decomposition" consisting of a single node n with $\chi(n) = V$. A tree has treewidth 1 and a cycle has treewidth 2. Among other interesting properties is that if a graph contains a clique $v_1, \ldots, v_k$, then in any of its tree decompositions there is a node n with $\{v_1, \ldots, v_k\} \subseteq \chi(n)$. Therefore the treewidth of a graph containing a k-clique is at least $k - 1$. Furthermore, if the graph is a $k \times k$ grid, its treewidth is k. Large cliques or grids within a graph therefore imply large treewidth.

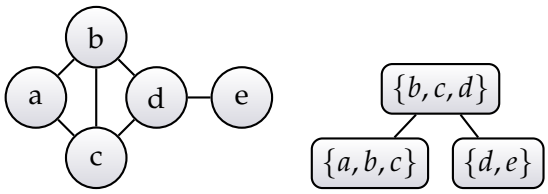


Figure 2.2 A graph with treewidth 2 and an (optimal) tree decomposition for it

Figure 2.2 shows a graph together with a tree decomposition of it that has width 2. This decomposition is optimal because the graph contains a cycle and thus its treewidth is at least 2.

Many problems that are intractable in general are tractable when the treewidth is bounded by a fixed constant. Considering treewidth as a parameter (compared to, say, solution size or the maximum clause size in a CNF formula) means to study the *structural* difficulty of instances. What makes treewidth especially attractive is that this parameter can be applied to *all* graph problems and even to many problems that do not work on graphs directly, by finding suitable graph representations of the instances. For example, we can also decompose hypergraphs by building a tree decomposition of the *primal graph* (also known as the *Gaifman graph*). Given a hypergraph $H = (V, E)$, where V are the vertices and $E \subseteq 2^V \setminus \{\emptyset\}$ are the hyperedges, the primal graph is defined as the graph $G = (V, F)$ with the same vertices as H and with the edges $F = \big\{\{x, y\} \subseteq V \mid \exists e \in E : \{x, y\} \subseteq e\big\}$; in other words, the graph where each pair of vertices appearing together in a hyperedge is connected by an edge.

Furthermore, it has been observed that instances occurring in practical situations often exhibit small treewidth (cf., e.g., [Thorup, 1998,Agarwal et al., 2011,Gramm et al., 2008,Huang and Lai, 2007,Latapy and Magnien, 2006,Melançon, 2006]). We have taken a look at some publicly available datasets[3] (from domains like co-authorship among scientists, electric power networks or biological networks) and found that indeed often the treewidth is quite small. This appears to be very promising, since it indicates that the proposed *Decompose, Guess & Check* approach might be practicable in many real-world applications because the treewidth is crucial for the runtime and memory requirements of dynamic programming algorithms on tree decompositions, as we will see in Section 2.4.3.

In general, determining a graph's treewidth and constructing an optimal tree decomposition are unfortunately intractable.

Theorem 2.27 ([Arnborg et al., 1987])**.** *Given a graph and a non-negative integer k, deciding whether the graph's treewidth is at most k is* NP*-complete.* □

However, the problem is fixed-parameter tractable w.r.t. the parameter k, i.e., if we are given a fixed k in advance, the problem becomes tractable.

[3]See, for instance, `http://wiki.gephi.org/index.php/Datasets`

Theorem 2.28 (Bodlaender's theorem [Bodlaender, 1996]). *For any fixed k, deciding whether a graph's treewidth is at most k, and, if so, constructing an optimal tree decomposition, are feasible in linear time.* □

This has important implications when we are dealing with a problem that can be efficiently solved *given* a tree decomposition of width bounded by some fixed constant k, because it means that, given k, we can also *construct* such a tree decomposition efficiently.

If no such bound on the treewidth can be given a priori, which is the case if we want to be able to process problems even if their treewidth is large, we are not necessarily doomed. Although finding an optimal tree decomposition is intractable in this case, there are efficient heuristics that produce a reasonably good tree decomposition [Bodlaender and Koster, 2010, Dermaku et al., 2008, Gottlob et al., 2002]. In practice, it is usually not necessary for the used tree decomposition to be optimal in order to take significant advantage of decomposing problem instances. In particular, having a non-optimal tree decomposition will typically imply higher runtime and memory consumption, but the optimality of the computed solution is not at stake.

2.4.2 Monadic Second-Order Logic

A famous result is due to [Courcelle, 1990]. It states that any graph property which can be expressed in the formalism of *monadic second-order logic* (MSO) can be decided in linear time when the treewidth is bounded. This allows us to identify a large class of problems that are fixed-parameter tractable w.r.t. treewidth by means of a relatively intuitive criterion. MSO enables us to express graph properties and extends classical first-order logic by allowing quantification over sets of vertices and edges. It is called *monadic* because – in contrast to general second-order logic – we may only quantify over *unary* relational variables.

In our discussion of MSO, we roughly follow [Niedermeier, 2006] and [Flum and Grohe, 2006]. First, we define the syntax. Beside the usual symbols familiar from first-order logic (logical connectives, quantifiers, parentheses and the equality symbol), our alphabet contains countably many individual variables (denoted by lower-case letters) and unary relational variables (i.e., set variables, denoted by upper-case letters), as well as the unary relational symbols V and E, and the binary relational symbol I.

In the following, let X by an arbitrary set variable and x, y be arbitrary individual variables. The set of MSO formulas is now defined as the smallest set satisfying the following conditions. $x = y$, $\mathrm{V}(x)$, $\mathrm{E}(x)$, $\mathrm{I}(x, y)$ and $X(x)$ are (atomic) formulas. If ϕ and ψ are formulas, then $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \supset \psi$, $\exists x\phi$, $\forall x\phi$, $\exists X\phi$ and $\forall X\phi$ are formulas, too.

Now we define the semantics of MSO. For a graph $G = (V, E)$ whose vertices and edges we consider our domain of discourse, let α denote an assignment such that for each individual variable x it holds that $\alpha(x) \in V \cup E$, and for each set variable X it

holds that $\alpha(X) \subseteq V \cup E$. The satisfaction relation is defined as follows. The semantics for equality, first-order quantifiers and the logical connectives is no different from first-order logic. For the other possible formulas, the following equivalences hold:

- $(G, \alpha) \models \mathrm{V}(x)$ iff $\alpha(x) \in V$
- $(G, \alpha) \models \mathrm{E}(x)$ iff $\alpha(x) \in E$
- $(G, \alpha) \models \mathrm{I}(x, y)$ iff $\alpha(x) \in V$, $\alpha(y) \in E$, and $\alpha(x)$ is an endpoint of $\alpha(y)$
- $(G, \alpha) \models X(x)$ iff $\alpha(x) \in \alpha(X)$
- $(G, \alpha) \models \exists X \phi$ iff $(G, \alpha') \models \phi$ for some $\alpha' \overset{X}{\sim} \alpha$ (i.e., $\alpha'(X)$ is an arbitrary subset of $V \cup E$ and α' coincides with α on all arguments distinct from X)
- $(G, \alpha) \models \forall X \phi$ iff $(G, \alpha') \models \phi$ for all $\alpha' \overset{X}{\sim} \alpha$

We simply write $G \models \phi$ iff $(G, \alpha) \models \phi$ for all α. This is useful, in particular, if the formula we are dealing with is a *sentence* (i.e., there are no free variables), as it then makes no difference which assignment we consider – solely the graph determines the sentence's truth value.

Theorem 2.29 (Courcelle's theorem [Courcelle, 1990])**.** *Let ϕ be a fixed MSO sentence and k be a fixed non-negative integer. Given a graph G with treewidth at most k, there is a linear-time algorithm deciding whether $G \models \phi$.* □

This theorem is central because it means that deciding whether a property holds for a graph is fixed-parameter tractable w.r.t. treewidth if the property is expressible in MSO.

In the following, we suppose loop-free graphs and use the abbreviation $\mathrm{adj}(x, y)$ for the formula $\exists e\big(I(x, e) \wedge I(y, e)\big)$. As an example, recall the 3-Col problem (cf. Section 2.1). The following MSO sentence encodes the property of being 3-colorable.

$$\exists X_0 \exists X_1 \exists X_2 \left(\mathrm{Part}(X_0, X_1, X_2) \wedge \forall x \forall y \Big(\mathrm{adj}(x, y) \supset \bigwedge_{0 \leq i < 3} \neg\big(X_i(x) \wedge X_i(y)\big) \Big) \right)$$

The subformula $\mathrm{Part}(X_0, X_1, X_2)$ expresses that X_0, X_1, X_2 form a partition of the vertices and is defined as follows.

$$\mathrm{Part}(X_0, X_1, X_2) := \forall v \left(V(v) \supset \left(\Big(\bigvee_{0 \leq i < 3} X_i(v) \Big) \wedge \bigwedge_{0 \leq i < j < 3} \neg\big(X_i(v) \wedge X_j(v)\big) \right) \right)$$

From this, we can conclude that 3-Col is fixed-parameter tractable w.r.t. treewidth.

There are also extensions of MSO that offer a handle on other problem types. For instance, without going into details about such an extension for optimization problems (see [Arnborg et al., 1991]), we can show Minimum Vertex Cover to be fixed-parameter tractable w.r.t. treewidth by stating the sentence

$$\min X \, \forall x \forall y \Big(\text{adj}(x, y) \supset \big(X(x) \vee X(y) \big) \Big).$$

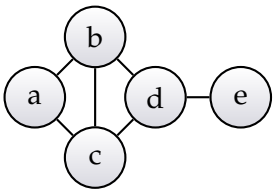
As we have seen in Section 2.2, the problem is tractable when restricted to trees. Because of the expressibility in MSO, it turns out that Minimum Vertex Cover is even fixed-parameter tractable w.r.t. treewidth. This is not a lucky isolated case – in fact, it has been observed that many problems that are easy to solve on trees share this convenient property.

2.4.3 Dynamic Programming on Tree Decompositions

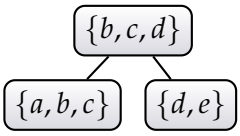
Although stating a problem in MSO not only establishes fixed-parameter tractability w.r.t. treewidth according to Courcelle's theorem, but can also be used to obtain an algorithm, such an algorithm is impractical in real-world settings because its runtime has huge constant factors [Niedermeier, 2006]. Special-tailored algorithms that work directly on tree decompositions are mostly employed in practice instead. Nevertheless, if we are not sure whether a particular problem is fixed-parameter tractable w.r.t. treewidth at all, it might be wise to first try to formulate the problem as an MSO formula and *then* try to come up with a proper algorithm.

Figure 2.3 shows how dynamic programming can be applied to a tree decomposition of a 3-Col instance. Each of the tree decomposition nodes in Figure 2.3b has a corresponding table in Figure 2.3c where there is a column for each bag element. Additionally, we have a column i that is used to store an identifier for each row such that an entry in the column j of a potential parent table can refer to the respective row. Eventually, each row will describe a valid 3-coloring of the subproblem represented by the bag.

Adhering to the approach of dynamic programming, the tables in Figure 2.3c are computed bottom-up. First all valid 3-colorings for the leaf bags are constructed and stored in the respective table. For each non-leaf node with already computed child tables, we then look at all combinations of child rows and combine those rows that coincide on the colors of common bag elements; that is to say we *join* the rows. In the example, the leaves have no common bag elements, therefore each pair of child rows joins. However, we must eliminate all results of the join that violate a constraint, i.e., where adjacent vertices have the same color. For instance, the combination of row 0 from the left child with row 2 from the right child is invalid because the adjacent vertices b and d are colored with "g"; the left row 0 combined with the right row 0 is valid, however, and gives rise to the row 3 in the root table. We store the identifiers of these child rows as a pair in the j column. Note that the entry of j in row 3 not only



(a) A 3-Col instance



(b) A tree decomposition of the instance

i	b	c	d	j
0	r	g	b	$\{(4,4),(4,5)\}$
1	r	b	g	$\{(2,2),(2,3)\}$
2	g	r	b	$\{(5,4),(5,5)\}$
3	g	b	r	$\{(0,0),(0,1)\}$
4	b	r	g	$\{(3,2),(3,3)\}$
5	b	g	r	$\{(1,0),(1,1)\}$

i	a	b	c
0	r	g	b
1	r	b	g
2	g	r	b
3	g	b	r
4	b	r	g
5	b	g	r

i	d	e
0	r	g
1	r	b
2	g	r
3	g	b
4	b	r
5	b	g

(c) The computed dynamic programming tables

Figure 2.3 Dynamic programming for 3-Col on a tree decomposition

contains $(0,0)$ but also $(0,1)$ because joining these rows produces the same row as we project onto the current bag elements b, c and d. Storing *all* predecessors of a row like this allows us to enumerate *all* 3-colorings with a final top-down traversal.

At any instant during the progress of a dynamic programming algorithm, the vertices in the current bag (i.e., the bag of the node whose table the algorithm currently computes) are called the *current* vertices. Current vertices that are not contained in any child node's bag are also called *introduced* vertices, whereas we call the vertices in a child node's bag that are no longer in the current bag *removed* vertices. Usually, a dynamic programming algorithm must not only decide which child rows to join but also how to extend partial solutions, represented by child rows, to account for the introduced vertices. In the case of 3-Col, we would simply guess a color for each introduced vertex such that no adjacent vertices have the same color. In the example, this only happens in the leaves.

Note that this algorithm's space and time requirements are both exponential in the decomposition width. However, when the treewidth can be considered bounded, this algorithm runs in linear space and time. This proves fixed-parameter tractability of 3-Col parameterized by the treewidth. It is a general property of the algorithms presented in this work that the width of the obtained decompositions is crucial for the performance.

Chapter 3

Decompose, Guess & Check

In this chapter, we introduce the problem solving technique we call *Decompose, Guess & Check*, which combines the concept of dynamic programming on tree decompositions with the *Guess & Check* method found in ASP.

Section 3.1 describes the general *Decompose, Guess & Check* approach, including a motivation and desiderata for a general tool implementing it. In Section 3.2, we present a guideline for designing dynamic programming algorithms on tree decompositions. Section 3.3 is concerned with the applicability of our approach; in particular we show that the MSO formula evaluation problem can be solved efficiently with *Decompose, Guess & Check*.

3.1 General Approach

Using ASP to solve problems via *Guess & Check* is not breaking news, and neither is dynamic programming on tree decompositions. We believe, however, that a combination of both can be very beneficial and deserves being studied on its own as an advanced approach to declarative problem solving.

It is useful to contrast the traditional way of using ASP for problem solving on the one hand with the proposed kind of strategy on the other hand. We therefore distinguish the paradigms of *monolithic* and *decomposed* problem solving.

On the one hand, by *monolithic problem solving* we denote the traditional strategy of solving an entire problem instance as a whole by writing an ASP program adhering to the *Guess & Check* principle. In general, this leads to exponential runtime.

Decomposed problem solving, on the other hand, denotes the strategy of not solving the whole instance in one piece but rather dividing it into smaller parts, for each of which *Guess & Check* is then employed locally such that the solutions of these parts can, in the end, be combined to obtain a global solution. In principle, it is thus possible to obtain a tractable algorithm provided the individual parts are sufficiently small. Even if they are not, it might be the case that a decomposition into, say, two parts has

advantages: If we can decompose an instance for which a monolithic program has a runtime of 2^n into two parts for which we "only" have a runtime of $2^{\frac{n}{2}+o}$ respectively, where o denotes some (hopefully small) overlap, then a much wider range of instances can be solved in practice.

We would like to propose the decomposed problem solving paradigm as a promising element in the toolbox of a computer scientist trying to deal with computationally hard problems. We have therefore implemented a framework that enables ASP programmers to easily take advantage of decomposing problem instances. In many cases, it can be used to achieve (fixed-parameter) tractability. But it should be kept in mind that *Decompose, Guess & Check* can also be appealing if it does not lead to tractability.

We will first give a motivation for a *Decompose, Guess & Check* framework in Section 3.1.1 and then give an extensive outline of our proposed method in general, as well as desiderata for a framework that implements it, in Section 3.1.2. Section 3.2 presents guidelines for designing dynamic programming algorithms on tree decompositions and Section 3.3 finally discusses what kinds of problems can be solved with such an approach.

3.1.1 Motivation

First, we would like to motivate why it makes sense to provide a framework for dynamic programming on tree decompositions using ASP.

ASP is generally appreciated as a language that allows for specifying succinct programs for generally intractable problems. Especially its declarative nature makes ASP code often quite readable and maintainable. In ASP, it is very natural to write programs that conceptually perform non-deterministic choices. Therefore, ASP is particularly well suited for NP-complete problems because of the characterization of NP via *Guess & Check* (cf. Section 2.1.2). Further, there are very efficient solvers available that offer a rich syntax to specify problems easily.

The question arises why it is ASP that we propose for dynamic programming on tree decompositions. In general, what we end up with when decomposing a problem instance is a collection of subproblems that are smaller than the original instance but can usually be tackled by locally using a problem solving methodology similar to the one used in a monolithic setting. That is, solving a subproblem is conceptually no easier than solving the whole problem instance – it is just the runtime that is smaller for a subproblem as only a part of the instance is considered. Since ASP is well suited for a lot of problems, it is often also well suited for parts of such problems, and we can benefit from its advantages also in decomposed problem solving.

While it is often a good idea to employ ASP for the local processing of subproblems, dynamic programming on tree decompositions of course demands more than just the problem-specific code. In particular, at least the following components are needed:

- For constructing a tree decomposition, it is first of all necessary to parse some

input that describes the graph that is to be decomposed.

- This graph has to be specified in some format and, after parsing it, needs to be stored in suitable data structures.
- Once the graph is represented in memory, we must construct a tree decomposition of it, ideally with small width.
- In order to perform the actual dynamic programming, we need to associate a table with each node and provide efficient data structures that can store the rows.
- Finally, boilerplate code is required for the tree traversal and the data flow, before the *actual* problem-specific dynamic programming algorithm is called.

What is actually of interest is just the last one of these, i.e., the dynamic programming algorithm. Each of the other tasks only becomes interesting when we are fine-tuning for efficiency and is otherwise considered as a burden that has nothing to do with the actual problem at hand. Hence, a tool that takes care of these parts is appreciated.

3.1.2 General Outline and Desiderata

We would now like to turn to the question of how *Decompose, Guess & Check* works in general. The answers to this question determine what is expected from a framework supporting *Decompose, Guess & Check* and are thus the guiding principles for the implementation we will present in Chapter 4.

In Section 2.4, we have seen that tree decompositions are a powerful means to decompose a wide range of problems and to solve many of them efficiently with dynamic programming. Tree decompositions are therefore the central objects in our approach.

We would like to stress that our method is not restricted to graph problems. Also more abstract instances can be decomposed when a reasonable *graph representation* can be found (for example, see Section 4.2.5).

The big picture of our approach is that each tree decomposition node is associated with a *table*. These tables will eventually contain *rows* that correspond to (partial) solutions. The computation of the tables proceeds in a bottom-up way by executing a user-provided, problem-specific ASP program for each node, having access to the already computed child tables.

Normalizations of Tree Decompositions

In the literature, algorithms for dynamic programming on tree decompositions can often be found to use a restricted class of decompositions, viz. normalized or semi-

normalized tree decompositions.[1] A *semi-normalized tree decomposition* is a tree decomposition where each node falls under one of the following types:

- *Leaf nodes* having no children
- *Exchange nodes* having one child
- *Join nodes* having two children with bags equal to the join node's bag

A *normalized tree decomposition* is a semi-normalized tree decomposition where the bags of adjacent nodes differ in at most one element.

Although such normalizations can be obtained from a given tree decomposition in linear time [Kloks, 1994] and facilitate some algorithms and correctness proofs because conceptually different operations of dynamic programming algorithms are separated, the restrictions they impose might not be necessary for a practical algorithm, especially if it is written in ASP. The advantages normalized tree decompositions bring (compared to non-normalized ones) for implementations in imperative languages are not useful here due to the natural modeling of non-determinism in ASP. Normalization can even affect performance adversely because more nodes must be processed and more data must be shoved around overall. In light of this, semi-normalizations might be beneficial. However, our case studies (cf. Section 4.2) have shown that algorithms that do not even require *any* distinction of different node types are not more difficult when compared to their more restricted counterparts. In fact, ASP allows us to easily combine *join* and *exchange* parts into a single program, and a separation of those is therefore often perceived to be artificial. This is the reason we favor general (i.e., non-normalized) tree decompositions in the presentation of our approach. Note that our framework offers the discussed normalizations as an optional feature to allow also implementing traditional algorithms, for instance the ones in [Bliem et al., 2012], more directly.

Problem Types

Although, from a theoretical point of view, decision problems are the primary objects of study, in practice it is often not very satisfactory to only obtain "yes" or "no" answers. For instance, when dealing with a problem like 3-Col, we might be primarily interested in actual 3-colorings of a graph rather than just knowing whether one exists. Therefore, a framework for *Decompose, Guess & Check* is expected not only to decide problems but also to provide features for, e.g., solution enumeration, counting and optimization. In this work, we will discuss the following problem types:

- *Decision problems* that require deciding if a solution exists

[1] Normalized tree decompositions are sometimes also called *nice*. There is also the concept of *semi-nice tree decompositions* [Dorn and Telle, 2009] which are, however, different from our semi-normalized ones and will not be covered in this work.

- *Counting problems* that require counting the number of solutions
- *Enumeration problems* that require printing all solutions

For each of these types we also consider optimization variants:

- Determining the *minimum cost* among all solutions (or reporting that none exist)
- *Counting* the number of *optimal solutions*
- *Enumerating* all *optimal solutions*

Technically it suffices to only consider optimization problems in an implementation since the non-optimization variants can be seen as special cases where all solutions have equal costs. Also note that minimizing the cost of solutions also allows for maximization problems by, e.g., using negative numbers.

Depending on the problem type, the rows in the tables of the tree decomposition nodes have different structure. For instance, when solving an optimization problem, rows must contain solution costs whereas this is not required for other problem types.

Table Rows

Each table row corresponds to a set of *partial solutions*.[2] A partial solution can be thought of as that part of a global solution that we obtain by disregarding all information that does not apply to the subgraph considered so far. More precisely, we restrict ourselves to the information about the subgraph induced by the vertices that are contained in any bag of the tree decomposition's subtree rooted at the current node. Thus, each row in the root table corresponds to a set of global solutions.

It is important to realize that in general a table row can represent more than one partial solution. This is because a row normally only contains information concerned with the current bag elements. For instance, when a vertex has been removed (i.e., is present in a child node's bag but not in the current bag), partial solutions from that child node are subsequently merged when they coincide on the information about the remaining bag elements. The single resulting table row thus represents all of the merged partial solutions that it extends.

What is contained in a table row obviously depends on the particular problem. For many problems, a table row contains a subset of the vertices (see, e.g., Sections 4.2.4 and 4.2.5). Frequently, also a mapping of values to vertices is used (see, e.g., Section 4.2.1) and often we even need more complicated structures (see, e.g., Sections 4.2.8 and 4.2.9). Hence, a tool for *Decompose, Guess & Check* must offer great flexibility regarding the structure of table rows in order to restrict the user's freedom of designing algorithms as little as possible.

[2] Actually it would be more appropriate to speak of partial solution *candidates* since a row can also represent an object that will not lead to a solution. However, to keep the terminology simpler and in accordance with [Bodlaender, 1997], we use the term "partial solution".

We therefore propose a table row to contain a set of arbitrary *items*. An item can be any ground term the user pleases. By means of terms containing uninterpreted function symbols, the user can even store structured data in an item. For instance, if we have a graph with the vertices x, y, z, the item set $\{x, y\}$ could be interpreted as a subset of the vertices; $\{\mathrm{e}(x, y),\ \mathrm{e}(y, z)\}$ could represent a subset of the edges; and $\{\mathrm{map}(x, \mathrm{red}),\ \mathrm{map}(y, \mathrm{green})\}$ could be used for a coloring of vertices.

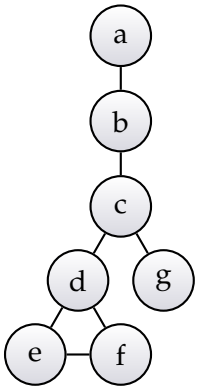
We also call the item set in a table row the *characteristic* of that row, in the style of the terminology in [Bodlaender, 1997]. As noted above, a table row contains *additional information* beside its characteristic, depending on the problem type.

- For *decision problems*, no additional information is stored.
- In algorithms for *counting problems*, a table row contains the number of partial solutions it represents.
- When solving *enumeration problems*, each table row r of a node that has n children contains a set of n-tuples of *extension pointers*, where the ith element of any such tuple references a row in the ith child table that has given rise to r. These pointers will be followed when materializing solutions.
- In order to determine the *minimum cost* among all solutions, a table row contains the cost of the cheapest partial solution it represents.
- For *counting optimal solutions*, a row is required to contain the minimum cost of all the partial solutions it represents, as well as the number of represented partial solutions having that cost.
- To *enumerate optimal solutions*, we need to store again the minimum cost of the represented partial solutions and, as for enumeration problems without optimization, a set of extension pointers – however, only such combinations of child rows may be in this set that (when combined with the current table row) actually yield a partial solution with that optimum cost.

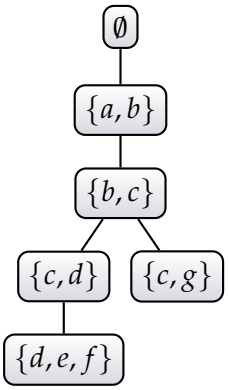
In Figure 3.1, the presented notions are illustrated by means of an example for the Minimum Vertex Cover problem. For an actual algorithm that proceeds like this, see Section 4.2.4. The tree decomposition in Figure 3.1b exhibits an empty root for reasons we will discuss shortly.

In the tables of Figure 3.1c, the column i contains an identifier for each row that can be used in the extension pointers in potential parent tables. The item sets constituting the characteristics of the rows are written in the column "items".

The partial solution depicted in Figure 3.1d is represented by row 0 of the table for the node with the bag $\{c, d\}$. Vertices highlighted in red are contained in a corresponding optimal solution. The parts that are drawn with solid lines constitute the partial solution. In subsequent steps of the algorithm, this partial solution would be



(a) A Minimum Vertex Cover instance



(b) A tree decomposition of the instance with an empty root

i	items	cost	extend
0	$\emptyset$	4	$\{(0),(1)\}$

i	items	cost	extend
0	$\{a\}$	4	$\{(1)\}$
1	$\{b\}$	4	$\{(0),(2)\}$
2	$\{a,b\}$	5	$\{(0),(2)\}$

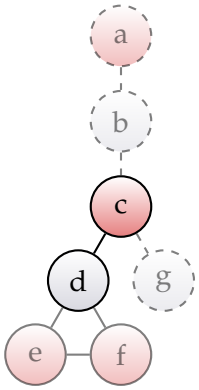
i	items	cost	extend
0	$\{b\}$	4	$\{(1,1)\}$
1	$\{c\}$	3	$\{(0,0),(2,0)\}$
2	$\{b,c\}$	4	$\{(0,0),(2,0)\}$

i	items	cost	extend
0	$\{c\}$	3	$\{(2)\}$
1	$\{d\}$	2	$\{(0),(1)\}$
2	$\{c,d\}$	3	$\{(0),(1)\}$

i	items	cost	extend
0	$\{c\}$	1	$\emptyset$
1	$\{g\}$	1	$\emptyset$
2	$\{c,g\}$	2	$\emptyset$

i	items	cost	extend
0	$\{d,e\}$	2	$\emptyset$
1	$\{d,f\}$	2	$\emptyset$
2	$\{e,f\}$	2	$\emptyset$
3	$\{d,e,f\}$	3	$\emptyset$

(c) The dynamic programming tables



(d) Visualization of a partial solution

Figure 3.1 Dynamic programming for Minimum Vertex Cover

extended to the parts that are drawn with dashed lines. Note that the vertices e and f have been removed from the current bag and are therefore shaded.

In Section 3.1.3, we will generalize the presented notion of a characteristic to allow also treating problems higher in the polynomial hierarchy than NP in a natural way. Until then, for the sake of clarity, we concentrate on this simpler special case because most problems of practical interest can be treated in this way, and the generalization does not invalidate the considerations presented here.

Computing the Tables

In order to compute the table of the current tree decomposition node during a bottom-up traversal by dynamic programming, the *Decompose, Guess & Check* approach utilizes ASP. The program that the user has provided for this purpose is combined with the problem instance and a description of the current node, its children and the child tables. The resulting answer sets are then used to populate the current table. To this end, the answer sets are expected to contain predefined predicates that, for instance, signal which items are to be filled into a table row's characteristic. A tool implementing the *Decompose, Guess & Check* approach must then take care of scanning the answer sets for these special predicates and subsequently storing appropriate rows in the current table.

A common workflow in the user-supplied programs that mirrors the intuition of bottom-up computation is to *guess* a combination of child table rows – one row for each child node – and *check* whether the guessed combination gives rise to a valid partial solution. If so, an answer set is returned that specifies an item set which is typically obtained by restricting the item sets of those preceding child rows to information about the current bag elements, and perhaps extending the resulting unified item set with information about introduced bag elements (i.e., current vertices that have been in no child node's bag).

In an algorithm like this, it may happen that several combinations of child rows lead to the same characteristic because of the merging of partial solutions due to reasons as discussed above. In such a case, multiple answer sets are returned that only differ in the guessed child rows but not in their item sets. A framework for *Decompose, Guess & Check* must in such a case by no means store a new table row for each answer set, for that would lead to a disastrous explosion of memory and render the whole decomposition pointless. Rather, the advisable course of action would be to only create a row for each new item set and merge all answer sets encoding that same item set in a way described next.

Hence, a proper *Decompose, Guess & Check* tool ensures that a table never contains more than one row with the same characteristic. For a given table, a characteristic thus always uniquely determines a row.

From Answer Sets to Table Rows

In the case where each characteristic consists of a single item set as presented so far – for the more general case, see Section 3.1.3 – obtaining the table rows from the answer sets is straightforward: Each answer set encodes a complete characteristic together with additional information (like a tuple of extension pointers, a count or a cost). Therefore, when an answer set is reported that encodes a characteristic not already stored in any row, a *Decompose, Guess & Check* tool inserts a new row into the table and fills it with the encoded characteristic and additional information. When another answer set arrives specifying the same characteristic, the tool looks up the existing table row and performs the following actions for the additional information:

1. If the answer set encodes a cost that exceeds the cost in the stored row, the answer set is discarded because there are already better partial solutions for that characteristic.

2. If the encoded cost is lower than the stored one, the old contents of the row are thrown away and replaced by the information contained in that answer set.

3. Otherwise, if the encoded cost equals the stored one (or if no optimization problem is solved and there are therefore no costs), the following actions are taken:

   (a) If the problem type demands enumeration of solutions, the tool inserts the encoded tuple of extension pointers into that row's set of such tuples.

   (b) If the problem type demands counting, the tool adds the encoded solution count to the stored count.

Post-Processing

For some problems it is necessary to perform final actions once all tables have been computed. For instance, it might turn out that some rows do not correspond to solutions because they lack some property. Often we can "kill" table rows without a crucial property right away, but sometimes we need to keep them around because that property could become satisfied after all (when that row gets extended) during processing an ancestor node later on.

It is often the case that rows are removed when some vertex that, accord to these rows, lacks a critical property disappears from the respective current bag – indicating that that vertex will never reappear due to the connectedness condition and that it has appeared together in some already encountered bag with each of its neighbors.

Hence, a row not corresponding to a solution can sometimes only be killed when vertices in the bag associated with this row are removed. For cases like this, it is reasonable to use tree decompositions with an empty root node. Having empty roots even enables us to perform additional post-processing steps that are different from

what happens when vertices are removed and thus *only* occur at the end (e.g., in the examples of Section 4.2.3 and 4.2.6).

Notice, for example, that in the root table in Figure 3.1c the child row 2 gets killed, i.e., does not give rise to an extension. This is because its cost is not minimal. However, it needs to be present in the table below the root, as it might be that the other rows in the end would lead to a higher cost because of vertices not considered yet. After all, at each step, the algorithm only knows about the subtree rooted at the current node. With an empty root, it can be ensured that only rows corresponding to valid and optimal solutions are contained in the root table.

Killing "bad" rows can, as in this example, be due to suboptimal cost, but it can also be because of problem-specific properties that – in contrast to the "cost" field of a row – a *Decompose, Guess & Check* framework is ignorant about (cf., e.g., Section 4.2.5). In the latter case, the user's program usually contains integrity constraints that disallow extending bad child rows.

If there are rows in the root table, it is ensured that the solution count, cost and extension pointers only refer to optimal solutions because suboptimal solutions are discarded when merging rows with equal characteristics as described above.

Materializing the Solutions

When all tables have been computed, the result of the computation should be reported. Which form this takes of course depends on the problem type.

- For a *decision problem*, we either report "yes" or "no" depending on whether there are rows in the root table or not.
- When dealing with a *counting problem*, we inspect the counts in the root table rows and thus print their sum as the total number of solutions.
- Given an *enumeration problem*, complete solutions can be constructed by recursively following the extension pointers in the root table rows. A global solution can be assembled by unifying the item sets of all rows that can be reached using these pointers.
- For determining the *optimum value* of a solution, we return the minimum cost of any row in the root table, or "no" if the table is empty.
- *Counting optimal solutions* is performed by summing the solution counts of only those root table rows that have minimal cost.
- *Enumerating optimal solutions* proceeds like enumerating all solutions but only performs the materialization for root table rows that have minimal cost. Because the *Decompose, Guess & Check* tool discards extension pointers that would give rise to suboptimal solutions during the construction of the tables as discussed above, it is ensured that thereby only optimal solutions are materialized.

The algorithm must, of course, provide the information that is required for the respective problem type (like counts, extension pointers or costs).

For an example of enumerating all optimal solutions, consider Figure 3.1c. There is only one row in the root table with an additional cost information of 4. That is, all optimal solutions contain 4 vertices. We illustrate the construction of the solution $\{a, c, e, f\}$ (depicted in Figure 3.1d). By recursively following always the first tuple of extension pointers that is contained in the respective "extend" row and unifying the item sets, we obtain the desired solution

$$\emptyset \cup \{a\} \cup \{c\} \cup \{c\} \cup \{c\} \cup \{e, f\} = \{a, c, e, f\}.$$

In a similar way, by following the other extension pointers, we can materialize all remaining solutions.

3.1.3 Requirements for Problems Beyond NP

To make our approach more flexible, we generalize the basic notions introduced in Section 3.1.2. This is useful for some more involved cases, but for the majority of practical problems the additional flexibility brought by the considerations in this section is not necessary to construct appropriate *Decompose, Guess & Check* algorithms.

Multi-Level Characteristics

Algorithms on tree decompositions for problems higher in the polynomial hierarchy than NP usually require a more involved table row structure than presented thus far, due to the polynomial hierarchy's characterization by quantifier alternation (cf. Section 2.1.3). For instance, [Jakl et al., 2009] provide an algorithm (cf. also Section 4.2.6 for a related implementation) for a problem on the second level of the polynomial hierarchy where each row can be associated with so-called certificates, i.e., information that may eventually witness that this row does not represent a solution.

In order to allow handling also such problems in a natural way, we generalize the notion of a table row's characteristic as follows. Any item set can possess arbitrarily many *subsidiary item sets*, each of which can also have an associated set of extension pointer tuples. This can be recursively utilized to obtain a *tree of item sets* within a single table row. In the use cases we have considered so far, these additional levels of item sets can be used to associate auxiliary information with the top-level item set.

To illustrate how such multi-level characteristics can, for instance, be employed when dealing with a problem on, say, the second level of the polynomial hierarchy, suppose the current tree decomposition node during a run of our dynamic programming algorithm is called n and we want to know if *there is* a set of objects A such that *for all* sets of objects B some property ϕ holds. The top-level item set in a table row of n could contain a guess of A, and its subsidiary item sets represent all choices for B. This way, when processing the parent node of n, that child table row can be decided

to be either extended or not, depending on whether ϕ holds *for all* combinations of the top-level item set with a subsidiary item set. We will shortly see an illustration of multi-level characteristics for a concrete example.

It is also conceivable to use more than one level of item sets even for problems in NP. Although the need for multi-level characteristics arose from the desire to handle problems higher in the polynomial hierarchy, using them for mere convenience of modeling can sometimes also be beneficial even if the problem is in NP.

For many problems, multi-level characteristics are not required and a characteristic is just a single (top-level) item set as proposed in Section 3.1.2. In such cases, we use the terms "characteristic" and "item set" interchangeably. The reader should keep in mind, however, that more complicated structures are possible.

The Minimum 3-Coloring Problem

As an example to make this more concrete, consider the following problem, which we call Minimum 3-Coloring. In contrast to 3-Col, here we call a subset of the vertices the "critical vertices" and we are only interested in those 3-colorings of the graph where the set of all critical vertices that are colored with a certain color – say, red – is subset-minimal among all proper colorings. Formally, we call a 3-coloring f *red-minimal* w.r.t. the critical vertices W if and only if there is no 3-coloring f' of the same graph such that $\{v \in W \mid f'(v) = \text{red}\} \subset \{v \in W \mid f(x) = \text{red}\}$. Minimum 3-Coloring is now defined as the following problem:

Input: A graph $G = (V, E)$, a set $W \subseteq V$ and a vertex $q \in W$
Question: Is q colored with "red" in a red-minimal 3-coloring of G w.r.t. W?

We are not aware of published complexity results for this problem. Therefore, we briefly prove its Σ_2^p-completeness as an interlude before subsequently presenting an algorithm that illustrates the use of multi-level characteristics.

Theorem 3.1. Minimum 3-Coloring *is* Σ_2^p*-complete.* □

Proof. As for membership, we can guess a mapping from the vertices to the three colors, check if this is a 3-coloring, and then, via an oracle for co-NP, check if there is no smaller 3-coloring, i.e., if there is no 3-coloring whose red vertices from W form a proper subset of the elements of W that the originally guessed coloring assigned "red".

To show hardness, we proceed by reduction from the Σ_2^p-complete problem Minimal Model Sat [Eiter and Gottlob, 1993] that asks, given a propositional formula ϕ (which we assume to be in 3-CNF) and an atom q, whether q is contained in a subset-minimal model of ϕ. Our construction takes ideas from the standard reduction from 3-Sat to 3-Col (see, e.g., [Goldreich, 2008]).

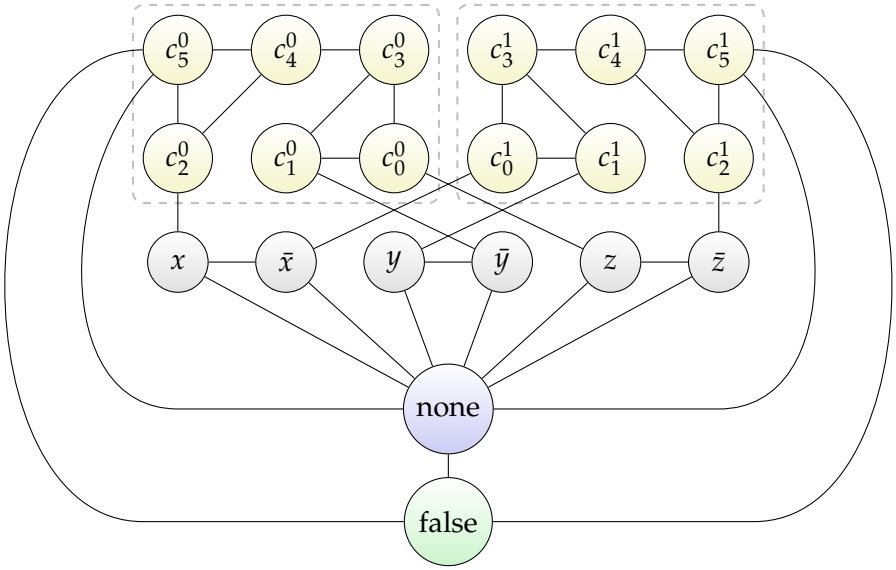


Figure 3.2 Reduction from MINIMAL MODEL SAT to MINIMUM 3-COLORING for the formula $(z \vee \neg y \vee x) \wedge (\neg x \vee y \vee \neg z)$

Let (ϕ, q) be an instance of MINIMAL MODEL SAT, $\mathrm{cl}(\phi)$ denote the set of clauses in ϕ and $\mathrm{at}(\phi)$ denote the set of atoms in ϕ. We define $R(\phi, q) = (V, E, W, q)$ to be an instance of MINIMUM 3-COLORING as follows.

$$
\begin{aligned}
V &= \{\mathsf{false},\ \mathsf{none}\} \cup \{x,\ \bar{x} \mid x \in \mathrm{at}(\phi)\} \cup \{c_i \mid c \in \mathrm{cl}(\phi),\ 0 \leq i < 6\} \\
E &= \{(\mathsf{false}, \mathsf{none})\} \\
&\quad \cup \big\{(x, \bar{x}),\ (x, \mathsf{none}),\ (\bar{x}, \mathsf{none}) \mid x \in \mathrm{at}(\phi)\big\} \\
&\quad \cup \big\{(l_0, c_0), (l_1, c_1), (l_2, c_2) \mid c \in \mathrm{cl}(\phi),\ c = (l_0, l_1, l_2)\big\} \\
&\quad \cup \big\{(c_0, c_1),\ (c_0, c_3),\ (c_1, c_3),\ (c_2, c_4),\ (c_2, c_5),\ (c_3, c_4),\ (c_4, c_5) \mid c \in \mathrm{cl}(\phi)\big\} \\
&\quad \cup \big\{(c_5, \mathsf{none}),\ (c_5, \mathsf{false}) \mid c \in \mathrm{cl}(\phi)\big\} \\
W &= \mathrm{at}(\phi) \cup \{\mathsf{false}, \mathsf{none}\}
\end{aligned}
$$

Figure 3.2 depicts an example.

We now show correctness of this reduction by proving that (ϕ, q) is a positive instance of MINIMAL MODEL SAT if and only if $R(\phi, q)$ is a positive instance of MINIMUM 3-COLORING.

"Only if" direction: Let ϕ be a formula in 3-CNF and $M \subseteq \mathrm{at}(\phi)$ be a subset-minimal model of ϕ containing the variable q. From this we construct a coloring f appropriate to $R(\phi, q)$ as follows. First, we define $f(\mathsf{false}) =$ green and $f(\mathsf{none}) =$ blue. For any

$x \in \text{at}(\phi)$, we set

$$f(x) = \begin{cases} \text{red} & \text{if } x \in M \\ \text{green} & \text{otherwise,} \end{cases} \qquad f(\bar{x}) = \begin{cases} \text{red} & \text{if } x \notin M \\ \text{green} & \text{otherwise.} \end{cases}$$

For any clause $c = (l_0, l_1, l_2)$, we first set $f(c_5) = \text{red}$. Observe that at least one literal must be true under M, i.e., $M \models l_i$ for some $0 \leq i < 3$, so the following case distinction is exhaustive.

1. If $M \models l_0$, then we set $f(c_0) = f(c_4) = \text{green}$, $f(c_3) = \text{red}$ and $f(c_1) = f(c_2) = \text{blue}$.
2. If $M \not\models l_0$ but $M \models l_1$, then we set $f(c_1) = f(c_4) = \text{green}$, $f(c_3) = \text{red}$ and $f(c_0) = f(c_2) = \text{blue}$.
3. Otherwise $M \not\models l_0$, $M \not\models l_1$ and $M \models l_2$ hold, and we set $f(c_2) = f(c_3) = \text{green}$, $f(c_0) = \text{red}$ and $f(c_1) = f(c_4) = \text{blue}$.

We will show that f is a red-minimal coloring of $R(\phi, q)$ w.r.t. W in which q is colored with red. $f(q) = \text{red}$ is obvious. It can also easily be seen that for each $(a, b) \in E$ it holds that $f(a) \neq f(b)$ by construction of f.

It remains to show that f is red-minimal w.r.t. W. Suppose to the contrary that there is a smaller coloring f', i.e., $\{x \in W \mid f'(x) = \text{red}\} \subset \{x \in W \mid f(x) = \text{red}\}$. From this we construct an interpretation $M' = \{x \in \text{at}(\phi) \mid f'(x) = \text{red}\}$. Since neither false nor none are colored red by f, there must be some vertex $x \in \text{at}(\phi)$ with $f(x) = \text{red}$ but $f'(x) \neq \text{red}$, so it holds that $M' \subset M$.

By assumption, M is a minimal model, so M' cannot be a model. Thus there is some clause $c = (l_0, l_1, l_2)$ with $M' \not\models c$. This can only be caused by f' assigning these literal vertices either green or blue, and all of these vertices have the same color due to the edges to none which is either green or blue. W.l.o.g. – because the other case is symmetric – we assume that $f'(l_0) = f'(l_1) = f'(l_2) = \text{green}$. Thus $f'(c_3) = \text{green}$ due to the clique $\{c_0, c_1, c_3\}$. We also get $f'(c_2) = \text{blue}$ due to $f'(c_5) = \text{red}$ and $f'(l_2) = \text{green}$, which implies $f'(c_4) = \text{green}$. This contradicts f' being a proper coloring due to the edge (c_3, c_4). So f must indeed be red-minimal w.r.t. W.

"If" direction: For a 3-CNF formula ϕ and an atom q, let f be a red-minimal coloring of $R(\phi, q)$ w.r.t. W such that $f(q) = \text{red}$. Let the color of none resp. false under f be blue resp. green. From f we construct an interpretation $M = \{x \in \text{at}(\phi) \mid f(x) = \text{red}\}$. Obviously, $q \in M$. It can be shown that $M \models \phi$ for if M did not satisfy some clause, we would obtain a contradiction by the same argument as in the "only if" direction of the proof.

It remains to show that M is minimal. Suppose to the contrary that there is a model $M' \subset M$. From this we construct a coloring f' appropriate to $R(\phi, q)$ analogous to the construction in the "only if" direction of the proof. It holds that $\{v \in W \mid f'(v) =$

red$\} \subset \{v \in W \mid f(v) = \text{red}\}$, which is a contradiction to f being a red-minimal coloring. We can thus conclude that M is a minimal model of ϕ. □

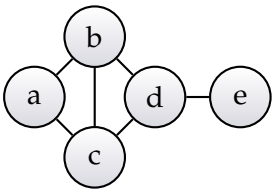
Multi-Level Characteristics for Minimum 3-Coloring

We now present a concrete example that illustrates the use of multi-level characteristics. Figure 3.3 shows a run of a dynamic programming algorithm for Minimum 3-Coloring on a graph where we want to minimize the color red on the set of all vertices. A characteristic in this algorithm has the following structure: The top-level item set encodes a 3-coloring of the subgraph induced by the current bag. Each subsidiary item set also encodes a 3-coloring of the same subgraph under the proviso that the critical vertices that are colored red by this coloring are a subset of the critical vertices that are colored red by the coloring encoded in the respective top-level item set. Just as each top-level item set in a non-leaf node extends top-level item sets from the child tables and thus implicitly represents a coloring of the subgraph induced by the bags of the subtree rooted at the current node, also each subsidiary item set can conceptually be expanded to a coloring of that subgraph. If the set of red critical vertices under this expanded subsidiary coloring is actually a *proper* subset of the set of red critical vertices under the expanded top-level coloring, a special item is contained in the subsidiary item set to indicate this. In Figure 3.3c, we denote this by a $\times$ symbol in the $\subset$ column. The rationale for this special item is that eventually only characteristics that possess no subsidiary item set marked with a $\times$ symbol in the $\subset$ column represent valid solutions, because each marked subsidiary item set can be expanded to a witness that the top-level coloring is not red-minimal w.r.t. the critical vertices.

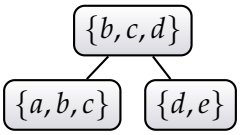
For space reasons, Figure 3.3 does not depict an empty root in the decomposition and thus also no root table. The root table would only contain a single row with a characteristic consisting of an empty top-level item set as well as an empty subsidiary item set. The extension pointers of this row would be $\{(0), (2), (3), (5), (7), (9)\}$. The non-referenced rows from the root's child table do not yield a valid solution due to the presence of marks in the $\subset$ column indicating that there is a "smaller" coloring.

For instance, row 8 of the upper table in Figure 3.3c represents a coloring that maps b and e to red, c to green, and a and d to blue. The marked subsidiary item sets however indicate witnesses that this is not a valid solution because there are two colorings that make only b red.

When considering all rows in the topmost table in Figure 3.3c, it can be observed that eight minimal colorings exist. In contrast, twelve (not necessarily minimal) colorings exist that could be obtained by ignoring the fact that some rows possess subsidiary item sets indicating that the respective row does not correspond to a *minimal* coloring. For instance, as discussed before, although row 8 of the topmost table represents a proper coloring, our algorithm will not consider it a solution because changing the color of e would yield a "smaller" coloring.



(a) Graph of a Minimum 3-Coloring instance



(b) A tree decomposition of the instance (empty root omitted for space reasons)

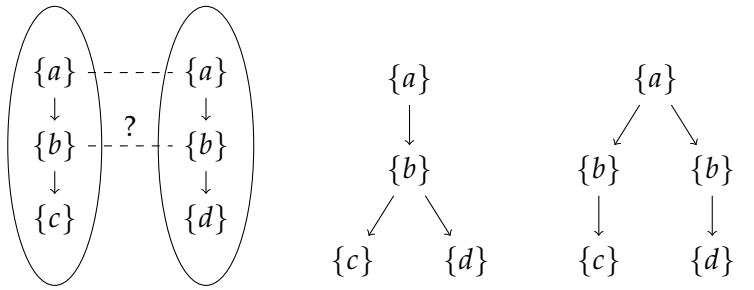
i	characteristic b	c	d	b	c	d	$\subset$	extend
0	b	g	r	b g	g b	r r		$\{(4,4),(4,5)\}$
1	b	r	g	b b g g	r r r r	g g b b	 × ×	$\{(2,3)\}$
2	b	r	g	b g	r r	g b		$\{(2,2)\}$
3	g	b	r	b g	g b	r r		$\{(5,4),(5,5)\}$
4	g	r	b	b b g g	r r r r	g g b b	 × ×	$\{(0,1)\}$
5	g	r	b	b g	r r	g b		$\{(0,0)\}$
6	r	b	g	r r r r	b b g g	g g b b	 × ×	$\{(3,3)\}$
7	r	b	g	r r	b g	g b		$\{(3,2)\}$
8	r	g	b	r r r r	b b g g	g g b b	 × ×	$\{(1,1)\}$
9	r	g	b	r r	b g	g b		$\{(1,0)\}$

i	characteristic a	b	c	a	b	c	$\subset$
0	b	g	r	b g	g b	r r	
1	b	r	g	b g	r r	g b	
2	g	b	r	b g	g b	r r	
3	g	r	b	b g	r r	g b	
4	r	b	g	r r	b g	g b	
5	r	g	b	r r	b g	g b	

i	characteristic d	e	d	e	$\subset$
0	b	g	b g	g b	
1	b	r	b b g g	g r b r	 × ×
2	g	b	b g	g b	
3	g	r	b b g g	g r b r	 × ×
4	r	b	b g r r	g b b g	 × ×
5	r	g	b g r r	g b b g	 × ×

(c) The dynamic programming tables

Figure 3.3 Dynamic programming for Minimum 3-Coloring



(a) Two paths of item sets

(b) Merging the $\{b\}$ nodes of the paths

(c) Not merging the $\{b\}$ nodes

Figure 3.4 Ambiguity when merging paths of different answer sets

From Answer Sets to Multi-level Characteristics

When multi-level characteristics are employed, compiling table rows from answer sets is more complicated than presented in Section 3.1.2, for then an answer set alone generally no longer specifies a complete characteristic. Rather, an answer set only encodes *one path* of a characteristic from the top-level item set to a leaf item set. This is because the most convenient way to generate such a tree of sets with ASP is guessing a set for each level from the root to a leaf.

Reconstructing the intended trees from the paths described by answer sets is, however, a relatively intricate task. Consider, for instance, building a characteristic from the two paths $(\{a\},\{b\},\{c\})$ and $(\{a\},\{b\},\{d\})$, where a, b, c and d are arbitrary items. The ambiguity of this situation is illustrated in Figure 3.4. Without further information, it is not clear whether the item sets $\{c\}$ and $\{d\}$ are intended to be children of a common node with the item set $\{b\}$, or whether there should be two distinct nodes having both the item set $\{b\}$. The latter case can be necessary when two item sets of a characteristic in a child node become equal because vertices have been removed from the bag. Merging these parts of the path as depicted in Figure 3.4b would then lead to incorrect results. For example, when we wish to determine whether *there is* a top-level item set such that *for all* its children *there is* in turn a child item set containing c, the tree in Figure 3.4b would represent a satisfying characteristic whereas the tree in Figure 3.4c would not.

Which course of action should be followed thus depends on which parts of the paths in the answer sets stem from which parts of the paths in the characteristics from the child nodes. An answer set must therefore state, for each node of the path, extension pointers referring to the predecessor of the respective node if such ambiguities should be avoided, just as extension pointers are used to specify predecessors of table rows for enumeration problems. The difference is that in the case of multi-level characteristics where such ambiguities are to be avoided, extension pointers not only

declare predecessors of the top-level item set but declare predecessors for any item set.[3]

Just as it is required that there is at most one table row with a given characteristic, a *Decompose, Guess & Check* framework must make sure that for each node in a characteristic it holds that there are no two equal subtrees rooted at a child of that node. Equal subtrees must therefore be merged, similar to merging rows with equal characteristics, in order to ensure that the space consumption can be bounded by a function only depending on the treewidth.

To summarize, when working with multi-level characteristics, the user can not only specify extension pointers for table rows but also for subsidiary item sets. If such pointers are present, a *Decompose, Guess & Check* tool uses this information to construct the characteristics from the answer sets accordingly. In either case, a framework for *Decompose, Guess & Check* must ensure that there is at most one table row for each characteristic, and that within a characteristic there are no duplicate subtrees at the same level.

3.2 Algorithm Design Methodology

We now compile a recommendation for the – often by no means trivial – procedure of implementing a tree-decomposition-based dynamic programming algorithm for a problem. [Bodlaender, 1997] lists a few steps that should be followed when designing such an algorithm, which also apply to our approach.[4]

1. First of all, the user should come to an understanding how *solutions* should be represented.

2. When this is done, the user should think about the structure of a *partial solution*, i.e., a part of a solution when considering only the subgraph induced by the bags of a tree decomposition node's descendants.

   This is often much more difficult than defining the structure of global solutions because the decomposition has to be taken into account. Usually, one can gain an understanding of partial solutions by imagining only those parts of a complete solution that apply to a subgraph, and then figuring out what information is required to extend such a partial solution. In some cases, it is enough to use the same basic structure of global solutions also for partial solutions. Due to the

[3] This does not mean that algorithms using multi-level characteristics must always support enumeration. The information to keep equal item sets stemming from different paths apart may well be arbitrary strings that are merely used as distinguishable identifiers and need not refer to an actual part of a child characteristic.

[4] It should be noted that [Bodlaender, 1997] uses nice tree decompositions. Our approach, on the other hand, uses arbitrary tree decompositions.

decomposition, however, it is usually necessary to store extra information that is used for combining partial solutions.

3. Having agreed on a way to represent partial solutions, it is now required to specify under which circumstances a solution is an *extension of a partial solution*.

   This is necessary for the step of extending or combining different partial solutions from child tables because it gives us a criterion to decide if a row cannot lead to a solution and must therefore not be stored in the table.

4. The user must define the structure of the *table rows*, i.e., define the representation of a table row's characteristic. Note that it is infeasible to store a whole partial solution in a table row. Rather, we must restrict ourselves to those parts of the information that are relevant for the current bag, i.e., the part that is used for combining the partial solutions of different nodes. The structure of the table rows needs to be chosen such that a row contains all the information necessary for deciding whether the partial solution corresponding to it can be extended.

5. It must be specified how to compute a table, given the tables of child nodes. For an algorithm that is tractable for bounded treewidth, the user must also make sure that these tables can be computed efficiently in a bottom-up way.

6. Finally, the user must show that the rows in the root table suffice to decide the problem efficiently and to efficiently construct the desired solutions depending on the problem type, as discussed in Section 3.1.2.

If counting or enumerating solutions is to be performed, it is further the user's responsibility to make sure that the partial solutions associated with each table row are mutually disjoint. Otherwise, duplicates will be generated. Algorithms for a problem where this issue occurs are given in [Pichler et al., 2010].

3.3 Applicability

The outline of the *Decompose, Guess & Check* approach that was given above is the result of investigating several problems and how they could be solved via dynamic programming on tree decompositions. We present a few of them in Section 4.2. Of primary interest, of course, are problems that are fixed-parameter tractable for bounded treewidth, as this is the main reason why dynamic programming on tree decompositions is commonly used in the first place. After all, this technique has its roots in complexity-theoretic deliberations.

For this reason, we will present an algorithm that solves the MSO formula evaluation problem efficiently for bounded treewidth in Section 3.3.1 and thus show that our approach is applicable to any MSO-definable problem. In Section 3.3.2 we will discuss further results regarding the applicability of *Decompose, Guess & Check*.

3.3.1 Evaluation of MSO Formulas

We now present a dynamic programming algorithm that solves the MSO formula evaluation problem in linear time for graphs of bounded treewidth. An actual implementation can be found in Section 4.2.9. This is primarily interesting because it implies that *Decompose, Guess & Check* can be effectively employed for *any* MSO-definable problem. In fact, the algorithm can be used to obtain an alternative proof of Courcelle's theorem. Related approaches can be found in [Grohe, 1999, Kneis et al., 2011].

Preliminaries

For the sake of simplicity, we only consider the case of MSO where the universe of discourse consists of the graph vertices, and the vocabulary is $\{\text{edge}\}$, i.e., the only available predicate symbol is the binary predicate symbol "edge" whose semantics is that $(G, \alpha) \models \text{edge}(x, y)$, for a graph G, an assignment α and individual variables x and y, if and only if $(\alpha(x), \alpha(y))$ is an edge in G. Courcelle's theorem of course also holds for this version of MSO, i.e., anything expressible in this formalism is solvable in linear time for graphs of bounded treewidth, although some problems that also enjoy this property (like Hamiltonian Cycle; cf. Section 4.2.8) cannot be expressed using only this restricted vocabulary. The algorithm presented here could in principle, however, be extended to vocabularies that lead to higher expressiveness.

We further restrict ourselves to MSO formulas in *prenex conjunctive normal form* with $\exists$ as the outermost quantifier. Such a formula has the form $\exists\Gamma_0 \forall\Gamma_1 \dots Q\Gamma_{\nu-1}\psi$ where Q is $\exists$ for even ν and $\forall$ for odd ν, each Γ_i $(0 \leq i < \nu)$ is a set of variables and ψ is a quantifier-free CNF formula in MSO over the vocabulary $\{\text{edge}\}$. Further, let $\Gamma = \bigcup_{0 \leq i < \nu} \Gamma_i$, and for any $\Delta \subseteq \Gamma$, let $\mathcal{IV}(\Delta)$ and $\mathcal{SV}(\Delta)$ denote the set of individual and set variables in Δ, respectively.

Example 3.2. Consider the prenex CNF formula

$$\phi = \exists S\ \forall x \forall y\ \Big(\overbrace{\big(\underbrace{\neg\,\text{edge}(x,y) \vee S(x)}_{c_0} \big) \wedge \big(\underbrace{\neg\,\text{edge}(x,y) \vee S(y)}_{c_1} \big)}^{\psi} \Big)$$

which expresses that there is a set containing all vertices that are endpoints of an edge. We have $\Gamma = \Gamma_0 \cup \Gamma_1$ with $\Gamma_0 = \{S\}$ and $\Gamma_1 = \{x, y\}$. It holds that $\mathcal{IV}(\Gamma) = \{x, y\}$ and $\mathcal{SV}(\Gamma) = \{S\}$. Of course, any graph satisfies ϕ, but it should suffice to illustrate the concepts of this section and we will return to this example. △

For a fixed MSO formula ϕ over the vocabulary $\{\text{edge}\}$ in prenex CNF with $\exists$ as the outermost quantifier, we define Mso Formula Evaluation to be the following problem.

Input:	A graph G
Question:	Does $G \models \phi$ hold?

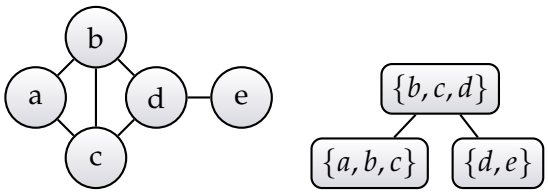


Figure 3.5 A graph and one of its tree decompositions

According to Courcelle's theorem, this problem is fixed-parameter tractable w.r.t. treewidth. To construct a dynamic programming algorithm on a tree decomposition, we proceed according to the methodology presented in Section 3.2.

Before presenting the algorithm, we define some terminology. In the following, let $(\mathcal{T}, \chi)$ be a tree decomposition of the input graph $G = (V, E)$. For any node $n \in \mathcal{T}$, let $\mathcal{T}_n$ denote the subtree of $\mathcal{T}$ rooted at n. We further define V_n as the set of vertices occurring in some bag of $\mathcal{T}_n$, i.e.,

$$V_n = \bigcup_{m \in \mathcal{T}_n} \chi(m).$$

Definition 3.3. When we speak of an *assignment* to a set of variables $\Delta \subseteq \Gamma$, we mean a mapping α such that for any $x \in \mathcal{IV}(\Delta)$ it holds that $\alpha(x) \in V$ and for any $X \in \mathcal{SV}(\Delta)$ it holds that $\alpha(X) \subseteq V$. □

We can combine assignments to the variables belonging to the individual quantifier levels to an assignment to all variables as follows.

Definition 3.4. Given assignments $\alpha_0, \ldots, \alpha_{\nu-1}$ to $\Gamma_0, \ldots, \Gamma_{\nu-1}$, respectively, we define $\alpha_0 \circ \cdots \circ \alpha_{\nu-1}$ as the assignment to Γ that, for each $0 \leq i < \nu$, maps any $x \in \Gamma_i$ to $\alpha_i(x)$. □

Now we can state the structure of a solution of MSO FORMULA EVALUATION.

Definition 3.5. By a *solution* we understand a tree A having the following properties:

- Each leaf is at depth $\nu - 1$.
- Each node at depth d is an assignment to Γ_d.
- For each node at depth $d < \nu - 1$, the set of children is exactly the set of all possible assignments to Γ_{d+1}.
- Let α_0 be the assignment of the root of A. For each assignment α_1 at depth 1, there is a child α_2 at depth 2 such that for each child α_3 at depth 3, and so on, it holds that $(G, \alpha_0 \circ \cdots \circ \alpha_{\nu-1}) \models \psi$. □

Example 3.6. The unique solution of the MSO FORMULA EVALUATION problem for the formula from Example 3.2 and the graph in Figure 3.5 is depicted in Figure 3.6. Observe that for each root-to-leaf path (α_0, α_1) it holds that $(G, \alpha_0 \circ \alpha_1) \models \psi$. △

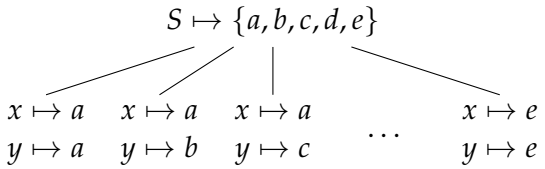


Figure 3.6 A solution of MSO FORMULA EVALUATION for the formula from Example 3.2 and the graph from Figure 3.5

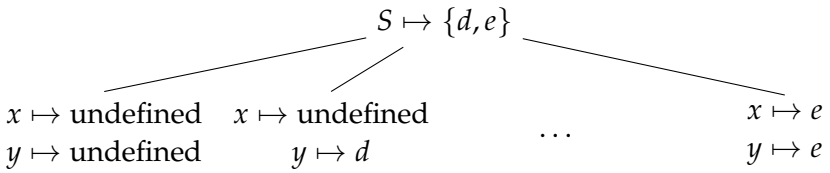


Figure 3.7 A partial solution for the formula from Example 3.2 and the graph from Figure 3.5

For partial solutions we have to take into consideration that not all variables need to be assigned yet.

Definition 3.7. When we speak of a *partial assignment* to a set of variables $\Delta \subseteq \Gamma$ relative to a node $n \in \mathcal{T}$, we mean a mapping α such that for any $x \in \mathcal{IV}(\Delta)$ it holds that $\alpha(x) \in V_n \cup \{\text{undefined}\}$ and for any $X \in \mathcal{SV}(\Delta)$ it holds that $\alpha(X) \subseteq V_n$. □

A partial assignment thus may leave some individual variables undefined so that they can later be set to vertices yet to be encountered in the tree decomposition.

Definition 3.8. By a *partial solution* at a node $n \in \mathcal{T}$ we understand a tree A having the following properties:

- Each leaf is at depth $\nu - 1$.
- Each node at depth d is a partial assignment to Γ_d relative to n.
- For each node at depth $d < \nu - 1$, the set of children is exactly the set of all possible partial assignments to Γ_{d+1} relative to n. □

Example 3.9. For the formula from Example 3.2 and the graph and tree decomposition in Figure 3.5, a partial solution at the leaf node with the bag $\{d, e\}$ is depicted in Figure 3.7. This partial solution will eventually be extended to the solution depicted in Figure 3.6. The other partial solutions (that will not give rise to a solution) at the same node are identical except mapping S to any proper subset of $\{d, e\}$. △

We say that a solution A *extends* a partial solution A' if for each $x \in \mathcal{IV}(\Gamma_0)$ and each $v \in V$ it holds that $\alpha'_0(x) = v \Rightarrow \alpha_0(x) = v$, and for each $X \in \mathcal{SV}(\Gamma_0)$ it holds that $\alpha'_0(X) \subseteq \alpha_0(X)$, where α_0 is the root of A and α'_0 is the root of A'.

In order to define the notion of a characteristic, we need to consider assignments that not only allow variables to be still unassigned (as partial assignments do) but also account for variables that have been set to vertices that are no longer in the current bag.

Definition 3.10. When we speak of a *local assignment* to a set of variables $\Delta \subseteq \Gamma$ relative to a node $n \in \mathcal{T}$, we mean a mapping α such that for any $x \in \mathcal{IV}(\Delta)$ it holds that $\alpha(x) \in \chi(n) \cup \{\text{undefined}, \text{taken}\}$ and for any $X \in \mathcal{SV}(\Delta)$ it holds that $\alpha(X) \subseteq \chi(n)$. □

Given a local assignment α relative to a node $n \in \mathcal{T}$, the intuition behind setting an individual variable x to "undefined" is that x has neither been assigned a vertex in V_n by α nor by some local assignment relative to a descendant of n that is considered to be an ancestor of α; the intuition behind "taken" is that x has been assigned a vertex in $V_n \setminus \chi(n)$ by a local assignment relative to a descendant of n that is considered to be an ancestor of α.

Having fixed these notions, we can now finally state the definition of a characteristic.[5]

Definition 3.11. We define a table row's *characteristic* at a node $n \in \mathcal{T}$ as a tree A having the following properties:

- Each leaf is at depth $\nu - 1$.
- Each node at depth d encodes a local assignment to Γ_d relative to n.
- Each leaf additionally encodes a set of clauses.
- For each node at depth $d < \nu - 1$ and each possible local assignment to Γ_{d+1} relative to n, there is at least one child of that node encoding this local assignment. □

By keeping track of extension pointers for each node in a characteristic, each local assignment in a characteristic can be extended to a set of partial assignments. Each path from the root item set to a leaf item set thus implicitly represents a set of partial assignments to Γ relative to the current node. This set of partial assignments can be obtained from a root-to-leaf path in a characteristic by unfolding the path to a tree in a bottom-up way by following the extension pointers.

[5]Note that the notions of a solution, a partial solution and an extension of a partial solution are, strictly speaking, not necessary for conceiving an algorithm. We have nevertheless introduced them before defining the structure of a characteristic, which *is* crucial for an actual algorithm, in order to comply with the algorithm design methodology found in Section 3.2. This methodology lists these initial definitions with good reason, however, because they help in designing such an algorithm and often act as guidelines for the continuing abstraction from the frequently obvious nature of a solution to the usually much more elusive structure of a characteristic. Another important justification is that if we want to prove the correctness of an algorithm, we normally cannot do without formalizing these fundamental notions.

Like a partial solution, a characteristic cannot "look into the future", which explains the need for the value "undefined". However, unlike a partial solution, which has access to information about already removed vertices, a characteristic cannot "look into the past" either. This is the reason that we need to allow "taken" as a value for individual variables that were set to a vertex that has meanwhile been removed from the current bag.

We also need to store in a characteristic which clauses have been satisfied so far – after a clause becomes satisfied when guessing an interpretation, the vertices that are the "reason" for the clause being satisfied might be removed from the bag and thus also vanish from the characteristic, making it necessary to conduct bookkeeping of satisfied clauses. The intuition behind the clauses encoded in a leaf is that all partial assignments to Γ represented by the path from the root of the characteristic to that leaf satisfy exactly those clauses. This way, the tree structure of a characteristic allows us to check if, for instance when $\nu = 2$, for the assignment to Γ_0 represented by the root of that characteristic, all assignments to Γ_1 satisfy all clauses. Checking if there is an assignment to Γ_0 such that for all assignments to Γ_1 all clauses are satisfied thus amounts to checking if there is a characteristic in the root table such that for each of the characteristic's root-to-leaf paths that represents a (complete) assignment all clauses are contained in the leaf.

Example 3.12. For the formula from Example 3.2 and the graph and tree decomposition in Figure 3.5, a characteristic at the root node is depicted in Figure 3.8 (in a table layout for the sake of clear arrangement).

Note that for instance there are two distinct root-to-leaf paths encoding the local assignment that maps $S \mapsto \{b\}$, $x \mapsto$ taken and $y \mapsto b$. The reason is that the corresponding satisfied clauses differ. In particular, the presence of the path that does not satisfy c_0 witnesses (among others) that this characteristic does not represent a valid solution. Such a situation – multiple root-to-leaf paths encoding the same local assignment but different satisfied clauses – can occur when two paths coincide on the current vertices but not on the satisfied clauses. If, on the other hand, two paths coincide on both the current vertices and the satisfied clauses, they are merged (as described in Section 3.1.2). This way, an explosion of memory is avoided and the size of a characteristic is always bounded by a function only depending on the treewidth.

We can explain the mentioned distinct paths encoding the same local assignment but different clauses by the fact that in one case x was previously set to the no longer present vertex a which led to c_0 not being satisfied, and in the other case x was previously set to the no longer present vertex e which led to both clauses being satisfied. Since both a and e are not in the current bag, x takes the value "taken" in either of the resulting paths. △

S	x	y	c_0	c_1
$\{b\}$	undefined	undefined		
	undefined	taken	✓	✓
	undefined	b		✓
	undefined	c		
	undefined	d		
	taken	b	✓	✓
	taken	b		✓
	taken	c	✓	✓
	taken	c		
	taken	d	✓	✓
	taken	undefined	✓	✓
	taken	taken	✓	✓
	b	undefined	✓	
	b	taken	✓	✓
	b	b	✓	✓
	b	c	✓	
	b	d	✓	
	c	undefined		
	c	taken	✓	✓
	c	b	✓	✓
	c	c	✓	✓
	c	d		
	d	undefined		
	d	taken	✓	✓
	d	taken		✓
	d	b	✓	✓
	d	c	✓	✓
	d	d	✓	✓

Figure 3.8 A characteristic at the root node of the tree decomposition in Figure 3.5 and the formula from Example 3.2

The Dynamic Programming Algorithm

We now describe how the tables are computed, i.e., the actual dynamic programming algorithm. A concrete implementation in terms of ASP code is presented in Section 4.2.9.

Leaf nodes. For a leaf node $n \in \mathcal{T}$, we insert a row into the table for each possible partial solution relative to n. This can be done by means of *Guess & Check* in an ASP program by guessing a root-to-leaf path and processing the answer sets according to the principles in Section 3.1.2. Additionally, for each root-to-leaf path of partial assignments $(\alpha_0, \ldots, \alpha_{\nu-1})$, we add the clauses that are satisfied by $\alpha_0 \circ \cdots \circ \alpha_{\nu-1}$ into the leaf item set.

Non-leaf nodes. For a non-leaf node $n \in \mathcal{T}$, we guess

- for each child node a root-to-leaf path appearing in a characteristic of that node's table,
- a local assignment to Γ relative to n, and
- a subset of the clauses in ϕ.

The checking part now makes sure that the following conditions hold.

- Let α be the guessed local assignment to Γ relative to n, n' be an arbitrary child of n, and α' be the local assignment to Γ relative to n' encoded in the guessed root-to-leaf path for n'. Intuitively, we require that α actually extends α'. We now make this formal.

  Let $x \in \mathcal{IV}(\Gamma)$ resp. $X \in \mathcal{SV}(\Gamma)$ be arbitrary individual resp. set variables. The following conditions must hold.

  - If $\alpha'(x) = \text{undefined}$, then $\alpha(x) \in \{\text{undefined}\} \cup (\chi(n) \setminus \chi(n'))$.
  - If $\alpha'(x) = \text{taken}$, then $\alpha(x) = \text{taken}$.
  - If $\alpha'(x) = v$ for some $v \in \chi(n') \setminus \chi(n)$, then $\alpha(x) = \text{taken}$.
  - If $\alpha'(x) = v$ for some $v \in \chi(n)$, then $\alpha(x) = v$.
  - For any $v \in \chi(n') \cap \chi(n)$, $v \in \alpha'(X)$ if and only if $v \in \alpha(X)$.

- For each pair of child nodes a and b, let the local assignments to Γ encoded by the guessed root-to-leaf paths for a and b be denoted by α and β, respectively. Intuitively, we require that α and β be compatible in a certain way. That is to say, for any individual variable, if the partial assignment corresponding to α assigns this variable to some vertex, then the partial assignment corresponding to β must not assign it to a different vertex; and for each set variable it must not be the case that a common vertex of a and b is contained in the interpretation of that variable

according to α while not so according to β. We now state these requirements in a more formal way.

Let $x \in \mathcal{IV}(\Gamma)$ resp. $X \in \mathcal{SV}(\Gamma)$ be arbitrary individual resp. set variables. The following conditions must hold.

- If $\alpha(x) = v$ for some $v \in \chi(a) \cap \chi(b)$, then it holds that $\beta(x) = v$.
- If $\alpha(x) = v$ for some $v \in \chi(a) \setminus \chi(b)$, then it holds that $\beta(x) =$ undefined.
- If $\alpha(x) =$ taken, then it holds that $\beta(x) =$ undefined.
- If $v \in \alpha(X)$ for some $v \in \chi(a) \cap \chi(b)$, then it holds that $v \in \beta(X)$.

- Each guessed clause is contained in the leaf of a guessed root-to-leaf path of a child node or is satisfied by the guessed local assignment to Γ relative to n.

If the current node is the root (which has an empty bag and a single child), the following additional checks are performed for the guessed root-to-leaf path from the child node.

- Let α be the local assignment to Γ encoded by the guessed root-to-leaf path from the child node. For each $x \in \mathcal{IV}(\Gamma)$ it must hold that $\alpha(x) \neq$ undefined.
- We check that the root of the guessed root-to-leaf path from the child node is labeled with "good" after performing the following labeling in each of the child table's characteristics. We label each node m at depth d, where m encodes a local assignment α_d to Γ_d relative to the child node, as follows:
  - We label m with "invalid" if there is an $x \in \mathcal{IV}(\Gamma_d)$ such that $\alpha_d(x) =$ undefined.
  - We label m with "invalid" if its parent is "invalid".
  - If m is a leaf that is not "invalid":
    If all clauses are contained in m, we label it with "good" and otherwise with "bad".
  - If m is no leaf, not "invalid", and d is even:
    If m has a child labeled with "bad", we label m with "bad" and otherwise with "good".
  - If m is no leaf, not "invalid", and d is odd:
    If m has a child labeled with "good", we label m with "good" and otherwise with "bad".

This way, we make sure that table rows that do not correspond to solutions are eliminated – either because some variables are still unassigned or because it does not hold that for all assignments to the variables on level 1 there is an assignment to the variables on level 2, and so on, such that all clauses are satisfied.

Finally, to track which part of a characteristic stems from which parts of child characteristics, for each node at depth d of the resulting path (if it survives the checking part) we declare a tuple of extension pointers where each pointer references the node at depth d of the respective guessed root-to-leaf-path from the child characteristic.

This concludes the description of the dynamic programming algorithm for Mso Formula Evaluation.

Correctness and Complexity of the Algorithm

We now argue that the presented algorithm is correct and efficient for graphs of bounded treewidth. It can be shown that an instance is positive if and only if the root table contains a row. We only give a rough idea for a proof. A full elaboration is left for future work.

We can extend each table row associated to any node $n \in \mathcal{T}$ to a set of partial solutions by recursively following the extension pointers. Each root-to-leaf path of a characteristic thus corresponds to a set of paths in these partial solutions. Every such path represents a partial assignment to Γ relative to n. Consider an arbitrary root-to-leaf path of a characteristic. Let the clauses encoded in the leaf be denoted by C and the set of paths in the corresponding partial solutions be denoted by P. Each path in P encodes a partial assignment to Γ relative to n. The clauses in C are exactly the clauses that each such partial assignment satisfies. This is because we started at the tree decomposition's leaves with all possible partial assignments together with the respective satisfied clauses, and each time we extended a local assignment from a child, we added the clauses that got satisfied due to this extension to the respective item set, thus collecting all satisfied clauses during the progress of the computation. The final checks in the root table make sure that only those partial solutions that are actually solutions survive.

Using the extension pointers, we can finally materialize all solutions, so this algorithm also allows for counting and enumeration. The decision problem can indeed be solved in linear time for graphs of bounded treewidth. To prove this, let the treewidth be some constant w. According to Bodlaender's theorem (cf. Section 2.4), because the treewidth is fixed, we can construct an optimal tree decomposition in linear time. Therefore, this decomposition also has linear size. We assume that each node has constantly many children. This assumption can be justified because any tree decomposition can be normalized in linear time (cf. Section 3.1.2).[6] It remains to show that each table can be computed in constant time. For this, we first show that each table has constant size. A local assignment relative to a given node can map an individual variable to at most $w + 3$ values because the bag size is at most $w + 1$ and there are the

[6]The linear time result holds also for non-normalized tree decompositions because there can only be constantly many nodes that have linearly many children – otherwise the decomposition would not have linear size. Assuming a constant number of children for each node, however, seems to make the proof easier.

two special values "undefined" and "taken". A set variable can be mapped to at most 2^{w+1} values. Both of these figures are constant and there are constantly many variables, thus there are only constantly many possible local assignments for each node in $\mathcal{T}$. A root-to-leaf path in a characteristic encodes, in addition to a local assignment, a set of clauses of which there are also constantly many. Therefore, there are only constantly many possible root-to-leaf paths in a characteristic. The number of characteristics in a table cannot be greater than the number of possible root-to-leaf paths and is therefore also constant. The extension pointers in the table also have constant size due to the bound on the number of children and the child table sizes. From this it follows that each table has constant size. Thus, it can easily be seen that from the constant bound on the size of the tables it follows that the checking part is feasible in constant time.

3.3.2 Further Applicability Results

The preceding section shows that our proposed approach is able to solve any MSO-definable problem efficiently for graphs of bounded treewidth. *Decompose, Guess & Check* is, however, not restricted to MSO-definable problems. For instance, in Section 4.2.2, we show that we can solve problems that are provably not fixed-parameter tractable and thus also not expressible in MSO. Therefore, decomposition can also sometimes be beneficial for problems that are not fixed-parameter tractable w.r.t. treewidth, or maybe even intractable for instances of bounded treewidth. We suspect that even if it results in an exponential-time algorithm, for some practical problems, at least some instances that have previously been out of reach could be solved.

Although dynamic programming on tree decompositions is commonly used for problems in NP or, more generally, in the polynomial hierarchy, it is applicable to even harder problems. From the existence of the algorithm presented in the preceding section it follows that, since MSO Formula Evaluation is PSPACE-complete in general, by using *Decompose, Guess & Check* it is possible to solve problems efficiently for bounded treewidth that are PSPACE-complete in general. In Section 4.2.10 we will see a fixed-parameter tractable algorithm for the PSPACE-complete problem Qsat.

Chapter 4

The D-FLAT System

The D-FLAT system is our implementation of a *Decompose, Guess & Check* framework that fulfills the demands discussed in Section 3.1. The acronym abbreviates the full name *Dynamic Programming Framework with Local Execution of ASP on Tree Decompositions*. Its purpose is to take care of everything that surrounds dynamic programming algorithms on tree decompositions and to leave the user only with the responsibility of providing the problem-specific parts in ASP.

D-FLAT can be considered as a tool for rapid prototyping of dynamic programming algorithms but also for educational purposes. ASP users are provided with an easy-to-use interface to decompose problem instances – an issue which might allow large instances of practical importance to be solved, which so far could not be handled by ASP systems.

The system can be obtained as free software from `http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/`. It is written in C++ and can be compiled for many platforms.

A first prototype was already introduced in [Bliem et al., 2012], but the present work significantly extends that version in many respects.

In this chapter, we will first give an overview of the system by describing its organization, the interplay of its components, and the interface to the user's programs in Section 4.1. We then present some concrete examples for solving problems via *Decompose, Guess & Check* by means of ASP programs for the D-FLAT system in Section 4.2. Finally, we briefly mention performance aspects in Section 4.3.

4.1 System Overview

We now describe the different parts of D-FLAT and their roles in solving problems in a decomposed way. Figure 4.1 depicts the control flow during the execution of an algorithm with D-FLAT and illustrates the interplay of the system's components.

The system consists of the following basic elements:

- The D-FLAT core; it coordinates the data flow between all other components and takes care of parsing the input, storing and processing the tables, as well as materializing solutions.

  The D-FLAT core is tightly intertwined with a software called *SHARP*[1] [Morak, 2011] which is a framework for working on tree decompositions using C++ as a language for the algorithms.

- An ASP solving system; currently we use *Gringo* for grounding and *clasp* for solving. These programs are part of the Potsdam Answer Set Solving Collection [Gebser et al., 2011b] and are currently the generally most efficient ASP solving tools available [Denecker et al., 2009].

- A tree decomposition library; we currently employ *htdecomp* [Dermaku et al., 2008] for this purpose.

Initially, D-FLAT parses the problem instance – a set of facts describing a graph –, stores this graph and constructs a tree decomposition. Now the bottom-up processing is initiated. Until all tables have been computed, D-FLAT visits the next node whose child tables are already completed. It takes these child tables and flattens the rows contained in them such that a string representing a set of facts is obtained. It is passed to the integrated ASP solver together with the user's ASP program and a description of the bags of the current node and its children. The answer sets which result are now scanned for predicates that instruct D-FLAT to store certain values in a table row. This way, D-FLAT populates the current node's table and continues with the next node. When all tables have thus been computed, the solutions are materialized depending on the problem type. For instance, for a decision problem, "yes" or "no" is returned depending on whether the root table has a row or not; for a counting problem, the number of solutions is read off from the table rows; and for an enumeration problem, complete solutions are obtained by following extension pointers stored in the table rows for this purpose. Note that for optimization problems, D-FLAT automatically takes care of only counting or enumerating optimal solutions.

In Section 4.1.1 we present the steps during an invocation of the tool in more detail. We then give a summary of command-line options in Section 4.1.2 and of the reserved predicates that are used as an interface to the user's programs in Section 4.1.3.

4.1.1 Description of Individual Steps

The dynamic programming algorithm is provided by the user by means of an ASP program. We now describe the different steps during the execution of such an algorithm with D-FLAT in more detail.

[1] `http://www.dbai.tuwien.ac.at/research/project/sharp/`

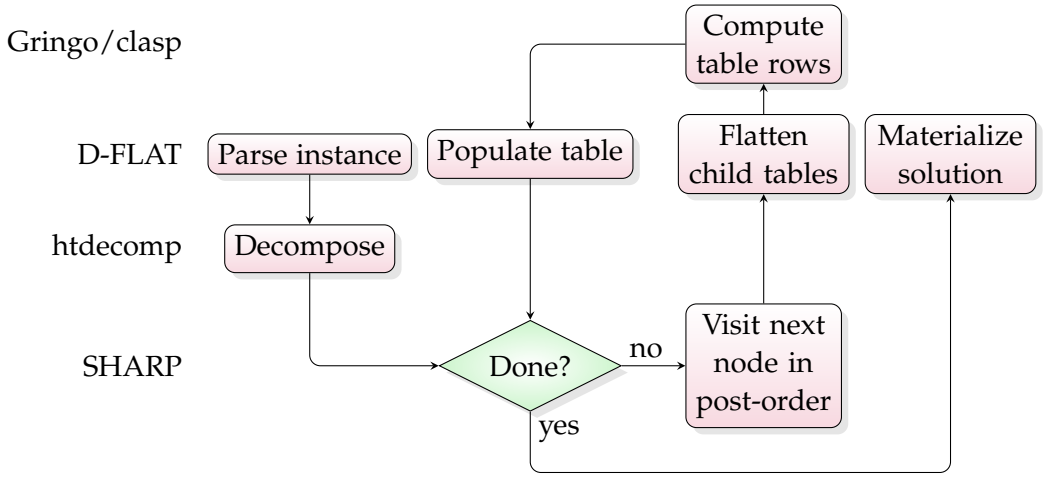


Figure 4.1 A flowchart illustrating the (simplified) interplay of D-FLAT's components

Parsing the Input

As we are using tree decompositions as a way to decompose problem instances, we require the input to be a graph.[2] This input graph must of course be made available to the user's ASP program because it contains the problem-specific information and is needed to compute tables. Since we use ASP as the language for the user's program, using ASP also for specifying the input graph is a natural choice. This way, we can simply pass the input directly to the ASP solver together with the user's program, a description of the bags and the child tables. Hence, the input is required to be given as a set of facts, exactly as one would provide input for a monolithic ASP program for a problem.

For instance, the following input describes the graph from Figure 3.1a:

```
vertex(a;b;c;d;e;f;g).
edge(a,b). edge(b,c). edge(c,d). edge(c,g).
edge(d,e). edge(d,f). edge(e,f).
```

In order to recognize how this set of facts represents a graph, D-FLAT expects the user to specify as command-line arguments which predicates in the input denote edges – in this case, `edge/2`.

The constants which appear as arguments of the edge predicates in the input are considered to be exactly the vertices of the graph – thus, the vertices do not need to be declared explicitly (although no one is preventing the user from doing so if convenient, as we have done in the listing using the `vertex/1` predicate).

[2]To be precise, D-FLAT can also handle hypergraphs as described in Section 2.4.1, but here we only speak of graphs for the sake of simplicity.

Decomposing the Graph

The graph that was obtained in the previous step is now decomposed. D-FLAT leaves the details of this to an external library that is concerned with constructing a tree decomposition heuristically that has near-optimal width. At the moment, the only available possibility for controlling this is specifying on the command-line which normalization should be performed (cf. Section 3.1.2). The correct choice, of course, depends on what the particular dynamic programming algorithm expects as a tree decomposition.

Note that D-FLAT constructs a tree decomposition such that bag of the root is empty, as depicted in Figure 3.1b, for the reasons discussed in Section 3.1.[3]

Traversing the Tree Decomposition

When the dynamic programming phase begins, a bottom-up traversal of the tree decomposition is performed to compute the tables. This way, upon reaching a tree decomposition node, the child tables are already computed. In each node, D-FLAT invokes the ASP solver to compute the new table, which is described next.

Inside a Tree Decomposition Node

Figure 4.2 depicts the data flow when D-FLAT visits a node. D-FLAT first flattens the table of each child node, i.e., it builds a set of facts which describes all the rows in that table. Then, to compute the new table, D-FLAT invokes the integrated ASP solver with the following input:

- The dynamic programming algorithm provided by the user as an ASP program
- The entire problem instance (given to D-FLAT as a set of facts)
- A description of the current bag as a set of facts stating which vertices are present in that bag
- For each child node, the set of facts describing its table and a description of its bag contents

The answer sets resulting from this constitute the new table. The user's dynamic programming algorithm must instruct D-FLAT which values it should store in a table

[3]It should be noted that when D-FLAT is asked to construct a semi-normalized tree decomposition, it also provides for empty leaf nodes because the algorithms presented in [Bliem et al., 2012], in contrast to the ones in this work, consist of two separate encodings – one for exchange nodes and one for join nodes. Note the lack of a "leaf program" which is because we construct semi-normalized tree decompositions in such a way that all leaves have empty bags and a table that always consists of an empty row. This construction does not restrict the user in any way and, in all considered cases, leads to clearer ASP code than abusing the exchange program for leaves. We refer to [Bliem et al., 2012] for details.

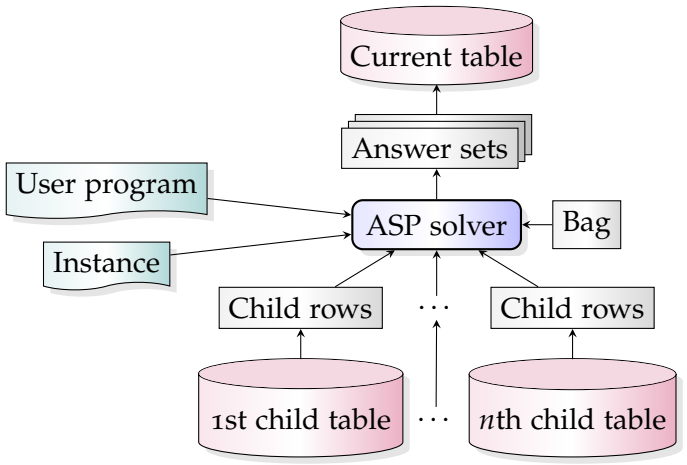


Figure 4.2 Data flow while processing a node with n children

row by means of special predicates that are used for communication between D-FLAT and the user's program (cf. Section 4.1.3). D-FLAT scans the answer sets for these predicates and inserts rows into the table accordingly, as discussed in Section 3.1.

For the example in Figure 3.1c, consider as the current node the one corresponding to the bag $\{a, b\}$. D-FLAT describes the current bag as just the two facts:[4]

```
current(a;b).
```

D-FLAT then declares the children and their bags and tables as follows:

```
childNode(c0).
childBag(c0,b;c).
childRow(c0r0;c0r1;c0r2,c0).
childItem(c0r0,b).    childCost(c0r0,4).
childItem(c0r1,c).    childCost(c0r1,3).
childItem(c0r2,b;c).  childCost(c0r2,4).
```

For convenience, D-FLAT also passes facts using the predicates `introduced/1` and `removed/1` to the user's program. They are, strictly speaking, redundant; but using them in algorithms is very common. Their semantics is in accordance with the following rules:

```
-introduced(X) ← childBag(_,X).
introduced(X) ← current(X), not -introduced(X).
removed(X) ← childBag(_,X), not current(X).
```

[4]To emphasize that these are input predicates provided by D-FLAT, we print them in color (see Section 4.1.3).

Materializing the Solutions

When all nodes have been visited, materializing the solutions proceeds as discussed in Section 3.1. The specific solution enumeration strategy of our implementation still needs some clarification, however.

In general, there can be exponentially many solutions. When materializing solutions for enumeration, it is therefore infeasible to construct all of them simultaneously and subsequently iterate over this set to print every solution, because this way the huge set of all complete solutions must be stored in memory. We rather require a "lazy materialization" technique that only materializes one solution at a time, prints it and then proceeds to the next one. To this end, D-FLAT implements an iterator interface for enumerating solutions that avoids the explosion of memory. This enumeration is even possible with just linear delay, provided the treewidth is bounded and the user designs the dynamic programming algorithm such that the size of each table can be considered as bounded by a constant (which is the case if it is bounded by a function only depending on the treewidth).

4.1.2 Command-Line Interface

The file name of the user's ASP program must be specified as a command-line argument and the instance will be read from the standard input.

On semi-normalized tree decompositions (see Section 3.1), D-FLAT distinguishes *two* programs – one for *exchange nodes* and one for *join nodes* – that have to be specified separately. In [Bliem et al., 2012], we introduced encodings for this setting. When a program for exchange nodes but not for join nodes is specified, D-FLAT executes a default join implementation which joins all pairs of candidate rows that have the same characteristic. We refer to [Bliem et al., 2012] for a discussion of the default join implementation.

Table 4.1 summarizes the command-line options that control D-FLAT's behavior. Executing the D-FLAT binary `dflat` is illustrated by the following example call, presupposing a Minimum Vertex Cover instance with the file name `instance.lp` and an encoding with the file name `mvc.lp` (cf. Section 4.2.4), instructing D-FLAT to enumerate all optimal solutions (which also prints their number and costs):

```
dflat -p opt-enum -e edge mvc.lp < instance.lp
```

4.1.3 Interface to ASP Programs: Reserved Predicates

The user's program communicates with D-FLAT by means of reserved predicates. These are partitioned into two sets:

- *Input predicates* are provided by D-FLAT as input for the user's program and describe the relevant bags and child tables.

Argument	Meaning
`-e` *hyperedge_predicate*	The predicate *hyperedge_predicate* in the input denotes a hyperedge connecting the argument vertices. At least one `-e` argument must be given.
`-j` *join_program*	*join_program* is the file name of an ASP program that is to be executed in join nodes of (semi-)normalized tree decompositions. This is only allowed if `-n semi` or `-n normalized` is present. If it is omitted and an exchange program is given via `-x`, the default join implementation will be used.
`--multi-level`	If this option is present, multi-level characteristics can be used. If it is absent, D-FLAT uses a more efficient implementation that can, however, only be used for single-level characteristics.
`-n` *normalization*	*normalization* specifies the normalization to be performed. Possible values are `none` (default), `semi` and `normalized`.
`--only-decompose`	If specified, D-FLAT terminates after decomposing and, if requested by `--stats`, printing statistics.
`-p` *problem_type*	*problem_type* specifies the type of the problem to be solved. Possible values are `enumeration` (default), `counting`, `decision`, `opt-enum`, `opt-counting` and `opt-value`.
`-s` *seed*	This sets the seed of the pseudo-random number generator used for the tree decomposition heuristic to *seed*.
`--stats`	This prints statistics about the constructed tree decomposition.
`-x` *exchange_program*	*exchange_program* is the file name of an ASP program that is to be executed in exchange nodes of (semi-)normalized tree decompositions. This is only allowed if `-n semi` or `-n normalized` is present.
program	*program* is the file name of an ASP program that is to be executed in each tree decomposition node. This is incompatible with the `-x` and `-j` options.

Table 4.1 Command-line options for D-FLAT

Input predicate	Meaning
`root`	The current node is the root node.
`childNode`(N)	N is the identifier of a child node.
`childBag`(N, V)	V is a vertex contained in the bag of the child node N.
`current`(V)	V is an element of the current bag.
`introduced`(V)	V is a current vertex but was in no child node's bag.
`removed`(V)	V was in a child node's bag but is not in the current one.
`childRow`(R, N)	R is the identifier of a row in the table of child node N.
`sub`(R, S)	If R is the identifier of a child row, S is the identifier of an item set that is a child of that row's top-level item set. Otherwise, R is the identifier of a subsidiary item set and S is the identifier of an item set that is a child of R.
`childItem`(S, I)	If S is the identifier of a child row, that row's top-level item set contains I. Otherwise, S is the identifier of a subsidiary item set containing I.
`childCount`(R, C)	C is the number of partial solutions corresponding to the child row R.
`childCost`(R, C)	C is the total cost of the partial solution corresponding to the child row R.

Table 4.2 Predicates supplied to the user's program by D-FLAT

- *Output predicates* occurring in an answer set instruct D-FLAT to store certain information in the table currently under consideration.

These predicates are summarized in Tables 4.2 and 4.3, respectively. Their usage will be extensively illustrated in Section 4.2.

For better readability, we use colors to highlight input predicates and output predicates in our listings.

4.2 Case Studies

To show that D-FLAT can indeed be applied to various problems from different domains, in this section we present a collection of case studies. Some of the examples in this section have also been treated in [Bliem et al., 2012]. The current work, however,

Output predicate	Meaning
`item`(I)	Let I be in the top-level item set.
`extend`(R)	Declare R to be the identifier of a child row that is extended by the currently described one. This is required for enumerating solutions.
`count`(C)	Let C be the number of partial solutions the currently described row corresponds to. This is required for counting problems if `extend/1` is not used.
`cost`(C)	Let C be the total cost of the current partial solution. This is required for optimization problems.
`currentCost`(C)	Let C be the local cost of the current table row. This is only required when solving an optimization problem and using the default join implementation for semi-normalized tree decompositions. See [Bliem et al., 2012] for details.
`levels`(L)	When using multi-level characteristics, declare L to be the number of levels of the root-to-leaf path (within a characteristic) described by the current answer set.
`item`(L, I)	Let I be in the item set at level L of the multi-level characteristic. It holds that $L \geq 0$, with 0 being the top level.
`extend`(L, S)	Declare that the item set at level L described by this answer set stems from the item set S from a child characteristic. This is used for constructing the tree of the characteristic as described in Section 3.1.2.

Table 4.3 Predicates in the ASP program's answer sets recognized by D-FLAT

generalizes those algorithms to work on non-normalized tree decompositions.

Sections 4.2.1, 4.2.2 and 4.2.3 are devoted to graph coloring problems where the latter section gives a first example of an algorithm that uses multi-level characteristics. Subsequently, in Section 4.2.4, we display an algorithm for the Minimum Vertex Cover problem. What is especially interesting about this is that we we show how a classical optimization problem can be solved. Section 4.2.5 covers the Boolean Satisfiability problem to illustrate that also problems whose instances are not directly graphs – in this case a problem in logic – can be solved. This is taken further in Section 4.2.6 where we list an algorithm that solves Disjunctive Asp. Section 4.2.7 is about the Cyclic Ordering problem and discusses issues concerning the discrepancy between solving decision problems and solving counting or enumeration problems. In Section 4.2.8, we present an algorithm for the Hamiltonian Cycle problem which is especially interesting because it cannot be expressed in MSO over just the vocabulary $\{\text{edge}\}$. To prove that D-FLAT is powerful enough for all MSO-definable problems, in Section 4.2.9 we display an implementation of the algorithm for Mso Formula Evaluation that we have presented in Section 3.3.1. Finally, Section 4.2.10 covers another problem that is PSPACE-complete in general, namely Qsat.

4.2.1 Graph Coloring

The 3-Col problem is defined as follows.

Input:	A graph $G = (V, E)$
Question:	Is there a proper 3-coloring of G, i.e., a mapping $f : V \to \{\text{red}, \text{green}, \text{blue}\}$ such that for each edge $(a, b) \in E$ it holds that $f(a) \neq f(b)$?

We have already seen a dynamic programming algorithm for 3-Col in Section 2.4.3. The D-FLAT encoding in Listing 4.1 implements an algorithm following the presented approach. Each answer set here constitutes exactly one new row in the current table. The output predicate `item/1` is used to specify the (partial) coloring encoded in the new row. We thereby assign a color to each vertex in the current bag. The output predicate `extend/1` indicates which child rows are the predecessors of the new row. Of course, `extend/1` must only have one child row identifier for each child node as its extension. This predicate is required for the reconstruction of complete solutions after all tables have been computed and could therefore be omitted if just the decision problem should be solved.

This program suffices to solve 3-Col with D-FLAT and exhibits linear runtime for graphs of bounded treewidth. Notice, in particular, that parts of it very much resemble a monolithic program as presented in Section 2.3.4. The most notable difference is that instead of guessing a coloring for all vertices at once, our D-FLAT encoding only

Listing 4.1 A D-FLAT encoding for 3-Col

```
color(red;green;blue).
% Select one row per child node for extension
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
% Retain coloring of vertices that have not been removed
item(map(V,C)) ← extend(R), childItem(R,map(V,C)),
    current(V).
% Joined rows must coincide on common vertices
← item(map(V,C0;C1)), C0 ≠ C1.
% Guess coloring of introduced vertices
1 { item(map(V,C)) : color(C) } 1 ← introduced(V).
% Ensure that the resulting coloring violates no constraints
← edge(X,Y), item(map(X;Y,C)).
```

guesses a coloring for the *introduced* vertices which augments "inherited" colorings of the other current vertices.

4.2.2 List Coloring

The List Coloring problem generalizes 3-Col in the following way:

Input:	A graph $G = (V, E)$ and for each vertex $v \in V$ a list of colors $l(v)$
Task:	Is there a proper coloring of G such that each vertex $v \in V$ is assigned a color in $l(v)$?

In [Szeider, 2010], the author proves that this problem is not fixed-parameter tractable w.r.t. treewidth. Therefore, we cannot express it in MSO. Despite this, we can solve this problem with D-FLAT by a trivial modification of the program from Section 4.2.1. The allowed colors now depend on the vertex to be colored, so the input facts must, in addition to the graph, also encode the list of allowed colors for each vertex, say by turning `color/1` into a binary predicate where the first argument denotes the vertex and the second one of its allowed colors.

The modified program is given in Listing 4.2. Note that the only difference to Listing 4.1 is that now the colors are no longer part of the D-FLAT program, and we have replaced `color`(C) by `color`(V, C).

Of course, this algorithm no longer features linear runtime for graphs of bounded treewidth. The reason for this is that the size of a table can now be exponential in the input size because there can be linearly many colors for a vertex.

Listing 4.2 A D-FLAT encoding for List Coloring

```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
item(map(V,C)) ← extend(I), childItem(I,map(V,C)),
   current(V).
← item(map(V,C0;C1)), C0 ≠ C1.
1 { item(map(V,C)) : color(V,C) } 1 ← introduced(V).
← edge(X,Y), item(map(X;Y,C)).
```

What is interesting about this algorithm is that it proves that it can also be reasonable to apply *Decompose, Guess & Check* to problems that are provably not fixed-parameter tractable. We expect that the decomposition nevertheless allows for substantial speedups when compared to a monolithic algorithm.

4.2.3 Minimum 3-Coloring

The Minimum 3-Coloring problem is a variant of 3-Col where additionally subset-minimality with respect to the set of red vertices (from a subset of all vertices that determine which are critical for minimization) is taken into account. A 3-coloring f is *red-minimal* w.r.t. a set of critical vertices W if and only if there is no 3-coloring f' such that $\{v \in W \mid f'(v) = \text{red}\} \subset \{v \in W \mid f(x) = \text{red}\}$. Minimum 3-Coloring is defined as follows.

Input: A graph $G = (V, E)$, a set $W \subseteq V$ and a vertex $q \in W$
Question: Is q colored with "red" in a red-minimal 3-coloring of G w.r.t. W?

We have introduced this problem in Section 3.1.3, proved its Σ_2^p-completeness and presented a *Decompose, Guess & Check* algorithm there. For a description of the algorithm, we refer to that section.

An implementation in D-FLAT could look as in Listing 4.3. The input graph is again given by means of `vertex/1` and `edge/2`, and additionally the set of critical vertices (i.e., the vertices where we want to minimize the color red) is given by means of `criticalVertex/1`. Note that, because we are dealing with multi-level characteristics, the predicates `extend/2` and `item/2` are binary and each answer set must specify the number of its declared item sets by means of `levels/1`. Another difference to the case where characteristics consist of only a single level is that here the predicate `sub/2` is provided by D-FLAT to the user's encoding such that `sub`(R, S) means that the item set with the identifier S is a subordinate item set of the item set identified by R.

The marker that we denoted by a $\times$ symbol in the column $\subset$ of the tables in Figure 3.3c is realized by means of an item `smaller` that is contained in a subsidiary

Listing 4.3 A D-FLAT encoding for Minimum 3-Coloring

```
color(red;green;blue).
% Red is the color that we want to minimize (w.r.t. set inclusion)
criticalColor(red).
% Each answer set encodes a top-level item set and a subsidiary one
levels(2).
% Guess a root-to-leaf path in a characteristic for every child node
1 { extend(0,R) : childRow(R,N) } 1 ← childNode(N).
1 { extend(1,S) : sub(R,S) } 1 ← extend(0,R).
item(L,map(V,C)) ← extend(L,S), childItem(S,map(V,C)),
    current(V).
% Only join matching colorings
← item(L,map(V,C0;C1)), C0 ≠ C1.
% Guess colorings for the introduced vertices
1 { item(L,map(V,C)) : color(C) } 1 ← introduced(V), L=0..1.
← edge(X,Y), item(L,map(X;Y,C)).
% Level 1 coloring must not be bigger (w.r.t. critical color and critical
    vertices) than level 0
← criticalColor(C), criticalVertex(V), item(1,map(V,C)),
    not item(0,map(V,C)).
% Inherit (or extend) markers indicating that the level 1 coloring is smaller
item(1,smaller) ← extend(1,S), childItem(S,smaller).
item(1,smaller) ← criticalColor(C), criticalVertex(V),
    item(0,map(V,C)), not item(1,map(V,C)).
% Make sure that eventually only minimal colorings survive
← root, extend(0,R), sub(R,S), childItem(S,smaller).
```

item set if and only if there would be such a marker in the table, indicating that the subsidiary item set witnesses that the coloring represented by the top-level item set is not minimal.

The last line of Listing 4.3 makes sure that in the root all rows are eliminated that would lead to colorings that are not minimal. This is realized by a constraint enforcing that non-minimal colorings (i.e., rows that have a subsidiary item set that contains `smaller`) are not extended.

Note that although extension pointers are given for both level 0 and level 1, D-FLAT disregards any subsidiary levels when finally materializing solutions. In other words, only the colorings associated with the top level are constructed.

4.2.4 Minimum Vertex Cover

The case studies outlined so far suffice to instruct D-FLAT to solve enumeration, counting and decision problems. Often it is also desired to solve optimization problems, for which additional information regarding the cost of a (partial) solution must be supplied in the encodings. To illustrate this, we show how the "drosophila" of fixed-parameter algorithms, the Minimum Vertex Cover problem, can be solved.

A *vertex cover* of a graph is a subset of the vertices that contains at least one endpoint of each edge. Minimum Vertex Cover is the following problem:

Input: A graph Task: Determine the smallest size of all vertex covers of the graph.

We have already encountered this problem in Section 2.2.2, where we presented an efficient algorithm to solve the problem on trees, and in Section 3.1.2, where we illustrated the concept of table rows and partial solutions by means of a dynamic programming algorithm on tree decompositions for Minimum Vertex Cover that proceeds like the one presented in this section.

The edges of the input graph are declared via `edge/2`. The D-FLAT encoding in Listing 4.4 implements an algorithm that computes the tables as illustrated in Figure 3.1.

Note that in a monolithic ASP encoding for Minimum Vertex Cover, we would use a `#minimize` statement and thus leave it to the ASP solver to filter out suboptimal solutions. However, it is a mistake to use such an optimization statement when writing the D-FLAT encoding because then one would filter out rows whose local cost might exceed that of others but which in the end would yield a better global solution.

The `cost/1` predicate is recognized by D-FLAT and specifies the cost of the partial solution. This number is computed by adding to the sum of costs of the preceding rows (which are provided by D-FLAT by means of `childCost/2`) the cost which is due to the contents of the current item set. However, since the current bag might overlap

Listing 4.4 A D-FLAT encoding for Minimum Vertex Cover

```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
% Only rows that agree on all common vertices may be joined
← extend(A;B), childItem(A,X), childRow(B,N),
   childBag(N,X), not childItem(B,X).
item(X) ← extend(R), childItem(R,X), current(X).
{ item(X) : introduced(X) }.
← edge(X,Y), current(X;Y), not item(X;Y).
% Compute part of cost due to vertices in the current bag
localCost(C) ← C = #count { item(_) }.
% Compute sum of costs of extended child rows
sumCosts(S) ← S = #sum [ extend(R) : childCost(R,C) = C ].
% Which part of this sum is due to current vertices?
inExtendedRow(V,R) ← current(V), extend(R), childItem(R,V).
numCounted(V,N) ← current(V),
   N = #count { inExtendedRow(V,_) }.
% We must subtract vertices that have been counted multiple times.
subtract(C) ← C = #sum [ numCounted(_,N) = N ].
cost(S-M+L) ← sumCosts(S), subtract(M), localCost(L).
```

with the bags of child nodes, a part of this cost might be counted multiple times and must therefore be subtracted again according to the inclusion-exclusion principle.

It can be observed that a large part of this program deals with the arithmetic for computing the cost – which is in fact not a difficult task but one for which ASP is not particularly well suited. Beside the intricacy of specifying these computations in ASP and making the code – which would be quite concise and easily understandable if it were not for the arithmetic – less readable, the performance of the program also suffers heavily because of this. It is clear that in the future we must provide facilities to support the user in such cases that are conceptually very easy but expose weaknesses of ASP.

In general though, especially optimization problems can be solved by the *Decompose, Guess & Check* approach with great advantages compared to a monolithic ASP program. The reason is that in optimization problems stopping after the first solution has been found is not an option for traditional solvers, since yet undiscovered solutions might have lower cost. Furthermore, traditional solvers (at least in the case of clasp) require two runs for counting or enumerating all optimal solutions: The first run only serves to determine the optimum cost, while the second starts from scratch and outputs all models having that cost. D-FLAT, in contrast, only requires one run at the end of which it immediately has all the information needed to determine the optimum cost and materialize exactly the best solutions.

In [Bliem et al., 2012] we showed that a version of this algorithm for Minimum Vertex Cover that works on semi-normalized tree decompositions quickly outperforms a monolithic encoding. There, the arithmetic poses no problem because exchange nodes have exactly one child and the join can be executed via a default implementation in the D-FLAT core, which is, so far, not implemented for non-normalized tree decompositions. We plan to overcome the current disadvantages on non-normalized tree decompositions in the future and expect that eventually, due to fewer nodes, a non-normalized approach will be even more efficient than the implementation presented in [Bliem et al., 2012].

4.2.5 Boolean Satisfiability

Until now, we have only considered problems where the input contains a graph that we subsequently used for decomposition. The *Decompose, Guess & Check* approach is, however, not limited to graph problems. It is only required to provide some *graph representation* of the input to be able to decompose it. To illustrate this, we consider the Sat problem.

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

Listing 4.5 A D-FLAT encoding for Sat

```
% Make explicit when an atom is false or a clause is unsatisfied
false(R,X) ← childRow(R,N), childBag(N,X),
    not childItem(R,X).
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
% Only join rows that coincide on common atoms
← extend(X;Y), atom(A), childItem(X,A), false(Y,A).
% Some child rows cannot be extended since they are killed
← clause(C), removed(C), extend(R), false(R,C).
% True atoms and satisfied clauses remain so unless removed
item(X) ← extend(R), childItem(R,X), current(X).
% Guess truth value of introduced atoms
{ item(A) : atom(A) : introduced(A) }.
% Through the guess, clauses may become satisfied
item(C) ← current(C;A), pos(C,A), item(A).
item(C) ← current(C;A), neg(C,A), not item(A).
```

For this problem, we can build a graph representation of an instance by constructing the formula's *incidence graph*. The incidence graph of a formula is the graph obtained by considering the clauses and variables as vertices and connecting a clause vertex with a variable vertex by an edge if and only if the respective variable occurs in the respective clause. In order for D-FLAT to construct such a graph from the input set of facts, the user is only required to state that, say, `pos/2` and `neg/2` indicate edges, where `pos`(C, A) denotes that the atom A is contained positively in the clause C, and `neg`(C, A) indicates that $\neg A$ is contained in C. Additionally, we declare clauses via `clause/1` and atoms via `atom/1`.

A dynamic programming algorithm for the Sat model counting problem working on tree decompositions of the incidence graph is given in [Samer and Szeider, 2010]. Listing 4.5 shows a possible D-FLAT encoding following that work's general idea of such an algorithm and generalizes it for non-normalized tree decompositions. It can be observed that, like for most problems, parts of the D-FLAT encoding bear some resemblance to a monolithic encoding (cf. Section 2.3.4). More precisely, a monolithic encoding is often the basis for that part of a D-FLAT encoding that deals with calculating the solutions for subproblems.

It should be noted that we primarily assign truth values to propositional atoms in the current bag like we would in a monolithic encoding. However, as a consequence of this, we also assign true or false to each clause in the current bag depending on

whether or not it is satisfied by the partial interpretation represented by the table row. We need this information on the status of a clause because, when a clause is removed, all rows not satisfying this clause must be eliminated.

4.2.6 Disjunctive Answer Set Programming

In Section 2.3, we defined the heads of ASP rules to consist of exactly one atom. If one allows rule heads to be arbitrary disjunctions of atoms, the complexity of deciding answer set existence increases in the propositional case from NP-complete to Σ_2^P-complete [Eiter and Gottlob, 1995]. This is because in the absence of disjunction, the Gelfond-Lifschitz reduct always possesses at most one least model that can be computed efficiently by means of a fixed-point operator, which is not possible if there are disjunctions in the head.

In this section, we will present a D-FLAT encoding that solves the Disjunctive Asp problem which we define as follows.

Input: A ground disjunctive logic program Π
Question: Does Π have an answer set?

Our encoding is influenced by [Jakl et al., 2009] where the authors present a dynamic programming algorithm on tree decompositions that solves this problem efficiently for bounded treewidth of the incidence graph. Directly implementing this algorithm with D-FLAT leads, however, to a relatively complicated program. This can be explained by the fact that, unsurprisingly, difficulties arise when using D-FLAT to implement an algorithm that was probably designed with a different problem solving paradigm in mind than *Decompose, Guess & Check*. That is is why our approach only relies on the basic ideas of that work and uses a slightly modified procedure.

Rules and atoms of the input program are declared by `rule/1` and `atom/1`, respectively. `pos`(R, A) (resp. `neg`(R, A)) denotes that the atom A occurs in the positive (resp. negative) body of the rule R, and `head`(R, A) indicates that A is in the head of R.

Listing 4.6 shows our encoding for Disjunctive Asp. It makes considerable use of the ideas presented in our encodings for Minimum 3-Coloring (Section 4.2.3) and Sat (Section 4.2.5) since Disjunctive Asp bears close resemblance to certain aspects of these problems such as only permitting subset-minimal solutions (as Minimum 3-Coloring) or using the incidence graph and eliminating rows when they do not satisfy removed rules (as for clauses in Sat).

4.2.7 Cyclic Ordering

In the case studies so far, we have always used a graph representation of the problem instance. For some problems, however, it can also be desired to use a hypergraph

Listing 4.6 A D-FLAT encoding for DISJUNCTIVE ASP

```
levels(2).
false(R,X) ← childRow(R,N), childBag(N,X),
    not childItem(R,X).
false(S,X) ← childRow(R,N), childBag(N,X), sub(R,S),
    not childItem(S,X).
% A child item set is "killed" when a removed rule is unsatisfied by it
killed(S) ← rule(X), removed(X), false(S,X).
1 { extend(0,R) : childRow(R,N) } 1 ← childNode(N).
1 { extend(1,S) : sub(R,S) } 1 ← extend(0,R).
← extend(L,X;Y), atom(A), childItem(X,A), false(Y,A).
% Some child item sets cannot be extended since they are killed
← extend(_,S), killed(S).
item(L,X) ← extend(L,S), childItem(S,X), current(X).
{ item(0;1,A) : atom(A) : introduced(A) }.
item(0,R) ← current(R;A), head(R,A),     item(0,A).
item(0,R) ← current(R;A), pos(R,A),  not item(0,A).
item(0,R) ← current(R;A), neg(R,A),      item(0,A).
item(1,R) ← current(R;A), head(R,A),     item(1,A).
item(1,R) ← current(R;A), pos(R,A),  not item(1,A).
% If a negative body atom is true on the top level, the rule disappears from
    reduct (w.r.t. the top level)
item(1,R) ← current(R;A), neg(R,A),      item(0,A). % (sic!)
← atom(A), item(1,A), not item(0,A).
item(1,smaller) ← extend(1,S), childItem(S,smaller).
item(1,smaller) ← atom(A), item(0,A), not item(1,A).
% Make sure that eventually only minimal models of the reduct survive
← root, extend(0,R), sub(R,S), childItem(S,smaller),
    not killed(S).
```

Listing 4.7 A D-FLAT encoding for Cyclic Ordering

```
% Guess an ordering of the current bag elements
1 { item(map(V,1..N)) } 1 ← current(V),
   N = #count { current(_) }.
← item(map(V0;V1,K)), V0 ≠ V1.
% Determine satisfied triples
lt(V0,V1) ← item(map(V0,K0)), item(map(V1,K1)), K0 < K1.
sat(A,B,C) ← order(A,B,C), lt(A,B), lt(B,C).
sat(A,B,C) ← order(A,B,C), lt(B,C), lt(C,A).
sat(A,B,C) ← order(A,B,C), lt(C,A), lt(A,B).
% All triples that cover all current bag elements must be satisfied
← order(A,B,C), current(A;B;C), not sat(A,B,C).
% Now check if we can find a compatible predecessor in each child table
gtChild(R,V0,V1) ← childItem(R,map(V0,K0);map(V1,K1)),
   current(V0;V1), K0 > K1.
incompatible(R) ← lt(V0,V1), gtChild(R,V0,V1).
match(N) ← childRow(R,N), not incompatible(R).
← childNode(N), not match(N).
```

representation. An example where we could proceed like this is the NP-complete Cyclic Ordering problem [Galil and Megiddo, 1977] that is defined as follows.

Input:	A set of vertices V and a set of triples $T \subseteq V^3$
Question:	Is there an ordering $<$ on V such that for each $(a, b, c) \in T$ it holds that $a < b < c$ or $b < c < a$ or $c < a < b$?

One can naturally represent an instance as a hypergraph where one considers the elements of V as vertices and the triples in T as hyperedges. This ensures that the vertices in each triple must appear together in at least one bag of any tree decomposition, and each constraint can therefore be checked by the user's program.

The D-FLAT program shown in Listing 4.7 decides Cyclic Ordering efficiently if the treewidth of the hypergraph representation is bounded.

Note that this algorithm can only be employed for the decision problem since combining a row with a predecessor does not make sense here. This is because each row determines an ordering that only considers the current bag. Therefore, a row only tells us something about the position of a bag element relative to the other bag elements, but it does not tell us anything about the position relative to the other vertices. If we

instead tried to assign each vertex a number from 1 up to the total number of vertices, we would violate the principles of dynamic programming because subproblems would then depend on the whole problem. Although the presented algorithm therefore does not allow us to combine the solutions of subproblems in order to obtain a solution for the whole problem, it suffices as a decision procedure. This is mirrored by the fact that the presented encoding does not make use of the `extend/1` predicate.

We plan to further investigate this problem in the future. Possible research directions include studying whether different representations of the instances (e.g., as incidence graphs) or alternative approaches for an algorithm could lead to fixed-parameter tractable counting or enumeration algorithms.

One possible approach would be to let the dynamic programming algorithm construct directed acyclic graphs (DAGs) such that a directed edge from a to b specifies that a precedes b in an ordering, and the DAGs are in this way consistent with the triples.[5] To finally count or enumerate solutions with such an approach, it would remain to find all *topological orderings* of the resulting DAGs, i.e., linearizations that are consistent with the edges in the DAG. This problem is also known as the topological sorting problem. Unfortunately, counting the number of topological orderings of a DAG is #P-complete [Brightwell and Winkler, 1991]. Interestingly, however, an algorithm that constructs DAGs that are consistent with the triples would make it possible to solve the *search problem* of Cyclic Ordering efficiently for bounded treewidth, since, given a DAG, finding *some* topological ordering is easy.

4.2.8 Hamiltonian Cycle

A Hamiltonian cycle in a (directed) graph is a (directed) cycle that encompasses all vertices without going through a vertex twice. We now present a D-FLAT encoding for the Hamiltonian Cycle problem which we define as follows.

Input:	A directed graph G
Question:	Is there a Hamiltonian cycle in G?

This problem is NP-complete [Karp, 1972] and fixed-parameter tractable when parameterized by the treewidth [Flum and Grohe, 2006]. What is especially interesting in this respect is that there is no MSO sentence over the vocabulary {edge} that expresses Hamiltonian Cycle, but the problem *can* be expressed over a vocabulary that allows statements about the incidence graph, i.e., in a setting where the domain consists not only of vertices but also of edges, and there are two unary predicates with the intended meaning that the argument is a vertex resp. edge, as well as a binary predicate saying that the first argument is an edge, the second one is a vertex, and this vertex is an endpoint of that edge.

Our D-FLAT encoding in Listing 4.8 has the following properties.

[5] We would like to thank Stefan Szeider for hints in this direction.

Listing 4.8 A D-FLAT encoding for Hamiltonian Cycle

```
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
% Do not join rows if a vertex would then have more than one successor
← extend(R;S), childItem(R,path(X,Y)),
   childItem(S,path(X,Z)), Y ≠ Z.
% Paths are different if they lead through previously removed vertices
← extend(R;S), R ≠ S, childItem(R;S,path(X,Y)),
   not childItem(R;S,to(X,Y)).
% A row is invalid if the start or end vertex of a path is removed
integrated(R,X) ← childItem(R,path(X,_);path(_,X)).
← extend(R), childRow(R,N), childBag(N,X), removed(X),
   not integrated(R,X).
% Retain information from children and account for removed vertices
item(to(X,Y)) ← extend(R), childItem(R,to(X,Y)),
   current(X;Y).
reachesViaRemoved(R,X,X) ← childRow(R,N), childBag(N,X).
reachesViaRemoved(R,X,Z) ← reachesViaRemoved(R,X,Y),
   removed(Y), childItem(R,path(Y,Z)).
item(path(X,Z)) ← extend(R), childItem(R,path(X,Y)),
   reachesViaRemoved(R,Y,Z), current(X;Z).
% If Y is successor of X, it must be so in each child containing both X and Y
← item(to(X,Y)), extend(R), childRow(R,N), childBag(N,X;Y),
   not childItem(R,to(X,Y)).
% Connect existing paths or introduced vertices
{ item(to(X,Y)) : edge(X,Y) : current(X;Y) }.
item(path(X,Y)) ← item(to(X,Y)).
% A guess is invalid if a vertex has more than one successor or predecessor
← item(path(X,Y;Z)), Y ≠ Z.
← item(path(X;Y,Z)), X ≠ Y.
path(X,Y) ← item(path(X,Y)).
path(X,Z) ← path(X,Y), item(path(Y,Z)).
% Disallow cycles not encompassing all current vertices
← path(X,X), current(Y), not path(X,Y).
```

- A *solution* is a Hamiltonian cycle.
- A *partial solution* is a set of paths such that each vertex has at most one predecessor and one successor.
- A solution *extends* a partial solution if each path in the partial solution is a subpath of the solution.
- In a *table row* we wish to represent the paths in a partial solution but must take into account that some vertices may have been removed. We must only state information regarding the current vertices and therefore distinguish between two kinds of items in a table row in order to express that parts of a path have been removed.
  - If a vertex X has a successor Y and both vertices are in the current bag, then we indicate this by the presence of an item $\texttt{to}(X, Y)$.
  - We say that a vertex X has a vertex Y as its "indirect successor" if there is a path $(X, r_1, \ldots, r_n, Y)$ with $n \geq 0$ such that all of $r_1, \ldots, r_n$ have been removed. If X has Y as its indirect successor and both X and Y are in the current bag, then an item $\texttt{path}(X, Y)$ is contained in the table row.

  It should be emphasized that a successor is also always an indirect successor. In other words, the presence of an item $\texttt{to}(X, Y)$ implies the presence of an item $\texttt{path}(X, Y)$.
- To compute the tables, we guess a combination of child rows such that for each child table a row is selected. We then check if this combination is valid. For this, the following conditions must hold.
  - It must not be the case that in one preceding row a vertex X has an (indirect) successor Y while in another preceding row X has a different (indirect) successor.
  - It must not be the case that two different preceding rows state that a vertex X has a proper indirect successor Y (i.e., that X has an indirect successor Y with at least one removed vertex in between) because this would indicate that the vertices in between are different, due to the connectedness condition.
  - No preceding row may describe a path that has a removed vertex as an endpoint.

  Subsequently, we extend these guessed preceding rows. For this, we retain all items $\texttt{to}(X, Y)$ and $\texttt{path}(X, Y)$ if both X and Y are still in the current bag, and for each pair of current bag elements X and Y, we store an item $\texttt{path}(X, Y)$ if Y can be reached from X (using the relation specified by the `path/2` items) via only

removed vertices. Additionally, we perform a guess that non-deterministically extends the paths by potentially appending introduced vertices or connecting paths that have so far been separated. Finally, we check that no vertex ends up having more than one predecessor or successor, and that there are no cycles that exclude some vertex.

- This procedure makes sure that in the root only rows survive that can be extended to valid Hamiltonian cycles. Therefore, Hamiltonian Cycle can be decided in linear time for graphs of bounded treewidth.

4.2.9 Evaluation of MSO Formulas

Recall the problem definition of the Mso Formula Evaluation problem for a fixed MSO formula ϕ.

Input:	A graph G
Question:	Does $G \models \phi$ hold?

In Section 3.3.1 we have presented an algorithm that solves this problem efficiently for graphs of bounded treewidth. We now show an implementation of this algorithm with D-FLAT. For a detailed description of how the algorithm works, we refer to that section. Listing 4.9 consists of auxiliary rules which are used in Listing 4.10 for the root-specific part and in Listing 4.11 which represents the core of the encoding.

In Listing 4.11, first for each child node a root-to-leaf path in a characteristic is guessed. Subsequent checks make sure that the guessed paths are compatible. Then introduced vertices are non-deterministically assigned to variables. The section of the encoding dealing with the `sat/1` predicate is responsible for determining which clauses become true under the current interpretation. Finally, the item sets are filled accordingly. The code in Listing 4.10 only comes into effect in the root of the tree decomposition and performs a labeling of the child characteristics followed by a check that makes sure that only rows correspond to valid solutions survive.

It should be noted that this is a rather complex implementation that is more of theoretical interest as a proof of concept. It shows that D-FLAT can indeed solve any MSO-definable problem efficiently for graphs of bounded treewidth and is thus a powerful realization of *Decompose, Guess & Check*. For many practical problems, the ASP code is usually much simpler, as the previous sections have illustrated.

4.2.10 Quantified Boolean Formulas

As another example for a problem that is PSPACE-complete in general, we present an encoding for Qsat (cf. Section 2.1.3) in Listing 4.12. The problem is defined as follows.

Listing 4.9 Auxiliary rules for the Mso Formula Evaluation encoding

```
% Associate item sets with their respective levels and nodes
itemSet(0,R,N) ← childRow(R,N).
itemSet(L+1,S,N) ← itemSet(L,R,N), sub(R,S).
assigned(S,X) ← childItem(S,is(X,_)).
assigned(S,X) ← childItem(S,taken(X)).
% Determine which individual variables are assigned by a guessed path
assignedByGuess(N,X) ← extend(L,S), itemSet(L,S,N),
   assigned(S,X).
isRemoved(S,X) ← childItem(S,is(X,V)), removed(V).
% Make implicit negative information explicit
-is(S,X,V) ← individualVar(X,L), itemSet(L,S,N),
   childBag(N,V), not childItem(S,is(X,V)).
-in(S,V,X) ← setVar(X,L), itemSet(L,S,N), childBag(N,V),
   not childItem(S,in(V,X)).
% Retain information from preceding rows
sat(C) ← extend(L,S), childItem(S,sat(C)).
in(V,X) ← extend(L,S), childItem(S,in(V,X)), current(V).
is(X,V) ← extend(L,S), childItem(S,is(X,V)), current(V).
taken(X) ← extend(L,S), isRemoved(S,X).
taken(X) ← extend(L,S), childItem(S,taken(X)).
```

Listing 4.10 Root-specific part of the MSO FORMULA EVALUATION encoding

```
% Determine which item sets are invalid, which are ∀-quantified and which
    are ∃-quantified
invalidItemSet(S) ← root, individualVar(X,L),
    itemSet(L,S,_), not assigned(S,X).
invalidItemSet(S) ← invalidItemSet(R), sub(R,S).
forallItemSet(S) ← root, childRow(S,_), sub(S,_),
    not invalidItemSet(S).
forallItemSet(S) ← existsItemSet(R), sub(R,S), sub(S,_),
    not invalidItemSet(S).
existsItemSet(S) ← forallItemSet(R), sub(R,S), sub(S,_),
    not invalidItemSet(S).
% Determine which item sets are "bad"
bad(S) ← root, levels(L), itemSet(L-1,S,_),
    not invalidItemSet(S), clause(C), not childItem(S,sat(C)).
bad(S) ← forallItemSet(S), sub(S,T), bad(T),
    not invalidItemSet(T).
-bad(S) ← existsItemSet(S), sub(S,T), not bad(T),
    not invalidItemSet(T).
bad(S) ← existsItemSet(S), not -bad(S).
% Only rows that are not "bad" may survive
← root, extend(0,R), bad(R).
% All individual variables must be assigned
← root, individualVar(X,_), not taken(X).
```

Listing 4.11 A D-FLAT encoding for MSO FORMULA EVALUATION to be used with the rules from Listings 4.9 and 4.10

```
1 { extend(0,R) : childRow(R,N) } 1 ← childNode(N).
1 { extend(L+1,S) : sub(R,S) } 1 ← extend(L,R), sub(R,_).
← extend(L,S0;S1), childItem(S0,is(X,V)), -is(S1,X,V).
← extend(L,S0;S1), childItem(S0,in(V,X)), -in(S1,V,X).
← extend(L,S0;S1), childItem(S0,is(X,Y)),
   childItem(S1,is(X,Z)), Y ≠ Z.
← extend(L,S0;S1), childItem(S0,is(X,_)),
   childItem(S1,taken(X)).
← extend(L,S0;S1), S0 ≠ S1, childItem(S0;S1,taken(X)).
% Account for introduced vertices
{ is(X,V) : individualVar(X,_) } ← introduced(V).
{ in(V,X) : setVar(X,_) } ← introduced(V).
← individualVar(X,_), is(X,V), is(X,W), V ≠ W.
← individualVar(X,_), is(X,_), taken(X).
% Determine clauses that become satisifed
sat(C) ← pos(C,eq(X,Y)), is(X,V), is(Y,V).
sat(C) ← neg(C,eq(X,Y)), is(X,V), is(Y,W), V ≠ W.
sat(C) ← neg(C,eq(X,Y)), extend(L,S), isRemoved(S,X),
   itemSet(L,S,N), not assignedByGuess(N,Y).
sat(C) ← neg(C,eq(X,Y)), extend(L,S), isRemoved(S,Y),
   itemSet(L,S,N), not assignedByGuess(N,X).
sat(C) ← pos(C,edge(X,Y)), is(X,V), is(Y,W), edge(V,W).
sat(C) ← neg(C,edge(X,Y)), is(X,V), is(Y,W), not edge(V,W).
sat(C) ← neg(C,edge(X,Y)), extend(L,S), isRemoved(S,X),
   itemSet(L,S,N), not assignedByGuess(N,Y).
sat(C) ← neg(C,edge(X,Y)), extend(L,S), isRemoved(S,Y),
   itemSet(L,S,N), not assignedByGuess(N,X).
sat(C) ← pos(C,in(X,Y)), is(X,V), in(V,Y).
sat(C) ← neg(C,in(X,Y)), is(X,V), not in(V,Y).
item(L-1,sat(C)) ← levels(L), sat(C).
item(L,is(X,V)) ← individualVar(X,L), is(X,V).
item(L,taken(X)) ← individualVar(X,L), taken(X).
item(L,in(V,X)) ← setVar(X,L), in(V,X).
```

Input:	A formula of the form $\exists X_1 \forall X_2 \exists X_3 \ldots Q X_i\ \phi$ where Q is $\exists$ for odd i and $\forall$ for even i, ϕ is a propositional formula and X_j (for $1 \leq j \leq i$) is a set of propositional variables
Question:	Is there a truth assignment to the variables in X_1 such that for any truth assignment to the variables in X_2 there is a truth assignment to the variables in X_3, and so on, such that ϕ is satisfied?

Although ASP only captures the class NP (cf. Section 2.3.3), we can solve PSPACE-complete problems such as this due to the fact that here multiple calls to an ASP solver are employed. QSAT is fixed-parameter tractable if both the treewidth and the number of quantifier alternations are bounded [Pan and Vardi, 2006].

The algorithm shares a few ideas with the one for MSO FORMULA EVALUATION in Section 4.2.9. In particular, because both problems deal with alternating quantifiers, the traversal of the characteristics to determine which rows must be eliminated is similar. An important difference is that here the formula is given in the input, whereas for MSO FORMULA EVALUATION the formula was fixed.

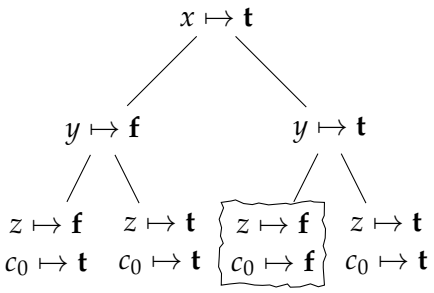
Although in both cases the characteristics are trees where each levels corresponds to a block of quantifiers, the contents of the characteristics are different. In the case of QSAT, we decompose the incidence graph of the (matrix of the) input formula as we did for the SAT problem (cf. Section 4.2.5) which is actually a special case of QSAT. In the current algorithm, an item set at level i contains a subset of the propositional atoms corresponding to the ith quantifier level that are present in the current bag. This subset indicates which of these atoms are interpreted as true by the respective root-to-leaf paths in the characteristic. We store in each leaf item set which of the clauses in the current bag are satisfied by the interpretation represented by the respective root-to-leaf path.

In contrast to the algorithm for MSO FORMULA EVALUATION, here we can dispose of rows not only at the root but we can detect that a row will not give rise to a solution at any node. That is why in this algorithm we perform a labeling of the trees of item sets from the child tables with "bad" (and, if appropriate, eliminate "bad" rows) not only at the root but at each node.

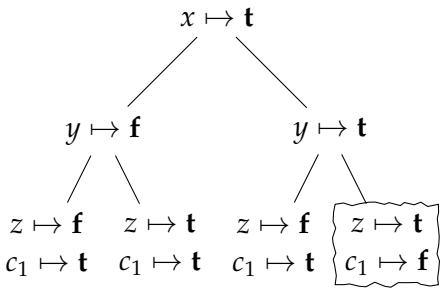
We make use of a special item called "fail" which we put into a leaf item set of a characteristic whenever a clause has been removed that was not satisfied by the respective root-to-leaf path. This additional item is required because sometimes we cannot eliminate a row from a child table only by looking at the labeling of its tree of item sets. To make this clearer, consider, for instance, a situation where a row originates from joining two child rows neither of which has its root labeled with "bad", as depicted in Figure 4.3. In this figure, labeling a node in the characteristic with "bad" is indicated by that node having ragged borders, and the word "fail" indicates that the special item is contained in the respective leaf item set. Observe that neither the row in Figure 4.3a nor the row in Figure 4.3b can be labeled with "bad" at the root because

Listing 4.12 A D-FLAT encoding for QSAT

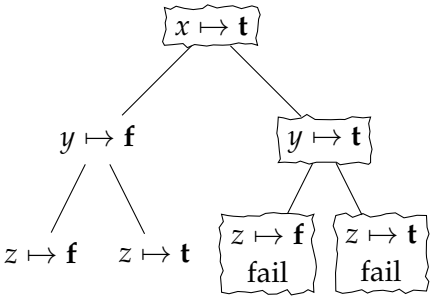
```
itemSet(0,R,N) ← childRow(R,N).
itemSet(L+1,S,N) ← itemSet(L,R,N), sub(R,S).
false(S,A) ← atom(L,A), itemSet(L,S,N), childBag(N,A),
    not childItem(S,A).
forallItemSet(S) ← childRow(S,_),                sub(S,_).
forallItemSet(S) ← existsItemSet(R), sub(R,S), sub(S,_).
existsItemSet(S) ← forallItemSet(R), sub(R,S), sub(S,_).
% An item set is "bad" either because an item indicates this...
bad(S) ← childItem(S,fail).
% ... or because it becomes "bad" due to clause removal
bad(S) ← clause(C), removed(C), levels(L),
    itemSet(L-1,S,N), childBag(N,C), not childItem(S,C).
bad(S)  ← forallItemSet(S), sub(S,T), bad(T).
-bad(S) ← existsItemSet(S), sub(S,T), not bad(T).
bad(S)  ← existsItemSet(S), not -bad(S).
1 { extend(L-1,S) : itemSet(L-1,S,N) } 1 ← childNode(N),
    levels(L).
extend(L-1,R) ← extend(L,S), sub(R,S).
% Do not extend rows that are "bad"
← extend(0,S), bad(S).
% Only join root-to-leaf paths that coincide on common atoms
← extend(L,X;Y), atom(L,A), childItem(X,A), false(Y,A).
% Retain true atoms and satisfied clauses unless they are removed, as well
    as the item indicating the item set is "bad"
item(L,X) ← extend(L,S), childItem(S,X), not removed(X).
% Add the item "fail" to the leaf item set if necessary
item(L,fail) ← levels(L+1), extend(L,S), bad(S).
% Guess truth value for introduced atoms
{ item(L,A) : atom(L,A) : introduced(A) }.
% Add clauses that become satisfied to the leaf item set
item(L-1,C) ← levels(L), pos(C,A), atom(LA,A),
    current(C;A), item(LA,A).
item(L-1,C) ← levels(L), neg(C,A), atom(LA,A),
    current(C;A), not item(LA,A).
```



(a) Row where only the third leaf is "bad"



(b) Row where only the fourth leaf is "bad"



(c) The row resulting from joining the rows from (a) and (b)

Figure 4.3 Rows from two different child tables and the row resulting from joining them to illustrate why the "fail" item is required

in either case *there is* an assignment to x (namely "true") such that *for all* assignments to y *there is* an assignment to z such that all clauses in the current bag are satisfied. When guessing a root-to-leaf path for each of these characteristics, we must not make the mistake of eliminating guesses that contain some "bad" item set for if we did that we would lose the entire subtree from Figure 4.3c that is rooted at $\{y \mapsto \mathbf{t}\}$. If we were then to label the resulting tree (which would only have two root-to-leaf paths), we would wrongly conclude that the root is not to be labeled with "bad" because there is an assignment to x (namely "true") such that for all assignments to y (namely just "false" since the other possibility has been lost) there is an assignment to z satisfying all clauses so far. This is why we require the special item "fail" that allows us to obtain the correct characteristic as depicted in Figure 4.3c and subsequently correctly classify the row as invalid.

4.3 Practical Performance

From a theoretical point of view, as has been shown in Section 4.2.9, D-FLAT meets all the prerequisites for being successfully employed in many practical cases where the treewidth is small. Of course, it remains to investigate its practical performance.

In [Bliem et al., 2012] we studied the performance of a prototype of D-FLAT for algorithms on (semi-)normalized tree decompositions. Note that the current version of D-FLAT still supports (semi-)normalization, so these results still apply in principle although the encodings used in that paper are different to the ones presented here. We briefly give an overview of these results.

Traditional ASP solvers employ clever heuristics to quickly either find some model or detect unsatisfiability, thereby being able to solve the decision variant of problems particularly well. In contrast, the dynamic programming approach of D-FLAT currently always calculates complete tables, whatever the problem variant may be – it is only in the final materialization stage that solutions are assembled differently, depending on the problem type. This motivates studying ways to improve D-FLAT's performance in such cases in the future.

3-Col and Sat are prime examples of problems where traditional ASP solvers are very successful in solving the decision variant efficiently. However, when it no longer suffices to merely find *some* solution (e.g., when dealing with counting or enumeration problems), the decomposition exploited by D-FLAT pays off for small treewidth, especially when there is a great number of solutions since traditional solvers have to explore huge parts of the exponential search space.

Where D-FLAT often excelled were counting problems. For instance for Sat, it could solve many instances in a matter of seconds, while the monolithic program ran out of time soon. Standard ASP-solvers do not provide a dedicated functionality for counting and thus have to implicitly enumerate all answer sets which leads to an explosion of runtime with increasing instance size, whereas it could be observed that

D-FLAT's runtime remained almost unaffected on average.

Although most of the time traditional ASP-solvers perform very well on decision problems, for some problems they have more difficulties, in particular when the grounding becomes huge. Our investigations showed that for the Cyclic Ordering problem, D-FLAT often outperformed the monolithic program, but it could also be observed that D-FLAT's running time is heavily dependent on the structure of the constructed tree decomposition.

Optimization problems like Minimum Vertex Cover proved to be a strong suit of D-FLAT. In general, when dealing with optimization problems, stopping after the first solution has been found is not an option for traditional solvers, since yet undiscovered solutions might have lower cost. Another advantage of D-FLAT is that traditional solvers (at least in the case of clasp) require two runs for counting or enumerating all optimal solutions: The first run only serves to determine the optimum cost, while the second starts from scratch and outputs all models having that cost. D-FLAT, in contrast, only requires one run at the end of which it immediately has all the information needed to determine the optimum cost and materialize exactly the best solutions.

We can conclude that D-FLAT is particularly successful for optimization, counting and enumeration problems (provided the treewidth is small), especially when the number of solutions or the size of the monolithic grounding explodes.

As we have mentioned, the results in [Bliem et al., 2012] only apply to (semi-)normalized tree decompositions. Since then, we have generalized our approach to also support non-normalized decompositions. However, this more general implementation currently lacks facilities like the default implementations for join nodes that we used in that work. We plan to provide default implementations also for the non-normalized case in the future. With such optimizations, we anticipate that D-FLAT will be more efficient on non-normalized tree decompositions than on semi-normalized ones. Therefore, we expect to further improve our results from that paper.

Chapter 5

Conclusion

This chapter concludes the thesis by reviewing and discussing the obtained results in Section 5.1, where we also compare *Decompose, Guess & Check* with related approaches and list possible future work. Finally, we summarize the most significant contributions in Section 5.2.

5.1 Discussion

This section is devoted to a discussion of the obtained results. First, in Section 5.1.1, we critically review the achievements. Section 5.1.2 then compares *Decompose, Guess & Check* with related approaches. In Section 5.1.3, we present open questions and tasks that might be worth investigating in the future.

5.1.1 Reflection

In this section, we reflect on the most notable theoretical and practical results of the current work.

General approach. In this work, we have investigated a declarative problem solving technique which we call *Decompose, Guess & Check*. It combines structural decomposition methods with the *Guess & Check* paradigm. In particular, problem instances are decomposed by constructing tree decompositions, and the problems are solved via dynamic programming on this data structure. The concrete details of such dynamic programming algorithms are specified using ASP. The natural expression of non-deterministic choices constitutes a key feature of ASP and it is especially this *Guess & Check* approach that makes ASP well suited for many hard problems.

We have seen in the presented examples that the benefits of ASP that make it an excellent choice for modeling many hard problems often carry over to a setting where

we employ it not only for specifying programs for solving these problems in a traditional sense (like in a monolithic problem solving setting) but where we additionally combine partial solutions following the concept of dynamic programming on tree decompositions.

Compared with corresponding implementations in imperative programming languages, ASP usually leads to more succinct and maintainable code thanks to its declarative nature. Our presented software framework facilitates development of such algorithms and makes it possible to perform rapid prototyping of algorithms for decomposed problems. Aside from the advantages which result from our use of ASP for the problem-specific parts, our framework aids the user by taking care of tasks that form the basis of algorithms for decomposed problems. Until now, technical obstacles such as the need for much code unrelated to the actual problem stood in the way of quickly implementing an idea for such an algorithm.

Normalizations of tree decompositions. In the early stages of the current work, we considered only algorithms on (semi-)normalized tree decompositions where nodes have at most two children and the bags of join nodes and its two children are always equal. We therefore distinguished two separate programs that had to be supplied by the user; which one was executed depended on the current node type. This first approach resulted in the publication [Bliem et al., 2012] where we were already able to deliver some promising experimental results.

One of the motivations why we further generalized our method is that the division of node types can be criticized for being artificial, and having only a single ASP program for all nodes would make the approach thus appear more natural. Additionally, this will eventually lead to better performance – although, from a theoretical point of view, normalizations do no harm since the decompositions only grow linearly, having to process less nodes of course usually means less runtime in practice.

While it can be argued that the approach is indeed more elegant on non-normalized tree decompositions, the practical performance is currently worse than on (semi-)normalized ones. This is mostly because in the more restricted case we can often employ a default implementation for join nodes that is written in C++ and is much more efficient. Another complication arises when dealing with optimization problems on non-normalized tree decompositions: The arithmetic required for determining the cost of partial solutions is more involved in this case and constitutes a task which exposes a weakness of ASP.

However, we believe that these issues can be overcome by providing default implementations for basic behavior of joins and cost computations. We suspect that in the end the non-normalized approach will be faster because joining, introducing and removing vertices at the same time should pose no problem for ASP and thus we could benefit from fewer nodes in the decomposition. Our approach therefore already offers promising performance if (semi-)normalized tree decompositions are employed, and

we expect that the non-normalized case will allow us to eventually even improve these results.

Problem types. Most of the problems we considered admit relatively straightforward algorithms that make it possible to enumerate solutions and thus also solve the counting and decision variants. We have, however, also encountered a problem (Cyclic Ordering) where we came up with an algorithm for just the decision variant. It remains to study whether this is because counting and enumeration for this problem are hard even for bounded treewidth or whether efficient algorithms for these variants can be found. It is conceivable that choosing a different graph representation of the instances might help. This would illustrate the importance of thinking carefully about how to represent instances as graphs, as there are usually multiple ways.

Table rows. It has turned out that choosing (trees of) item sets as the contents of the tables in the dynamic programming algorithms makes the *Decompose, Guess & Check* approach very flexible. In the early stages of the current work, we used a more restrictive structure for the tables where a row assigns some value to each current bag element. Owing to our generalization to arbitrary trees of item sets, it is now possible to formulate some problems more naturally (e.g., Hamiltonian Cycle) because sometimes we would like to store information about, say, edges rather than just vertices in a table row. Most importantly, however, algorithms such as Mso Formula Evaluation probably would not have been possible in a straightforward way if table rows were only allowed to encode mappings to the current vertices.

Applicability. A core contribution of this work is the result that Mso Formula Evaluation can be solved efficiently with *Decompose, Guess & Check* for graphs of bounded treewidth. We have implemented an algorithm for this problem in our framework and thus showed that our approach is able to solve any MSO-definable problem.

In fact, a correctness proof of our algorithm for Mso Formula Evaluation, for which we have given a sketch, amounts to an alternative proof of Courcelle's theorem [Courcelle, 1990]. What makes our approach interesting in this respect is that, in contrast to traditional proofs, it does not require relatively complex concepts such as tree automata or Ehrenfeucht-Fraïssé games. Furthermore, we anticipate that our alternative approach is well suited for obtaining extensions of Courcelle's theorem, for instance like the well-known extension for optimization problems [Arnborg et al., 1991].

In addition to showing that *Decompose, Guess & Check* is applicable to all MSO-definable problems, we have presented an algorithm for a problem (List Coloring) that is provably not fixed-parameter tractable w.r.t. treewidth. We expect that decomposition often also pays off in such cases.

We suspect that *Decompose, Guess & Check* can also offer advantages compared to monolithic problem solving when facing problems that are intractable even for bounded treewidth. Exactly how useful it is in such cases remains an open question.

5.1.2 Related Work

Quite some effort has been put into the development of tools that solve MSO-definable problems or facilitate dynamic programming. We are, however, not aware of any tool that combines the key features of our approach, namely performing dynamic programming on an automatically-generated tree decomposition via an ad-hoc encoding that follows the *Guess & Check* paradigm.

One category of related tools is the one of general MSO solvers. Such programs are given just the input graph and an MSO specification of the problem. Usually, they then transform the formula into an automaton. For instance, this can be done by constructing a tree automaton as in the proof of Courcelle's theorem. The main problem of such direct implementations, however, is that memory problems occur in practice because a power set construction is required that leads to a state explosion [Gottlob et al., 2010]. Highly optimized solvers like MONA [Klarlund et al., 2002] try to construct automata in a clever way. There are other general MSO solvers that do not use automata, like Sequoia which also uses dynamic programming on tree decompositions for evaluating formulas.

Another category are tools that allow the user to provide specific algorithms that follow dynamic programming on tree decompositions, like SHARP [Morak, 2011] which also automatically constructs a tree decomposition for an input graph. This category also subsumes our approach. What sets it apart from MSO solvers is that these are only concerned with the problem of evaluating MSO formulas, whereas we offer a means to implement ad-hoc dynamic programming algorithms. If fixed-parameter tractability w.r.t. treewidth has been established for a problem by formulating it in MSO, such solvers therefore immediately offer an algorithm for that problem and therefore require very little work from the user. Nevertheless, MSO solvers are ignorant of properties of the particular problem at hand, so, by nature, none of these tools can exploit properties of the problem like a tailored algorithm can. We expect that our approach is therefore usually more efficient because the programmer of an algorithm is aware of the properties of the specific problem.

Finally, languages like Datalog or Dyna can be used to make the required dynamic programming less painful. There have also been developments within the Prolog community like tabled logic programming [Guo and Gupta, 2008] or B-Prolog [Zhou, 2012] which use memoization.

We will now briefly sketch some of these related approaches in more detail.

SHARP. The SHARP framework[1] [Morak, 2011] is a C++ library that allows the specification of particular dynamic programming algorithms on tree decompositions which are automatically computed by heuristic methods. The most notable difference to our approach is that the SHARP user has to provide the algorithm in C++, which is often more cumbersome than in ASP. D-FLAT is built on top of a modified version of SHARP. Modifications were made due to the following properties of the current version of SHARP:

- SHARP requires a solution to be a subset of the vertices. D-FLAT generalizes this such that a solution can now be a set of arbitrary items.
- SHARP only allows for decision, counting and enumeration problems but not for, e.g., optimization problems as implemented in D-FLAT.
- SHARP does not offer dedicated facilities for counting the number of solutions. Rather, it can only count the number of solutions if extension pointers are given.
- When enumerating solutions, SHARP materializes all solutions simultaneously which causes an explosion of memory. D-FLAT uses a lazy evaluation technique eliminating this issue.

Sequoia. Sequoia [Kneis et al., 2011, Langer et al., 2012] is a general tool that solves decision, optimization and search problems expressible in MSO, given an MSO formula describing the problem and a graph for which this formula is to be evaluated. It does so by following a game-theoretic approach by means of a model checking game where two players – the *verifier* and the *falsifier* – take turns. The verifier tries to show the formula to hold on the graph, whereas the falsifier tries to show the opposite.

Solving the MSO model checking problem is PSPACE-complete in general but fixed-parameter tractable w.r.t. treewidth when the formula is fixed, according to Courcelle's theorem. The motivation behind Sequoia is to exploit this fixed-parameter tractability while avoiding the problems of directly constructing tree automata. Sequoia therefore performs dynamic programming on tree decompositions to evaluate formulas – similar to our approach in Section 3.3.1.

Originally, Sequoia followed a "monolithic" algorithm to evaluate MSO formulas recursively by checking all possible assignments to the variables. This algorithm was then refined to use dynamic programming on tree decompositions and now features linear runtime for bounded treewidth. Experimental results were promising for some practical examples.

Monadic Datalog. Datalog is a language prominent in the field of deductive databases that, very roughly, corresponds to ASP with very restricted use of negation.[2]

[1] `http://www.dbai.tuwien.ac.at/research/project/sharp/`

[2] To be precise, Datalog only allows for *stratified negation* which we will not define here, however.

Hence, Datalog programs always have exactly one model. *Monadic Datalog* is a fragment of Datalog where all predicates occurring in rule heads are unary.

[Gottlob et al., 2010] show that each MSO-definable graph property can also be expressed as a program in (a special fragment of) Monadic Datalog – with the graph and a description of a normalized tree decomposition as input – that can be evaluated in linear time, in particular because an explosion of the grounding can be avoided.

The result the authors obtain is thus similar to Courcelle's theorem, except that Datalog is used instead of MSO as a specification language. The main advantage of Datalog over MSO is that, due to being declarative and possessing an operational semantics, not only the intuition behind the specified problems but also behind the dynamic programming algorithms is captured; thus Datalog combines benefits of MSO and tree automata (which are traditionally used for turning MSO formalizations into algorithms) while avoiding their respective weaknesses.

Despite these appealing results, the approach also has its downsides. The proposed Datalog programs usually make heavy use of set arithmetic which is not part of plain Datalog, and therefore they must be transformed into relatively cumbersome programs where set operations are instantiated by blowing up the arity of the predicates which is possible due to the bound on the treewidth. This "trick" thus leads to monstrous predicates as, given a bound on the treewidth of w, each set variable is replaced by $w + 1$ individual variables. This drawback could, however, at least be alleviated by providing built-in implementations of set arithmetic (e.g., in an imperative language) that are exposed to the Datalog program by special predicates. The ASP system *DLV-Complex* [Calimeri et al., 2009], for instance, provides such built-in predicates for set arithmetic.

Our approach is therefore arguably more elegant. Many of the advantages that apply for the Datalog approach also apply for ours because of the similarity of ASP and Datalog. Using ASP, however, gives the additional advantage of reflecting the intuition behind the problem even better because we can, for instance, use *Guess & Check* which is not possible in Datalog.

Note that one could also use ASP instead of Datalog and, additionally to the input graph, simply feed facts describing the tree decomposition to the user's program. This would make a tool like D-FLAT pointless to a certain extent. We initially performed experiments with such a method. To our surprise, it was very inefficient which is presumably because the ASP solver had problems with recognizing the bottom-up data flow of dynamic programming algorithms.

Dyna. Dyna [Eisner and Filardo, 2010] is a declarative programming language with roots in the field of natural language processing [Eisner et al., 2005, Eisner et al., 2004]. Originally, it was designed to allow for abstract specifications of dynamic programming algorithms in the context of statistical artificial intelligence, machine learning and natural language processing, with typical applications like parsing probabilistic

context-free grammars or calculating the edit distance between strings. It was later generalized and is now a Turing-complete programming language suited for a wide range of problems. In particular, the authors' goal is to provide a general-purpose weighted logic programming language that is well-suited for statistical artificial intelligence [Eisner and Filardo, 2010].

As the term "weighted logic programming language" indicates, Dyna's syntax is inspired by Prolog, but instead of Horn clauses it uses "Horn equations". That is, the user specifies a system of equations in a syntax reminding of Prolog, where the terms that constitute these equations are annotated with values – these are often weights which are typically used for, e.g., building parsers for probabilistic context-free grammars, but can even be arbitrary ground terms. This sets Dyna apart from traditional logic programming languages where atoms, like in Datalog or ASP, only carry truth values and are not associated with arbitrary structures.

Since Dyna is Turing-complete, termination is not guaranteed. If the program terminates, however, the computed values satisfy the system of equations given by the user.

Unlike Prolog, Dyna is not confined to backward chaining, i.e., starting with the goal that is to be proved and searching for conditions that can be fulfilled to infer that goal. Rather, the evaluation can also occur "bottom-up", i.e., starting with the available data and making inferences via forward-chaining. Which particular strategy is employed is not determined in advance. The declarative specification is independent of the choice of evaluation strategy. Thus, Dyna can be used for classical "bottom-up" dynamic programming techniques and "top-down" memoization.

Although Dyna can be employed for dynamic programming, it follows a different approach than *Decompose, Guess & Check* (albeit certainly being able to also construct tree decompositions and working on them in principle, due to Turing-completeness). Notably, it does not relieve the user of parsing the input and constructing a tree decomposition automatically. Further, for the actual computations of the dynamic programming algorithm, it does not use ASP which, as we have seen, often allows for a natural and succinct specification of many problems. Although being a general-purpose tool, Dyna is therefore not particularly geared to the *Decompose, Guess & Check* approach.

Situating Decompose, Guess & Check among related approaches. This section has shown that although many related approaches exist, none of them combines dynamic programming on tree decompositions with the *Guess & Check* paradigm of ASP. It can therefore be concluded that *Decompose, Guess & Check* represents a novel contribution in the field of declarative problem solving.

5.1.3 Future Work

There still are various open questions that we would like to pursue in the future. Here we list interesting possible research directions.

From a theoretical point of view, we would like to investigate the usefulness of *Decompose, Guess & Check* for problems that are intractable for bounded treewidth. We suspect that even in such cases some improvements can be made such that instances could be solved that are out of reach for current systems.

An interesting prospect is to investigate whether solving (certain classes of) non-ground ASP programs can be done via *Decompose, Guess & Check*. If we can find an algorithm for this, then an automatic translation procedure from a program for a monolithic setting to a dynamic programming algorithm working on tree decompositions is possible.

Regarding the implementation, the D-FLAT system should be decoupled from the SHARP framework. D-FLAT started as an application of SHARP but due to the desired generality of *Decompose, Guess & Check* modifications had to be made such that now we are using merely a stripped-down version of SHARP. The remaining parts – the code for constructing tree decompositions and for tree traversal – should be integrated into D-FLAT.

In order to eliminate the performance issues that are due to expensive join operations and arithmetic when computing solution costs, we would like to investigate options to improve this situation. For instance, default implementations for such tasks could be provided that fit many problems.

Since it is often cumbersome to debug dynamic programming algorithms, we would like to provide better support for this. Recently, a dedicated language was proposed in [Vos et al., 2012] for annotating ASP programs to allow for, e.g., specifying assertions and performing automated unit testing.

When materializing solutions, currently only the top level of multi-level characteristics is taken into account. For some problems it might be more reasonable to materialize whole trees of item sets.

D-FLAT currently computes all the tables bottom-up *in their entirety*, even though often if would suffice to only compute *some* of the rows until a solution can be found. For optimization problems, we expect that sometimes table rows that will not lead to optimal solutions can be identified early when we already know that there is a better solution. We would therefore like to integrate facilities that allow for a lazy evaluation of the tables. This would enable us to solve problems much more efficiently when they are concerned with merely deciding existence of (or searching for) *any* solution. We plan to study how especially *reactive ASP solving* [Gebser et al., 2011a] might allow for such a lazy evaluation.

Because decomposing a problem often leads to independent subproblems, there is obviously potential for improvements by exploiting today's multi-core architectures. Also, there has recently been progress in parallelizing even single calls to an ASP solver

[Gebser et al., 2012]. The current version of D-FLAT lacks parallelization facilities, however. We would like to investigate how it can best be augmented with proper parallelization features.

Finally, the system should be extensively benchmarked to determine where the most potential for optimization lies.

5.2 Summary

We have presented a method for declarative problem solving called *Decompose, Guess & Check*. Its primary feature is that instances are decomposed by means of tree decompositions which form the basis of subsequently solving the problems following a dynamic programming approach using ASP for the problem-specific computations. This way, the *Guess & Check* paradigm – which is one of the reasons ASP is an attractive language for specifying algorithms for many hard problems – is augmented by a decomposition step.

Using ASP as a language to specify the dynamic programming algorithms, *Decompose, Guess & Check* benefits from efficient solvers as well as from a rich language that allows for succinct, readable and maintainable code.

The analysis of our approach has shown it to be powerful enough to efficiently solve all MSO-definable problems on graphs of bounded treewidth. Furthermore, we have presented a software framework called D-FLAT that follows this method and makes rapid prototyping of dynamic programming algorithms working on tree decompositions possible. We are not aware of any previously existing systems that combine the concept of decomposition with the power and convenience of a declarative language like ASP. Hence, this work paves the way for making dynamic programming on tree decompositions more accessible, which is useful for educational purposes and research.

Decompose, Guess & Check is, however, not only attractive from a theoretical point of view. Following our method, it is possible to solve many NP-hard problems in linear time when the treewidth of graph representations of the input is bounded. Because instances in practical applications often exhibit small treewidth, our approach also has practical relevance. For many problems that are hard in general, *Decompose, Guess & Check* is thus a promising candidate for solving large instances that have so far been out of reach for existing ASP systems.

Bibliography

[Agarwal et al., 2011] Agarwal, R., Godfrey, P. B., and Har-Peled, S. (2011). Approximate distance queries and compact routing in sparse graphs. In *Proc. INFOCOM*, pages 1754–1762. IEEE.

[Alviano et al., 2011] Alviano, M., Calimeri, F., Faber, W., Ianni, G., and Leone, N. (2011). Function symbols in ASP: Overview and perspectives. In *Nonmonotonic Reasoning – Essays Celebrating Its 30th Anniversary*, pages 1–24. College Publications, London.

[Arnborg et al., 1987] Arnborg, S., Corneil, D. G., and Proskurowski, A. (1987). Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284.

[Arnborg et al., 1991] Arnborg, S., Lagergren, J., and Seese, D. (1991). Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340.

[Aschinger et al., 2011] Aschinger, M., Drescher, C., Gottlob, G., Jeavons, P., and Thorstensen, E. (2011). Structural decomposition methods and what they are good for. In *Proc. STACS*, volume 9 of *LIPIcs*, pages 12–28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Baral, 2003] Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

[Bliem et al., 2012] Bliem, B., Morak, M., and Woltran, S. (2012). D-FLAT: Declarative problem solving using tree decompositions and answer-set programming. *TPLP*, 12(4-5):445–464.

[Bodlaender, 1993] Bodlaender, H. L. (1993). A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22.

[Bodlaender, 1996] Bodlaender, H. L. (1996). A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317.

[Bodlaender, 1997] Bodlaender, H. L. (1997). Treewidth: Algorithmic techniques and results. In *Proc. MFCS*, volume 1295 of *LNCS*, pages 19–36. Springer.

[Bodlaender, 2005] Bodlaender, H. L. (2005). Discovering treewidth. In *Proc. SOFSEM*, volume 3381 of *LNCS*, pages 1–16. Springer.

[Bodlaender and Koster, 2010] Bodlaender, H. L. and Koster, A. M. C. A. (2010). Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275.

[Brewka et al., 2011] Brewka, G., Eiter, T., and Truszczyński, M. (2011). Answer set programming at a glance. *Commun. ACM*, 54(12):92–103.

[Brightwell and Winkler, 1991] Brightwell, G. and Winkler, P. (1991). Counting linear extensions is #P-complete. In *Proc. STOC*, pages 175–181. ACM.

[Calimeri et al., 2008] Calimeri, F., Cozza, S., Ianni, G., and Leone, N. (2008). Computable functions in ASP: Theory and implementation. In *Proc. ICLP*, volume 5366 of *LNCS*, pages 407–424. Springer.

[Calimeri et al., 2009] Calimeri, F., Cozza, S., Ianni, G., and Leone, N. (2009). An ASP system with functions, lists, and sets. In *Proc. LPNMR*, volume 5753 of *LNCS*, pages 483–489. Springer.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press, 3rd edition.

[Courcelle, 1990] Courcelle, B. (1990). The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75.

[Dantsin et al., 2001] Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425.

[Dasgupta et al., 2006] Dasgupta, S., Papadimitriou, C., and Vazirani, U. (2006). *Algorithms*. McGraw-Hill Higher Education.

[Denecker et al., 2009] Denecker, M., Vennekens, J., Bond, S., Gebser, M., and Truszczyński, M. (2009). The second answer set programming competition. In *Proc. LPNMR*, volume 5753 of *LNCS*, pages 637–654. Springer.

[Dermaku et al., 2008] Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B. J., Musliu, N., and Samer, M. (2008). Heuristic methods for hypertree decomposition. In *Proc. MICAI*, volume 5317 of *LNCS*, pages 1–11. Springer.

[Dorn and Telle, 2009] Dorn, F. and Telle, J. A. (2009). Semi-nice tree-decompositions: The best of branchwidth, treewidth and pathwidth with one algorithm. *Discrete Applied Mathematics*, 157(12):2737–2746.

[Downey and Fellows, 1999] Downey, R. G. and Fellows, M. R. (1999). *Parameterized Complexity*. Monographs in Computer Science. Springer.

[Eisner and Filardo, 2010] Eisner, J. and Filardo, N. W. (2010). Dyna: Extending datalog for modern AI. In *Proc. Datalog 2.0*, volume 6702 of *LNCS*, pages 181–220. Springer.

[Eisner et al., 2004] Eisner, J., Goldlust, E., and Smith, N. A. (2004). Dyna: A declarative language for implementing dynamic programs. In *Proc. ACL (companion volume)*, pages 218–221. The Association for Computer Linguistics.

[Eisner et al., 2005] Eisner, J., Goldlust, E., and Smith, N. A. (2005). Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc. HLT/EMNLP*. The Association for Computational Linguistics.

[Eiter and Gottlob, 1993] Eiter, T. and Gottlob, G. (1993). Propositional circumscription and extended closed world reasoning are Π_2^P-complete. *Theor. Comput. Sci.*, 114(2):231–245.

[Eiter and Gottlob, 1995] Eiter, T. and Gottlob, G. (1995). On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323.

[Eiter et al., 1997] Eiter, T., Gottlob, G., and Mannila, H. (1997). Disjunctive datalog. *ACM Trans. Database Syst.*, 22(3):364–418.

[Flum and Grohe, 2006] Flum, J. and Grohe, M. (2006). *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer.

[Galil and Megiddo, 1977] Galil, Z. and Megiddo, N. (1977). Cyclic ordering is NP-complete. *Theor. Comput. Sci.*, 5(2):179–182.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability*. W. H. Freeman.

[Gebser et al., 2011a] Gebser, M., Grote, T., Kaminski, R., and Schaub, T. (2011a). Reactive answer set programming. In *Proc. LPNMR*, volume 6645 of *LNCS*, pages 54–66. Springer.

[Gebser et al., 2010] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. (2010). A user's guide to gringo, clasp, clingo, and iclingo. Preliminary Draft. Available at `http://potassco.sourceforge.net`.

[Gebser et al., 2009] Gebser, M., Kaminski, R., Ostrowski, M., Schaub, T., and Thiele, S. (2009). On the input language of ASP grounder gringo. In *Proc. LPNMR*, volume 5753 of *LNCS*, pages 502–508. Springer.

[Gebser et al., 2011b] Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., and Schneider, M. T. (2011b). Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124.

[Gebser et al., 2012] Gebser, M., Kaufmann, B., and Schaub, T. (2012). Multi-threaded ASP solving with clasp. *TPLP*, 12(4-5):525–545.

[Gelfond and Leone, 2002] Gelfond, M. and Leone, N. (2002). Logic programming and knowledge representation – the A-Prolog perspective. *Artif. Intell.*, 138(1-2):3–38.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In *Proc. ICLP/SLP*, pages 1070–1080. The MIT Press.

[Gelfond and Lifschitz, 1991] Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386.

[Goldreich, 2008] Goldreich, O. (2008). *Computational Complexity – A Conceptual Perspective*. Cambridge University Press.

[Gottlob et al., 2002] Gottlob, G., Leone, N., and Scarcello, F. (2002). Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627.

[Gottlob et al., 2010] Gottlob, G., Pichler, R., and Wei, F. (2010). Monadic datalog over finite structures of bounded treewidth. *ACM Trans. Comput. Log.*, 12(1):3.

[Gramm et al., 2008] Gramm, J., Nickelsen, A., and Tantau, T. (2008). Fixed-parameter algorithms in phylogenetics. *Comput. J.*, 51(1):79–101.

[Grohe, 1999] Grohe, M. (1999). Descriptive and parameterized complexity. In *Proc. CSL*, volume 1683 of *LNCS*, pages 14–31. Springer.

[Guo and Gupta, 2008] Guo, H.-F. and Gupta, G. (2008). Simplifying dynamic programming via mode-directed tabling. *Softw., Pract. Exper.*, 38(1):75–94.

[Huang and Lai, 2007] Huang, X. and Lai, J. (2007). Parameterized graph problems in computational biology. In *Proc. IMSCCS*, pages 129–132. IEEE.

[Jakl et al., 2009] Jakl, M., Pichler, R., and Woltran, S. (2009). Answer-set programming with bounded treewidth. In *Proc. IJCAI*, pages 816–822.

[Karp, 1972] Karp, R. M. (1972). Reducibility among combinatorial problems. In *Proc. Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York.

[Klarlund et al., 2002] Klarlund, N., Møller, A., and Schwartzbach, M. I. (2002). MONA implementation secrets. *Int. J. Found. Comput. Sci.*, 13(4):571–586.

[Kloks, 1994] Kloks, T. (1994). *Treewidth: Computations and Approximations*, volume 842 of *LNCS*. Springer.

[Kneis et al., 2011] Kneis, J., Langer, A., and Rossmanith, P. (2011). Courcelle's theorem – a game-theoretic approach. *Discrete Optimization*, 8(4):568–594.

[Langer et al., 2012] Langer, A., Reidl, F., Rossmanith, P., and Sikdar, S. (2012). Evaluation of an MSO-solver. In *Proc. ALENEX*, pages 55–63. SIAM / Omnipress.

[Larson, 1967] Larson, R. E. (1967). A survey of dynamic programming computational procedures. *IEEE Trans. Automat. Contr.*, 12(6):767–774.

[Latapy and Magnien, 2006] Latapy, M. and Magnien, C. (2006). Measuring fundamental properties of real-world complex networks. *CoRR*, abs/cs/0609115.

[Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562.

[Lifschitz, 2008] Lifschitz, V. (2008). What is answer set programming? In *Proc. AAAI*, pages 1594–1597. AAAI Press.

[Marek and Remmel, 2003] Marek, V. W. and Remmel, J. B. (2003). On the expressibility of stable logic programming. *TPLP*, 3(4-5):551–567.

[Marek and Truszczyński, 1991] Marek, V. W. and Truszczyński, M. (1991). Autoepistemic logic. *J. ACM*, 38(3):588–619.

[Marek and Truszczyński, 1999] Marek, V. W. and Truszczyński, M. (1999). Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer.

[Melançon, 2006] Melançon, G. (2006). Just how dense are dense graphs in the real world? A methodological note. In *Proc. BELIV*, pages 1–7. ACM Press.

[Morak, 2011] Morak, M. (2011). dynASP – A dynamic programming-based answer set programming solver. Master's thesis, TU Wien, Vienna.

[Niedermeier, 2006] Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press.

[Niemelä, 1999] Niemelä, I. (1999). Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273.

[Pan and Vardi, 2006] Pan, G. and Vardi, M. Y. (2006). Fixed-parameter hierarchies inside PSPACE. In *Proc. LICS*, pages 27–36. IEEE Computer Society.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[Pichler et al., 2010] Pichler, R., Rümmele, S., and Woltran, S. (2010). Multicut algorithms via tree decompositions. In *Proc. CIAC*, volume 6078 of *LNCS*, pages 167–179. Springer.

[Robertson and Seymour, 1984] Robertson, N. and Seymour, P. D. (1984). Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64.

[Samer and Szeider, 2010] Samer, M. and Szeider, S. (2010). Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64.

[Schlipf, 1995] Schlipf, J. S. (1995). The expressive powers of the logic programming semantics. *J. Comput. Syst. Sci.*, 51(1):64–86.

[Szeider, 2010] Szeider, S. (2010). Not so easy problems for tree decomposable graphs. In *Proc. ICDM 2008*, number 13 in Lecture Notes Series, pages 179–190. Ramanujan Mathematical Society.

[Thorup, 1998] Thorup, M. (1998). All structured programs have small tree-width and good register allocation. *Inf. Comput.*, 142(2):159–181.

[Turing, 1936] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265.

[Vos et al., 2012] Vos, M. D., Kisa, D. G., Oetsch, J., Pührer, J., and Tompits, H. (2012). Annotating answer-set programs in Lana. *TPLP*, 12(4-5):619–637.

[Zhou, 2012] Zhou, N.-F. (2012). The language features and architecture of B-Prolog. *TPLP*, 12(1-2):189–218.