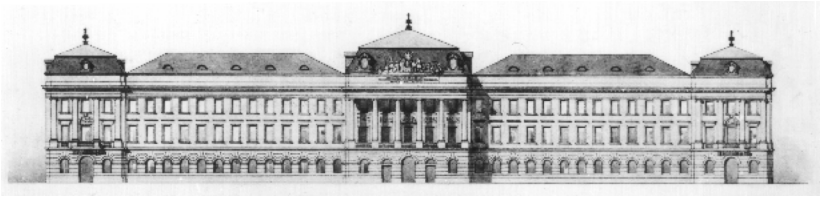
dbai

TECHNICAL REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

The D-FLAT System: User Manual

DBAI-TR-2017-107

DBAI TECHNICAL REPORT

2017

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18493
sek@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

TU WIEN
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI TECHNICAL REPORT
DBAI TECHNICAL REPORT DBAI-TR-2017-107, 2017

The D-FLAT System: User Manual

Bernhard Bliem [1] **Marius Moldovan** [1] **Stefan Woltran** [1]

Abstract. D-FLAT is a software system that combines Answer Set Programming (ASP) with problem solving on tree decompositions and can serve as a rapid prototyping tool for designing dynamic programming algorithms over tree decompositions in a fully declarative way. In terms of expressibility, it was shown that using D-FLAT can solve any problem expressible in monadic second-order logic in linear time given that the problem instance has bounded treewidth. In this report, we summarize all features of D-FLAT and highlight functionality that has been added recently. We give a comprehensive overview of our system and provide several examples.

[1]TU Wien. E-mail: {bliem,moldovan,woltran}@dbai.tuwien.ac.at

Acknowledgements: This work has been supported by the Austrian Science Fund (FWF) under grants P25607 and Y698, and by the Vienna University of Technology special fund "Innovative Projekte" (9006.09/008).

Contents

1 Introduction

Complex reasoning problems over large amounts of data arise in many of today's application domains for computer science. Bio-informatics, where structures such as proteins or genomes have to be analyzed, is one such domain; querying ontologies like SNOMED-CT is important in medicine, another such domain. Applications like the ones mentioned provide a great challenge to push the broad selection of logical methods from Artificial Intelligence and Knowledge Representation toward practical use. To successfully face this challenge, the following considerations appear crucial.

First, for formalizing and implementing complex problems, *declarative approaches* are desired. Not only do they lead to readable and maintainable code (compared to C code, for instance), they also ease the discussion with experts from the target domain when it comes to specifying their particular problems. Database query languages, which serve this purpose well in the business domain, are often too weak to capture concepts required in other domains, for instance, reachability in structures. A particular candidate for such an advanced declarative approach is Answer Set Programming (ASP) [23, 39] for which there are sophisticated solvers available that offer high efficiency and rich languages for modeling the problems at hand. A particular feature of ASP is the so-called *Guess & Check* methodology, where a *guess* is performed to open up the search space non-deterministically and a subsequent *check* phase eliminates all guessed candidates that turn out not to be solutions. Since many complex problems have a combinatorial structure, this method allows for a succinct description of the problem to be solved.

Second, handling computationally complex queries over huge data is an insurmountable obstacle for standard algorithms. One potential solution is to *exploit structure*. This is motivated by the facts that, for example, molecules are not random graphs and medical ontologies are not arbitrary sets of relations.

A prominent approach to exploit structure is to employ tree decompositions (see, e.g., [21] for an overview). This is particularly attractive because it allows us to successfully decompose any problem for which a graph representation can be found. Even more important, via Courcelle's famous result [27] it is known that many problems can be efficiently solved with dynamic programming (DP) algorithms on tree decompositions if the structural parameter "treewidth" is bounded, which means that the graph resembles a tree to a certain extent. The main feature of such an approach is that what causes an explosion of a traditional algorithm's running time can be confined to only this structural parameter instead of mere input size. Consequently, if the treewidth is bounded, even huge instances of many problems can be solved without falling prey to the exponential explosion. Empirical studies [6, 43, 44, 45, 51, 56, 57] indicate that in many practical applications the treewidth is usually indeed small. However, the implementation of suitable efficient algorithms is often done from scratch, if done at all.

All this calls for a suitable combination of declarative approaches on the one hand and structural methods on the other hand.

We focus here on *a combination of ASP and problem solving via DP on tree decompositions*. For this, we have implemented a free software system called D-FLAT[1] for rapid prototyping of DP algorithms in the declarative language of ASP. The success of ASP for solving hard problems

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

witnesses that this language is well suited for a lot of problems, and in fact it turns out that ASP is often also well suited for parts of such problems. This makes it an appealing candidate for working on decomposed problem instances.

The key features of D-FLAT are that

- ASP is used to specify the DP algorithm by declarative means (since ASP originated in part from research on databases, it can be conveniently used to specify table transitions which are the typical operations in DP);
- the burden of computation and optimization is delegated to existing tools for finding tree decompositions and to ASP solvers;
- D-FLAT relieves the user from tedious non-problem-specific tasks, but stays flexible enough to offer enough power to solve a great number of problems;
- in particular, D-FLAT can be applied to any problem whose fixed-parameter tractability follows from Courcelle's Theorem [27] in the sense that, given a suitable specification of the DP algorithm of the considered problem together with a tree decomposition of the instance, D-FLAT is able the solve the problem in fixed-parameter linear time (see [17]).

Comparing the standard way of Answer Set Programming with the dynamic programming approach via D-FLAT, the following differences and implications have to be emphasized:

- In standard ASP, an encoding describes how to solve the problem at hand when the instance is given in its entirety; we call such encodings also *monolithic*. An ASP solver is only invoked once and delivers the solutions.
- In D-FLAT, the problem instance is first decomposed and the encoding has to follow a certain dynamic programming style. In fact, the encoding is called many times on different components of the instance and the ASP solver delivers solutions to subproblems, which are then, roughly speaking, put together accordingly by D-FLAT.
- Finding such a dynamic programming encoding that works on tree decompositions might be considerably more involved than finding a monolithic encoding. However, the D-FLAT approach might help to overcome the well-known grounding bottleneck of ASP (since the encoding is only applied to small fractions of the instance) and allows for a straightforward parallelization since certain components can be solved in parallel (see [14]). Finally, for instances of small treewidth, D-FLAT can potentially outperform the standard ASP approach.

D-FLAT is free software written in C++ and internally uses the answer set solving systems *Gringo* and *Clasp* [38], as well as the *htd* framework [3, 4] for heuristically generating a tree decomposition of the input.

Since D-FLAT version 1.0.0, which has been described in a previous report [1], we have improved the system in terms of efficiency and additional features. On the one hand, D-FLAT now uses the decomposition framework *htd*. On the other hand, we added new functionality to D-FLAT

such as a lazy evaluation mode which is especially useful for search problems and for optimization problems, where it allows for anytime optimization [14]. Moreover, we added built-in counters, which take over even more of the computational burden from ASP [2]. We used the tool for implementing decomposition-based algorithms for various problems from diverse application areas, which demonstrates the usability of the method.

This report is structured as follows: We first provide background on Answer Set Programming and tree decompositions in Section 2. In Section 3, we then present the current version 1.2.5 of D-FLAT and describe its components, as well as the newer functionality, in detail. Subsequently, we turn to practical applications in Section 4, where we present D-FLAT encodings for several problems and illustrate how to use D-FLAT in practice. We conclude this work with a summary in Section 5, where we also mention some related work.

This report extends and reuses several parts from the first progress report from 2014 [1]. The main changes appear in Sections 3.2 and 3.7, where we discuss the decomposition library and the new functionality. Accordingly, we adapted Tables 2–5, which list the internal predicates for D-FLAT encodings, as well as Section 3.8, which describes the command-line interface of D-FLAT. We substantially reorganized Section 4, where we illustrate the usage of D-FLAT on some examples. That section now puts a focus on examples that make use of the new functionalities (Sections 4.4 and 4.5).

2 Background

In this section we describe the underlying concepts of the D-FLAT system. We largely follow the presentation in [12]. Section 2.1 is devoted to Answer Set Programming and Section 2.2 covers tree decompositions.

2.1 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. Logic programming under the answer set semantics is a formalism that allows us to succinctly specify programs that follow such a *Guess & Check* approach [10, 39, 46]. *Answer Set Programming* (ASP) denotes a fact-driven programming 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 [23, 47]. In [53, 50], ASP is proposed as a paradigm for declarative problem solving. A crucial observation is that the answer set semantics allows a logic program to have multiple models, which allows for modeling non-deterministic computations in a natural way. Since answer sets derive from nonmonotonic reasoning, also the concept of negation as failure is implied. [26]

2.1.1 Syntax

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

Definition 1. *Each variable and each constant is a* term*. Also, if f is a function symbol with arity n, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. A term is* ground *if it contains no variables. If p is an m-ary predicate symbol and $t_1, \ldots, t_m$ are terms, then we call $p(t_1, \ldots, t_m)$ an* atom*. A literal is either just an atom or an atom with the symbol "not" put in front of it. An atom or literal is called ground if only ground terms occur in it.*

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

Definition 2. *A* logic program *(sometimes just called "program" for short) is a set of rules of the form*

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

where a and $b_1, \ldots, b_n$ are atoms. Let r be a rule of a program Π. We call $h(r) = a$ the head *of r, and $b(r) = \{b_1, \ldots, b_n\}$ its* body *which is further divided into a* positive body*, $b^+(r) = \{b_1, \ldots, b_m\}$, and a* negative body*, $b^-(r) = \{b_{m+1}, \ldots, b_n\}$.*

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.

2.1.2 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 3. *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 $gr_\Pi(r)$, 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*

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

We now define the answer set semantics that have first been proposed in [40]. To this end, we first introduce the notion of answer sets for ground programs.

Definition 4. *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*

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

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 5. *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. D-FLAT uses the grounder *Gringo* and the solver *Clasp* [38].

2.1.3 Complexity and Expressive Power

Naturally, questions of computational complexity and expressive power of ASP arise. A survey of results is given in [28]. We will now mention results that are especially important for our purposes.

The unrestricted use of function symbols leads to undecidability in general [24, 7]. In this work, we therefore do not allow function symbols of positive arity to be nested, which is a very restrictive condition but already gives us a convenient modeling language. In this case, deciding whether a ground logic program has an answer set is NP-complete [49].

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 situations where the program stays the same for variable data (cf. Section 2.1.4 for an example encoding for the GRAPH COLORING problem), 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.

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. It has been shown that ASP captures NP [55], and also every search problem in NP can be expressed by a uniform ASP program [48]. 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 [41]) yields higher expressiveness at the cost of Σ_2^P-completeness for the problem of deciding answer set existence for ground programs [32, 33].

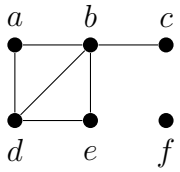


Figure 1 A 3-colorable graph

```
color(red). color(grn). color(blu).
vertex(a). vertex(b). vertex(c). vertex(d). vertex(e). vertex(f).
edge(a,b). edge(a,d). edge(b,c). edge(b,d). edge(b,e). edge(d,e).
```

Listing 1 Declaration of a GRAPH COLORING instance containing the graph from Figure 1

2.1.4 ASP in Practice

Various systems are available [38, 46] 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 [37, 36] in the program examples and use a `monospaced` font for typesetting rules in that language. The $\leftarrow$ symbol corresponds to `:-` and each rule is terminated by a period. In our listings, we will perform "beautifications" such as using $\leftarrow$ instead of `:-` and $\neq$ instead of `!=` for the sake of better readability.

As an introductory example, we will show how ASP can be applied to solve the GRAPH COLORING problem. Figure 1 depicts the graph in a possible instance of the GRAPH COLORING problem where the vertices shall be colored either in "red", "grn" or "blu" such that adjacent vertices never have the same color. This instance can be represented as a set of facts in the input language of Gringo as seen in Listing 1. The following program solves the GRAPH COLORING problem for instances specified in this way.

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

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

The first 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 if we assume the colors to be "red", "grn" and "blu":

```
1 { map(X,red), map(X,grn), map(X,blu) } 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 if 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,grn)` and `map(X,blu)` for any vertex `X`.

The integrity constraint in the second line ensures that no answer set maps the same color to adjacent vertices. This rule uses pooling (indicated by ";") and is expanded by the grounder to the equivalent rule:

```
← edge(X,Y), map(X,C), map(Y,C).
```

As another example, consider the DOMINATING SET problem, in which, given a graph (V, E) we are looking for sets S such that each $v \in V$ is either contained in S or adjacent to at least one vertex from S. Vertices from the latter group are called dominated, while the sets S are the dominating sets. The given input instance must now provide facts using only the predicates `vertex`/1 and `edge`/2 which describe the input graph. The following ASP program then solves the DOMINATING SET problem.

```
{ selected(X) : vertex(X) }.
dominated(Y) ← selected(X), edge(X,Y).
← vertex(X), not selected(X), not dominated(X).
```

In the first rule we guess for each vertex whether it should belong to the candidate dominating set or not. Next, we derive `dominated`/1 for each vertex that is adjacent to a vertex in the candidate dominating set. Finally, we check for each vertex whether it is dominated or in the candidate set. If neither is the case, we have to discard the solution candidate (which is done via the third rule). Hence, we obtain only answer sets which denote an actual solution to our problem.

2.2 Tree Decompositions

Many computationally hard problems on graphs are easy if the instance is a tree. It would of course be desirable if we could also efficiently solve instances that are "almost" trees. Fortunately, it is indeed possible to exploit "tree-likeness" in many cases. Tree decompositions and the associated concept of treewidth provide us with powerful tools for achieving 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 [54]. Since then, it has been widely acknowledged that treewidth represents a very useful parameter that is applicable to a broad range of problems. There are several overviews of this topic, such as [20, 18, 9, 52].

2.2.1 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. (But note that in Section 3 we will need to introduce yet another kind of node.)

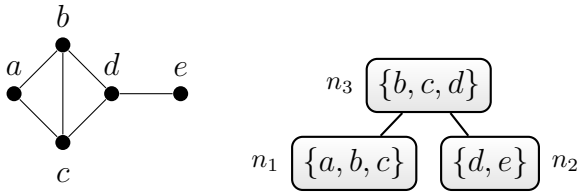


Figure 2 A graph with treewidth 2 and an (optimal) tree decomposition for it

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 6. *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 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., [56, 6, 43, 44, 45, 51]). We provide a collection of real world traffic network instances in ASP format, based on which the reader can deduce that indeed often the treewidth is quite small.[2] This appears to be very promising, since it indicates that the D-FLAT 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 2.2.2.

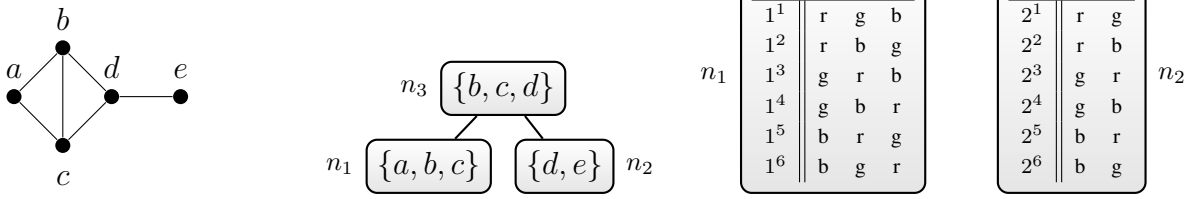
In general, determining a graph's treewidth and constructing an optimal tree decomposition are unfortunately intractable: Given a graph and a non-negative integer k, deciding whether the graph's treewidth is at most k is NP-complete [8]. However, the problem is fixed-parameter tractable w.r.t. the parameter k, i.e., if we are given a fixed k in advance, the problem becomes tractable: 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 [19]. 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 [22, 31, 42]. 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.

2.2.2 Dynamic Programming on Tree Decompositions

Figure 3 shows how dynamic programming can be applied to a tree decomposition of a GRAPH COLORING instance. Each of the tree decomposition nodes in Figure 3b has a corresponding table in Figure 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. This is done by means of the so-called extension pointer tuples (EPTs) stored in j. Eventually, each row will describe a proper coloring of the subproblem represented by the bag.

[2] See `https://github.com/daajoe/transit_graphs` for example instances and `https://github.com/daajoe/gtfs2graphs` for a downloader for instances which are in GTFS format



n_3

i	b	c	d	j
3^1	r	g	b	$(1^5, 2^5), (1^5, 2^6)$
3^2	r	b	g	$(1^3, 2^3), (1^3, 2^4)$
3^3	g	r	b	$(1^6, 2^5), (1^6, 2^6)$
3^4	g	b	r	$(1^1, 2^1), (1^1, 2^2)$
3^5	b	r	g	$(1^4, 2^3), (1^4, 2^4)$
3^6	b	g	r	$(1^2, 2^1), (1^2, 2^2)$

n_1

i	a	b	c
1^1	r	g	b
1^2	r	b	g
1^3	g	r	b
1^4	g	b	r
1^5	b	r	g
1^6	b	g	r

n_2

i	d	e
2^1	r	g
2^2	r	b
2^3	g	r
2^4	g	b
2^5	b	r
2^6	b	g

(a) A GRAPH COLORING instance

(b) A tree decomposition of the instance

(c) The computed dynamic programming tables

Figure 3 Dynamic programming for GRAPH COLORING on a tree decomposition

Adhering to the approach of dynamic programming, the tables in Figure 3c are computed in a bottom-up way. First all proper 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 is joined. 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 1^1 with row 2^3 is invalid because the adjacent vertices b and d are colored with "g"; row 1^1 combined with row 2^1 is valid, however, and gives rise to row 3^4 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 3^4 not only contains the EPT $(1^1, 2^1)$ but also $(1^1, 2^2)$ 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* proper 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 GRAPH COLORING, 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.

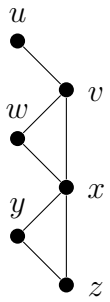
Suppose the number of colors is fixed. Even then, this algorithm's space and time requirements are both exponential in the decomposition width. However, when the treewidth can be considered

bounded, this algorithm runs in linear space and time. This proves fixed-parameter tractability of GRAPH COLORING parameterized by the treewidth and the number of colors. It is a general property of the algorithms presented in this work that the width of the obtained decompositions is crucial for the performance.

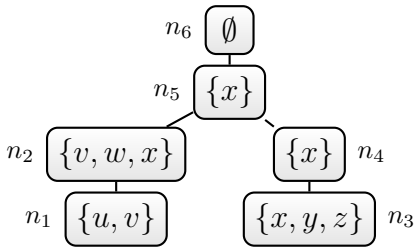
Our next example shows that optimization variants of certain problems are easily obtained via dynamic programming. To this end, let us consider the enumeration variant of the MINIMUM DOMINATING SET (MDS) problem on a graph $G = (V, E)$. This means that we want to determine all dominating sets S of minimal cardinality. An example graph and a possible tree decomposition are given in Figures 4a and 4b. The width of the tree decomposition is 2. Note that it contains an empty root and further, unnecessarily many nodes: We could obtain another valid tree decomposition for the graph by arranging n_3, n_2 and n_1 in a path. However, we chose this one to serve for our example because it is more suitable for illustrating DP algorithms. Figure 4 shows how dynamic programming can be applied to a tree decomposition of a MINIMUM DOMINATING SET instance. Figure 4c illustrates the DP computation for MINIMUM DOMINATING SET. The tables are computed as follows.

For a TD node n, each table row i contains information as to which of the vertices belonging to node n is selected into the dominating set. The second column contains the vertices selected into S, and the dominated ones, which are highlighted via a bar above the vertex name. Those which are neither selected nor dominated must be dominated during further steps of the tree traversal. Column j again contains the EPTs that denote the rows in the children where i was constructed from. The value in column C denotes the cost (number of selected vertices) of the cheapest solution which is consistent with the selection into S. Partial solutions with higher costs are not propagated. First consider node n_1: Here, $\chi(n_1) = \{u, v\}$ allows for four solution candidates. In n_2, the child rows are extended, the partial assignments are updated (by removing vertices not contained in $\chi(n_2)$ and guessing which of the vertices in $\chi(n_2) \setminus \chi(n_1)$ are to be selected and which become dominated). Observe that row 4^2 is constructed from two different child rows. In n_3 we proceed as described before. In n_4, data related to removed vertices y and z are projected away. In n_5, additionally only partial solutions that select the same subset of common vertices are to be joined. We continue this procedure recursively until we reach the TD's root. Note that the root node readily gives the minimum cardinality over all dominating sets.

The overall procedure is in FPT time because the number of nodes in the TD is bounded by the size of the input graph and each node n is associated with a table of size at most $\mathcal{O}(2^{|\chi(n)|})$ (i.e., the number of possible selections). The actual solutions (minimum dominating sets of the input instance) can be enumerated with linear delay by starting at the root and following the EPTs while combining the partial assignments associated with the rows. For instance, the minimum dominating set $\{v, x\}$ is constructed by starting at 6^1 and following EPTs (5^1), $(2^3, 4^1)$, (1^3) and (3^5), thereby combining $S(6^1) \cup S(5^1) \cup S(2^3) \cup S(4^1) \cup S(1^3) \cup S(3^5)$.



(a) A MINIMUM DOMINATING SET instance



(b) A tree decomposition of the instance

n_6

i	S	j	C
6^1		$(5^1), (5^2)$	2

n_5

i	S	j	C
5^1	x	$(2^3, 4^1), (2^7, 4^1)$	2
5^2	$\overline{x}$	$(2^5, 4^2)$	2

n_2

i	S	j	C
2^1	v, w, x	(1^3)	3
2^2	$v, w, \overline{x}$	(1^3)	2
2^3	$v, x, \overline{w}$	(1^3)	2
2^4	$w, x, \overline{v}$	(1^2)	3
2^5	$v, \overline{w}, \overline{x}$	(1^3)	1
2^6	$w, \overline{v}, \overline{x}$	(1^2)	2
2^7	$x, \overline{v}, \overline{w}$	(1^2)	2

n_1

i	S	C
1^1	u, v	2
1^2	$u, \overline{v}$	1
1^3	$v, \overline{u}$	1
1^4		0

n_4

i	S	j	C
4^1	x	(3^5)	1
4^2	$\overline{x}$	$(3^6), (3^7)$	1

n_3

i	S	C
3^1	x, y, z	3
3^2	$x, y, \overline{z}$	2
3^3	$x, z, \overline{y}$	2
3^4	$y, z, \overline{x}$	2
3^5	$x, \overline{y}, \overline{z}$	1
3^6	$y, \overline{x}, \overline{z}$	1
3^7	$z, \overline{x}, \overline{y}$	1
3^8		0

(c) The computed DP tables

Figure 4 Dynamic programming for MINIMUM DOMINATING SET on a tree decomposition

3 The D-FLAT System

This section first gives an overview of the D-FLAT system and then describes its components and functionality in more detail. The system is free software and can be downloaded at `http://dbai.tuwien.ac.at/research/project/dflat/system/`.

3.1 System Overview

D-FLAT[3] is a framework for developing algorithms that solve computational problems by dynamic programming on a tree decomposition of the problem instance. Such an algorithm typically encompasses the following steps.

1. It constructs a tree decomposition of the problem instance, thereby decomposing the instance into several smaller parts.

2. It solves the sub-problems corresponding to these parts individually and stores partial solutions in an appropriate data structure.

3. It combines the partial solutions following the principle of dynamic programming and prints all thus obtained complete solutions.[4]

Among these tasks, the one that is really problem-specific is the second one – solving the sub-problems. When faced with a particular problem, algorithm designers typically focus on this step. The others – constructing a tree decomposition and combining partial solutions – are often perceived as a distracting and tedious burden. This is why D-FLAT takes care of the first and third step in a generic way and lets the programmer focus solely on the problem at hand.

Furthermore, it is often much more convenient to solve problems using a declarative language when compared with an imperative implementation. Especially in the phase where the algorithm designer wants to explore an idea for an algorithm, it is of great help to be able to quickly come up with a prototype implementation that can easily be adapted if it turns out that some details have been missed. Therefore, D-FLAT offers the possibility of using the declarative language of Answer Set Programming (ASP) to specify what needs to be done for solving the sub-problem corresponding to a node in the tree decomposition of the input.

To summarize, D-FLAT allows problems to be solved in the following way:

1. D-FLAT takes care of parsing a representation of the problem instance and automatically constructing a tree decomposition of it using heuristic methods.

2. The framework provides a data structure (called *item tree*) that is suitable for representing partial solutions for many problems. The only thing that the programmer needs to provide is

[3]The acronym stands for *Dynamic Programming Framework with Local Execution of ASP on Tree Decompositions*.

[4]Note that, depending on the problem, printing all solutions may not be required. Often we just want, e.g., to decide whether a solution exists, to count the number of solutions, or to find an optimal solution. D-FLAT also offers facilities for such cases.

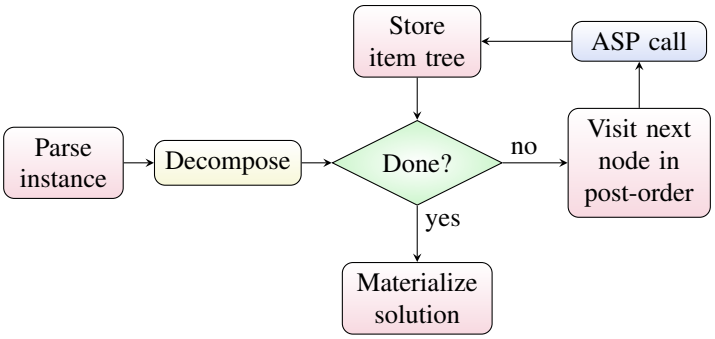


Figure 5 Control flow in D-FLAT

an ASP specification of how to compute the item tree associated with a tree decomposition node.

3. D-FLAT automatically combines the partial solutions and prints all complete solutions. Alternatively, it is also possible to solve decision, counting and optimization problems.

Regarding the applicability of D-FLAT, we have shown in [16, 17] that any problem expressible in monadic second-order logic can also be solved with D-FLAT in FPT time (i.e., in time $f(w) \cdot n^{\mathcal{O}(1)}$, where n is the size of the input, w is its treewidth and $f(w)$ depends only on w). This includes many problems from NP but also harder problems in PSPACE.

Figure 5 depicts the control flow during the execution of an algorithm with D-FLAT and illustrates the interplay of the system's components.

3.2 Constructing a Tree Decomposition

D-FLAT expects the input that represents a problem instance to be specified as a set of facts in the ASP language. For constructing a tree decomposition, D-FLAT first needs to build a hypergraph representation of this input. Along with the facts describing the instance, the user therefore must specify which predicates therein designate the hyperedge relation.[5]

Example 1. *Suppose we want to solve the* GRAPH COLORING *problem where an instance consists of a graph G together with a set of colors C. We want to find all proper colorings of G using colors from C.*

Let an instance be given by the graph depicted in Figure 1 and the colors "red", "grn" and "blu". This can be specified in ASP using the facts from Listing 1 in Section 2.1.4. Given the information that the predicates `vertex` *and* `edge` *shall denote hyperedges, D-FLAT builds a hypergraph representation that has the same vertices as the graph, and edges in the graph correspond to binary hyperedges. There is also a unary hyperedge relation induced by the predicate* `vertex`*, which is only used to make all vertices of the hypergraph known to D-FLAT.*

[5] Vertices not incident to any hyperedge can be included in the domain by adding unary hyperedges.

Once a hypergraph representation of the input has been built, the framework uses an external framework for heuristically constructing a tree decomposition of small width.[6] This framework relies on a bucket elimination algorithm [29] that requires an *elimination order* of the vertices.

Given a hypergraph H and an elimination order σ, a tree decomposition T can be constructed as follows. For each $v \in \sigma$: Make v simplicial (i.e., connect all its neighbors s.t. they form a clique) and remove v from H. Consequently, a new tree decomposition node, whose bag contains v and all its neighbors, is added to T. Connectedness is ensured by adding an edge to each already existing node in T in whose bag v appears as well.

The following heuristics for finding elimination orders are currently supported:

Min-degree Initially, the vertex with minimum degree is selected as the first one in the order. The heuristic then always selects the next vertex having the least number of not yet selected neighbors and repeats this step until all vertices are eliminated.

Min-fill Always select the vertex whose elimination adds the smallest number of edges to H until all vertices are eliminated.

In each heuristic, ties are broken randomly.

It is often convenient to presuppose tree decompositions having a certain normal form. This usually makes algorithms easier to specify as fewer cases have to be considered. On the other hand, the size of the tree decomposition thereby increases in general, but only linearly. The following optional *normalizations* of tree decompositions are offered:

Weak normalization In a *weakly normalized* tree decomposition, each node with more than one child is called a *join node* and must have the same bag elements as its children. We call unary nodes (i.e., nodes with one child) *exchange nodes*.

Semi-normalization A *semi-normalized* tree decomposition is weakly normalized – additionally, join nodes must have exactly two children.

Normalization A *normalized* (sometimes also called *nice*) tree decomposition is semi-normalized – additionally, each exchange node must be of one of two types: Either it is a *remove node* whose bag consists of all but one vertices from the child's bag; Or it is an *introduce node* whose bag consists of all vertices from the child bag plus another vertex.

D-FLAT additionally allows the user to choose whether the generated decomposition shall have leaves with empty bags, and whether the root shall have an empty bag.

Further, since it uses the *htd* framework, D-FLAT offers the possibility to create several tree decompositions and select the fittest one. The user can set the number of iterations for creating tree decompositions, out of which the fittest one will be selected according to the fitness criterion chosen by the user. The latter can be the minimal value of the maximum bag size, of the average join node bag size, of the median join node bag size or the minimal value of the number of join

[6] Starting with version 1.2.0 D-FLAT uses the *htd* framework [3, 4] for computing and customizing tree decompositions.

nodes. The impact of selecting an appropriate decomposition for the running time of DP algorithms has been thoroughly studied [5] and can be quite substantial. Thus, the incorporation of the *htd* framework allows for tuning the perfomance of D-FLAT also via the generation of suitable decompositions. However, the general methodology of D-FLAT works for any system that delivers tree decompositions (for further such systems, see e.g. [30]).

3.3 Item Trees

D-FLAT equips each tree decomposition node with an *item tree*. An item tree is a data structure that shall contain information about (candidates for) partial solutions. At each decomposition node during D-FLAT's bottom-up traversal of the tree decomposition, this is the data structure in which the problem-specific algorithm can store data.

Most importantly, each node in an item tree contains an *item set*. The elements of this set, called *items*, are arbitrary ground ASP terms. Beside the item set, an item tree node contains additional information about the item set as well as data required for putting together complete solutions, which will be described later in this section.

Item trees are similar to computation trees of Alternating Turing Machines (ATMs) [25]. Like in ATMs, a branch can be seen as a computation sequence, and branching amounts to non-deterministic guesses. We will repeatedly come back to the ATM analogy in the course of this section.

Usually we want to restrict the information within an item tree to information about the current decomposition node's bag elements. More precisely, we want to make sure that the maximum size of an item tree only depends on the bag size. The reason is that when this condition is satisfied and the decomposition width is bounded by a constant, the size of each item tree is also bounded. This allows us to achieve FPT algorithms.

Example 2. *Consider again the* GRAPH COLORING *instance from Example 1. Figure 6 shows a tree decomposition for the input graph (Figure 1) and, for each decomposition node, the corresponding item tree that could result from an algorithm for* GRAPH COLORING*. For solving this problem, we use item trees having a height of at most 1. Each item tree node at depth 1 encodes a coloring of the vertices in the respective bag. The meaning of the symbols* $\vee$*,* $\top$ *and* $\bot$ *will be explained in Section 3.3.2.*

3.3.1 Extension Pointers

In order to solve a complete problem instance, it is usually necessary to combine information from different item trees. For example, in order to find out if a proper coloring of a graph exists, we do not only have to check if a proper coloring of each subgraph induced by a bag exists but also if, for each bag, we can pick a local coloring in such a way that each vertex is never colored differently by two chosen local colorings.

For this reason each item tree node has a (non-empty) set of *extension pointer tuples*. The elements of such a tuple are called *extension pointers* and reference item tree nodes from children of the respective decomposition node. Roughly, an extension pointer specifies that the information

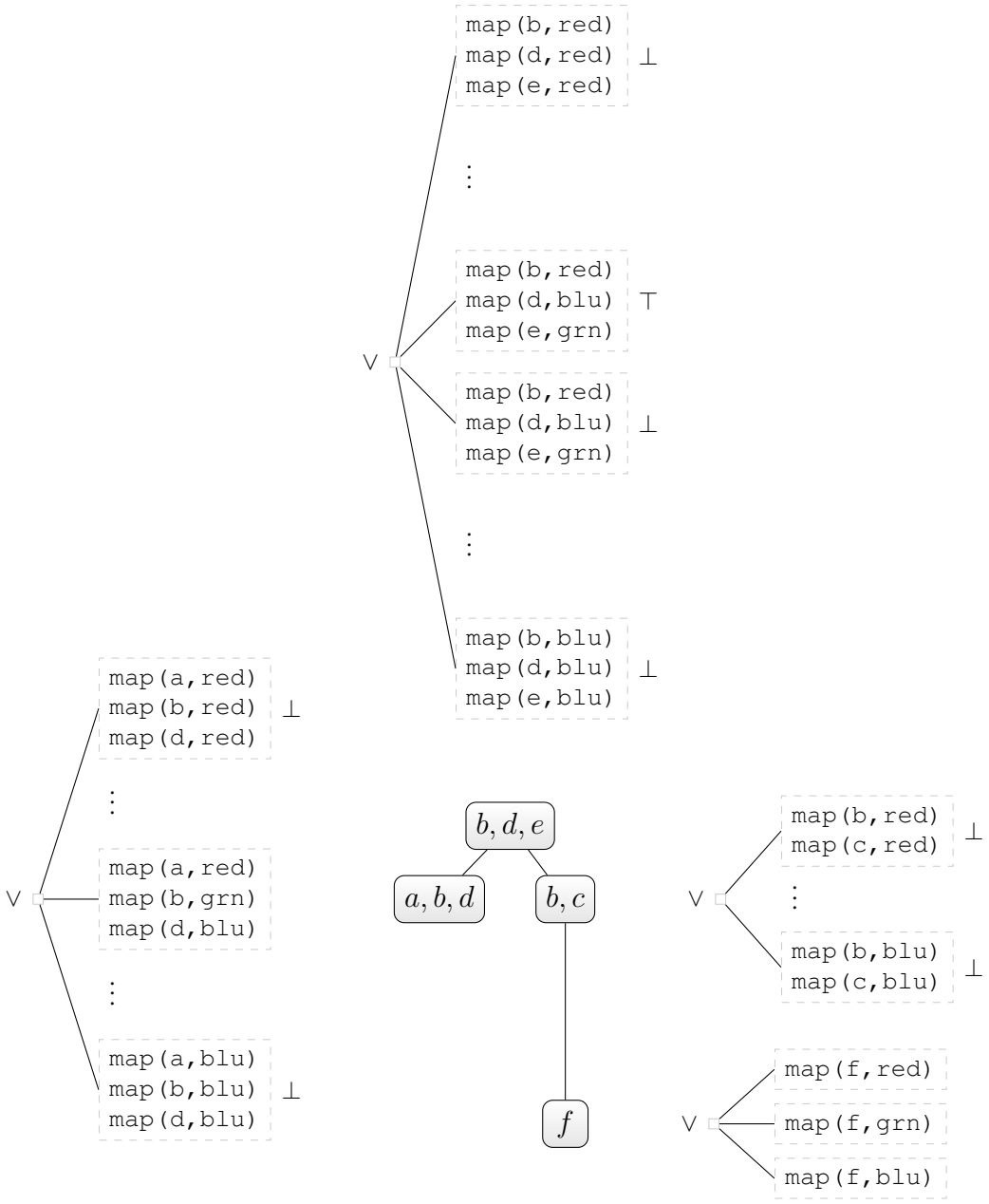


Figure 6 A tree decomposition with item trees for GRAPH COLORING (not showing extension pointers)

in the source and target nodes can reasonably be combined. We define these notions as follows. Let δ be a tree decomposition node with children $\delta_1, \ldots, \delta_n$ (possibly $n = 0$), and let $\mathcal{I}$ and $\mathcal{I}_1, \ldots, \mathcal{I}_n$ denote the item trees associated with δ and $\delta_1, \ldots, \delta_n$, respectively. Each extension pointer tuple in any node ν of $\mathcal{I}$ has arity n. Let $(e_1, \ldots, e_n)$ be an extension pointer tuple at a node at depth d of $\mathcal{I}$. For any $1 \leq i \leq n$, it holds that e_i is a reference to a node ν_i at depth d in $\mathcal{I}_i$. We then say

that ν *extends* ν_i.

Example 3. *Consider Figure 6 again. In this example, we use the notation* δ_S *to denote the decomposition node whose bag is the set* S*, and we write* $\mathcal{I}_S$ *to denote the item tree of that node.*

Although the figure does not depict extension pointers, we will explain how they would look like in this example. In $\mathcal{I}_{\{a,b,d\}}$ *and* $\mathcal{I}_{\{f\}}$*, all nodes have the same set of extension pointer tuples: the set consisting of the empty tuple, as these decomposition nodes have no children.*

In $\mathcal{I}_{\{b,c\}}$*, the situation is more interesting: The root has a single unary extension pointer tuple whose element references the root of* $\mathcal{I}_{\{f\}}$*. Each node at depth 1 of* $\mathcal{I}_{\{b,c\}}$ *has three unary extension pointer tuples – one for each node at depth 1 of* $\mathcal{I}_{\{f\}}$*.*

The set of extension pointer tuples at the root of $\mathcal{I}_{\{b,d,e\}}$ *consists of a single binary tuple – one element references the root of* $\mathcal{I}_{\{a,b,d\}}$*, the other references the root of* $\mathcal{I}_{\{b,c\}}$*. For a node* ν *at depth 1 of* $\mathcal{I}_{\{b,d,e\}}$*, the set of extension pointer tuples consists of all tuples* (ν_1, ν_2) *such that* ν_1 *and* ν_2 *are nodes at depth 1 of* $\mathcal{I}_{\{a,b,d\}}$ *and* $\mathcal{I}_{\{b,c\}}$*, respectively.*

3.3.2 Item Tree Node Types

Like states of ATMs, item tree nodes in D-FLAT can have one of the types "or", "and", "accept" or "reject". Unlike ATMs, however, the mapping in D-FLAT is partial. The problem-specific algorithm determines which item tree node is mapped to which type. The following conditions must be fulfilled.

- If a non-leaf node of an item tree has been mapped to a type, it is either "or" or "and".
- If a leaf node of an item tree has been mapped to a type, it is either "accept" or "reject".
- If an item tree node extends a node with defined type, it must be mapped to the same type.

When D-FLAT has finished processing all answer sets and has constructed the item tree for the current tree decomposition node, it propagates information about the acceptance status of nodes upward in this item tree. This depends on the node types defined in this section, and is described in Section 3.5.2. The node types also play a role when solving optimization problems – roughly, when something is an "or" node, we would like find a child with minimum cost, and if something is an "and" node, we would like to find a child with maximum cost. This is described in Section 3.5.3.

Example 4. *The item trees in Figure 6 all have roots of type "or", denoted by the symbol* $\vee$*. This is because an ATM for deciding graph colorability starts in an "or" state, then guesses a coloring and accepts if this coloring is proper. Therefore, we shall derive the type "reject" in our decomposition-based algorithm whenever we determine that a guessed coloring is not proper, and we derive "accept" once we are sure that a coloring is proper.*[7]

In $\mathcal{I}_{\{a,b,d\}}$ *and* $\mathcal{I}_{\{f\}}$*, for instance, we have marked all leaves representing an improper coloring with* $\bot$*. The types of the other leaves are left undefined, as guesses on vertices that only appear*

[7] It should be noted that the algorithm could be optimized by not even creating nodes encoding improper colorings. In order to remain faithful to the ATM analogy for the sake of presentation, we follow the habit of creating a branch for each non-deterministic guess, even if this choice leads to a rejecting state.

later could still lead to an improper coloring. At the root of the tree decomposition however, we mark all item tree leaves having a yet undefined type with $\top$ because all vertices have been encountered.

Note that it may happen that sibling nodes have equal item sets (like in $\mathcal{I}_{\{b,d,e\}}$). This is because nodes with equal item sets but different types or counter values (cf. Section 3.7.2) are considered unequal. Consider the two middle leaves in $\mathcal{I}_{\{b,d,e\}}$, for instance: The reason for one being marked with $\top$ is that it extends only nodes whose type has still been undefined, whereas the leaf marked with $\bot$ extends at least one "reject" node.

3.3.3 Solution Costs for Optimization Problems

When solving an optimization problem with an ATM, we assume that with each accepting run we can associate some kind of cost that depends on which non-deterministic choices have been made. Furthermore, we assume that the result of the ATM computation is now no longer "yes" or "no", depending on whether the root of the computation tree is accepting or rejecting, but rather a number that shall represent the optimum cost, for a certain notion of optimality that we will now define.

For a non-deterministic Turing machine without alternation, the straightforward result of a computation is the minimum cost among all runs. When alternation is involved, we can easily generalize this in the following way. Suppose each leaf of the computation tree is annotated with a cost. (Rejecting nodes have cost ∞.) The *optimization value* of a node ν can now be defined as (a) its cost if ν is a leaf, (b) the minimum cost among all children in case ν is an "or" node, and (c) the maximum cost among all children in case ν is an "and" node. The result of an ATM computation for an optimization problem is then the optimization value of the root of its computation tree.

In analogy to this procedure of solving optimization problems, D-FLAT allows leaves of item trees to contain a number that specifies the cost that the respective branch has accumulated so far. That is, the cost that is stored in the leaf of an item tree refers not only to the non-deterministic choices based on the current bag elements, but also on past choices (obtainable by following the extension pointers) from item trees further down in the decomposition.

3.4 D-FLAT's Interface for ASP

D-FLAT invokes an ASP solver at each node during a bottom-up traversal of the tree decomposition. The user-defined, problem-specific encoding is augmented with input facts describing the current bag as well as the bags and item trees of child nodes. Additionally, the original problem instance is supplied as input.[8] The answer sets of this ASP call specify the item tree that D-FLAT shall construct for the current decomposition node.

In Section 3.4.1 we describe the interface to the user's ASP encoding, i.e., the input and output predicates that are used for communicating with D-FLAT. Note that there is also a simplified

[8]To be precise, D-FLAT provides the ASP system not with the whole problem instance but only with the part that is induced by the current bag elements. This allows for fixed-parameter linear running time for decision and counting problems. (Otherwise the same algorithms would be fixed-parameter quadratic.)

version of the ASP interface, which we describe in Section 3.4.2, for dealing with problems in NP.

In all D-FLAT listings presented in this document, we use colors to highlight input and output predicates.

3.4.1 General ASP Interface

D-FLAT provides facts about the tree decomposition as described in Table 1. Additionally, it defines the integer constant `numChildNodes` to be the number of children of the current tree decomposition node. The item trees of these nodes are declared using predicates described in Table 2.

The answer sets of the problem-specific encoding together with this input give rise to the item tree of the current tree decomposition node. Each answer set corresponds to a branch in the new item tree. The predicates for specifying this branch are described in Table 3. We use the term "current branch" in the table to denote the branch that the respective answer set corresponds to; the "child item tree" shall denote an item tree that belongs to a child of the current tree decomposition node. One should keep in mind, however, that D-FLAT may merge subtrees as described in Section 3.5. Therefore, after merging, one branch in the item tree may comprise information from multiple answer sets.

Example 5. *A possible encoding for the* GRAPH COLORING *problem is shown in Listing 2. Note that it makes more sense to encode this problem using the simplified ASP interface for problems in* NP*, which we describe in Section 3.4.2.*

The first line in the listing is a modeline, which is explained in Section 3.8. Line 2 specifies that each answer set declares a branch of length 1, whose root node has the type "or". Line 3 guesses a color for each current vertex. The "reject" node type is derived in line 4 if this guessed coloring is improper. Lines 5 and 6 guess a branch for each child item tree. Due to line 7, the guessed combination of predecessor branches only leads to an answer set if it does not contradict the coloring guessed in line 3. This makes sure that only branches are joined that agree on all common vertices, as each vertex occurring in two child nodes must also appear in the current node due to the connectedness condition of tree decompositions. If a guessed predecessor branch has led to a conflict (denoted by a "reject" node type), this information is retained in line 8.[9] *Finally, line 9 derives the "accept" node type if no conflict has occurred. The last two lines instruct the ASP solver to only report facts with the listed output predicates, and we will justify their use later in this section.*

Note that some of the rules in Listing 2 can be used (with small modifications) for any problem that is to be solved with D-FLAT. In particular, lines 5 and 6 are applicable to all problems after adapting them to the item tree depth of the problem. Usually rules similar to line 3 are used for performing a guess on bag elements (which of course depends on the problem), and checks are done with rules similar to lines 4 and 7.

[9] Unless the user disables D-FLAT's pruning of rejecting subtrees (cf. Section 3.5.2), this rule in fact never fires, but we list it anyway for a clearer presentation and to be consistent with the item trees in Figure 6.

Input predicate	Meaning
`initial`	The current tree decomposition node is a leaf.
`final`	The current tree decomposition node is the root.
`currentNode`(N)	N is the identifier of the current decomposition node.
`childNode`(N)	N is a child of the current decomposition node.
`bag`(N, V)	V is contained in the bag of the decomposition 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.

Table 1 Input predicates describing the tree decomposition

```
1  %dflat: -e vertex -e edge --no-empty-leaves --no-empty-root
2  length(1). or(0).
3  1 { item(1,map(X,C)) : color(C) } 1 ← current(X).
4  reject ← edge(X,Y), item(1,map(X,C;Y,C)).
5  extend(0,S) ← rootOf(S,_).
6  1 { extend(1,S) : sub(R,S) } 1 ← rootOf(R,_).
7  ← item(1,map(X,C0)), childItem(S,map(X,C1)), extend(_,S), C0 ≠ C1.
8  reject ← childReject(S), extend(_,S).
9  accept ← final, not reject.
10 #show item/2. #show extend/2. #show length/1.
11 #show or/1.   #show accept/0. #show reject/0.
```

Listing 2 D-FLAT encoding for GRAPH COLORING using the general ASP interface

Input predicate	Meaning
atLevel(S, L)	S is a node at depth L of an item tree.
atNode(S, N)	S is an item tree node belonging to decomposition node N.
root(S)	S is the root of an item tree.
rootOf(S, N)	S is the root of the item tree at decomposition node N.
leaf(S)	S is a leaf of an item tree.
leafOf(S, N)	S is a leaf of the item tree at decomposition node N.
sub(R, S)	R is an item tree node with child S.
childItem(S, I)	The item set of item tree node S contains I.
childAuxItem(S, I)	The auxiliary item set (for the default join) of item tree node S contains I.
childCost(S, C)	C is the cost value corresponding to the item tree leaf S.
childCounter(S, T, C)	C is the counter value corresponding to the item tree leaf S and the counter T.
childOr(S)	The type of the item tree node S is "or".
childAnd(S)	The type of the item tree node S is "and".
childAccept(S)	The type of the item tree leaf S is "accept".
childReject(S)	The type of the item tree leaf S is "reject".

Table 2 Input predicates describing item trees of child nodes in the decomposition

Output predicate	Meaning
`item`(L, I)	The item set of the node at level L of the current branch shall contain the item I.
`auxItem`(L, I)	The auxiliary item set (for the default join) of the node at level L of the current branch shall contain the item I.
`extend`(L, S)	The node at level L of the current branch shall extend the child item tree node S.
`cost`(C)	The leaf of the current branch shall have a cost value of C.
`currentCost`(C)	The leaf of the current branch shall have a current cost value of C.
`counter`(T, C)	The counter T of the current branch shall have a value of C.
`currentCounter`(T, C)	The current counter T of the current branch shall have a value of C.
`counterInc`(T, C)	The value of the counter T of the current branch shall be increased by a value of C.
`currentCounterInc`(T, C)	The value of the current counter T of the current branch shall be increased by a value of C.
`counterRem`(T)	The counter (and current counter) T of the current branch shall be removed.
`length`(L)	The current branch shall have length L.
`or`(L)	The node at level L of the current branch shall have type "or".
`and`(L)	The node at level L of the current branch shall have type "and".
`accept`	The leaf of the current branch shall have type "accept".
`reject`	The leaf of the current branch shall have type "reject".

Table 3 Output predicates for constructing the item tree of the current decomposition node

Errors and Warnings. In order to support its users, D-FLAT issues errors and warnings to avoid unintended behavior. To list the most important ones, an error is raised if one of the following conditions is violated.

- All output predicates from Table 3 are used with the correct arity.
- All atoms involving the `extend`/2 predicate refer to valid child item tree nodes.
- Each answer set contains exactly one atom involving `length`/1.
- Items, extension pointers or node types are placed at levels between 0 and the current branch length.
- All extension pointer tuples specified in an answer set have arity n, where n is the number of children in the decomposition.
- Items and auxiliary items are disjoint.
- Each extension pointer for level 0 points to the root of an item tree.
- Each extension pointer at level $n + 1$ points to a child of an extended node at level n.
- All answer sets agree on the (auxiliary) item sets and extension pointers at level 0.
- If a node type is specified for a leaf, it is "or" or "and".
- If a node type is specified for a non-leaf, it is "accept" or "reject".
- A node is assigned at most one type.
- In the final decomposition node, each "and" and "or" node must have at least one child with defined node type, or (in case such children have been pruned by D-FLAT) there must be some node reachable from that node via extension pointers that has a child with a defined node type.
- Only one (current) cost value is specified.
- Only one (current) counter value is specified for a certain first argument.
- Costs are specified only if all types of non-leaf nodes are defined.
- If in an answer set a `currentCost`/1 atom is contained, so must be an atom with `cost`/1.
- If in an answer set a `currentCounter`/2 atom is contained, so must be an atom with `counter`/2, that has the same first argument as the former atom.

Furthermore, a warning is printed if one of the following conditions is violated.

- At least one `#show` statement occurs in the user's encoding. (It should be used for performance reasons, and because D-FLAT can then check for specific `#show` statements, as described next. This is in order to check if the user has not forgotten about an output predicate that is usually required for obtaining reasonable results.)
- A `#show` statement for `length`/1 occurs in the program.
- A `#show` statement for `item`/2 occurs in the program.
- A `#show` statement for `extend`/2 occurs in the program.
- A `#show` statement for `or`/1 or `and`/1 occurs in the program.
- A `#show` statement for `accept`/0 or `reject`/0 occurs in the program.
- All predicates used in a `#show` statement are recognized by D-FLAT.

3.4.2 Simplified Interface for Problems in NP

When dealing with problems in NP, the user of D-FLAT can choose to use a simpler interface than the one in Section 3.4.1. The reason for providing a second, less general, interface is that for problems in that class it is usually sufficient to not use a tree-shaped data structure to store partial solutions, but rather to use just a "one-dimensional" data structure: a *table*.

For instance, for the GRAPH COLORING problem we could just store a table at each tree decomposition node. Each row of such a table would then encode a coloring of the bag vertices.

Answer sets in D-FLAT's table mode describe the rows of the current decomposition node's table. A table is an item tree of height 1 where the root always has the type "or" and an empty item set.[10] Each item tree node at depth 1 corresponds to a row in the table. At the final decomposition node, the type of each item tree node at level 1 is automatically set to "accept" by D-FLAT. The user therefore cannot (and does not need to) explicitly set item tree node types. Computations leading to a rejecting state should – instead of deriving "reject" – simply not yield an answer set, which can be achieved by means of a constraint.

This way, algorithms for problems in NP can be achieved that are usually quite easy to read (and write).

The input predicates describing the decomposition are the same as in the general case, listed in Table 1. The input predicates declaring the child item trees (which we now call "child tables") are different, though, and described in Table 4. The output predicates specifying the current table are also different in table mode. They are described in Table 5.

Example 6. *The* GRAPH COLORING *problem admits a D-FLAT encoding using the simplified ASP interface for problems in* NP. *A possible encoding using this table mode is shown in Listing 3.*

[10]The case that tables in D-FLAT are empty cannot occur: As soon as a call to the ASP solver does not yield any answer sets (presumably because the respective part of the problem does not allows for a solution), D-FLAT terminates and reports that no solutions exist.

```
%dflat: --tables -e vertex -e edge --no-empty-leaves --no-empty-root
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
item(map(X,C)) ← extend(R), childItem(R,map(X,C)), current(X).
← item(map(X,C0;X,C1)), C0 ≠ C1.
1 { item(map(X,C)) : color(C) } 1 ← introduced(X).
← edge(X,Y), item(map(X,C;Y,C)).
#show item/1. #show extend/1.
```

Listing 3 D-FLAT encoding for GRAPH COLORING using the table-mode ASP interface

Input predicate	Meaning
$\texttt{childRow}(R, N)$	R is a table row belonging to decomposition node N.
$\texttt{childItem}(R, I)$	The item set of table row R contains I.
$\texttt{childAuxItem}(R, I)$	The auxiliary item set (for the default join) of table row R contains I.
$\texttt{childCost}(R, C)$	C is the cost value corresponding to the table row R.
$\texttt{childCounter}(R, T, C)$	C is the counter value corresponding to the table row R and the counter T.

Table 4 Input predicates (in table mode) describing tables of child nodes in the decomposition

Output predicate	Meaning
`item`(I)	The item set of the current table row shall contain the item I.
`auxItem`(I)	The auxiliary item set (for the default join) of the current table row shall contain the item I.
`extend`(R)	The current table row shall extend the child table row R.
`cost`(C)	The current table row shall have a cost value of C.
`currentCost`(C)	The current table row shall have a current cost value of C.
`counter`(T, C)	The counter T of the current table row shall have a value of C.
`currentCounter`(T, C)	The current counter T of the current table row shall have a value of C.
`counterInc`(T, C)	The value of the counter T of the current table row shall be increased by a value of C.
`currentCounterInc`(T, C)	The value of the current counter T of the current table row shall be increased by a value of C.
`counterRem`(T)	The counter (and current counter) T of the current table row shall be removed.

Table 5 Output predicates (in table mode) for constructing the table of the current decomposition node

The first line differs from that in Listing 2 in the additional presence of the `--tables` *option to enable D-FLAT's table-mode interface. Line 2 guesses a predecessor row whose coloring of the current vertices is retained due to line 3. Line 4 makes sure that only compatible rows are joined. For the new vertices introduced into the current bag, line 5 guesses a coloring. The constraint in line 6 makes sure that the resulting coloring of the bag elements is proper. This way, each table will only contain rows that can be extended to proper colorings of the whole subgraph induced by the current bag and all vertices from bags further down in the decomposition. Line 7 again makes the ASP solver report only the relevant output predicates.*

Some of the rules in Listing 3 are suited (after small adjustments) for any problem that is to be solved with D-FLAT in table mode. In particular, a rule like in line 2 is usually part of all table-mode encodings. Such encodings typically guess over the current vertices and then check that this guess does not conflict with extended rows. Alternatively, as can be seen in the listing, it is often possible to retain information from extended rows (line 3) and only guess on introduced vertices (line 5). In any case, it has to be made sure that the extended rows do not contradict each other or the guessed information on the bag elements (line 4).

Errors and Warnings. In table mode, an error is raised if any of the following conditions is violated:

- All output predicates from Table 5 are used with the correct arity.
- All atoms involving the `extend`/2 predicate refer to valid child item tree nodes.
- All extension pointer tuples specified in an answer set have arity n, where n is the number of children in the decomposition.
- Items and auxiliary items are disjoint.
- Only one (current) cost value is specified.
- Only one (current) counter value is specified for a certain first argument.
- If in an answer set a `currentCost`/1 atom is contained, so must be an atom with `cost`/1.
- If in an answer set a `currentCounter`/2 atom is contained, so must be an atom with `counter`/2, that has the same first argument as the former atom.

3.5 D-FLAT's Handling of Item Trees

Every time the ASP solver reports an answer set of the user's program for the current tree decomposition node, D-FLAT creates a new branch in the current item tree, which results in a so-called *uncompressed item tree*. This step is described in Section 3.5.1. Subsequently D-FLAT prunes subtrees of that tree that can never be part of a solution, as described in Section 3.5.2, in order to avoid unnecessary computations in future decomposition nodes. For optimization problems,

D-FLAT then propagates information about the optimization values upward in the uncompressed item tree. This is described in Section 3.5.3. The item tree so far is called uncompressed because it may contain redundancies that are eliminated in the final step, as described in Section 3.5.4.

3.5.1 Constructing an Uncompressed Item Tree from the Answer Sets

In an answer set, all atoms using `extend`, `item`, `or` and `and` with the same depth argument, as well as `accept` and `reject`, constitute what we call a *node specification*. To determine where branches from different answer sets diverge, D-FLAT uses the following recursive condition: Two node specifications coincide (i.e., describe the same item tree node) iff

1. they are at the same depth in the item tree,
2. their item sets, counter values (for leaf nodes) – if specified, extension pointers and node types ("and", "or", "accept" or "reject") are equal, and
3. both are at depth 0, or their parent node specifications coincide.

In this way, an (uncompressed) item tree is obtained from the answer sets.

3.5.2 Propagation of Acceptance Statuses and Pruning of Item Trees

In Section 3.3.2 we have defined the different node types that an item tree node can have ("undefined", "or", "and", "accept" and "reject"). When D-FLAT has processed all answer sets and constructed the uncompressed item tree, these types come into play. That is to say, D-FLAT then prunes subtrees from the uncompressed item tree.

First of all, if the current tree decomposition node is the root, D-FLAT prunes from the uncompressed item tree any subtree rooted at a node whose type is still undefined.[11] Then, regardless of whether the current decomposition node is the root, D-FLAT prunes subtrees of the uncompressed item tree depending on the *acceptance status* of its nodes. The acceptance status of a node can either be "undefined", "accepting" or "rejecting", which we define now.

A node in an item tree is *accepting* if (a) its type is "accept", (b) its type is "or" and it has an accepting child, or (c) its type is "and" and all children are accepting. A node is *rejecting* if (a) its type is "reject", (b) its type is "or" and all children are rejecting, or (c) its type is "and" and it has a rejecting child. Nodes that are neither accepting nor rejecting are said to have an *undefined* acceptance status.

After having computed the acceptance status of all nodes in the current item tree, D-FLAT prunes all subtrees rooted at a rejecting node, as we can be sure that these nodes will never be part of a solution.

Note that in case the current decomposition node is the root, there are no nodes with undefined acceptance status because D-FLAT has pruned all subtrees rooted at nodes with undefined type. Therefore, in this case, the remaining tree consists only of accepting nodes. For decision problems,

[11] In order to keep the semantics meaningful, D-FLAT issues an error if an "and" or "or" node has only children with undefined type in the final decomposition node. It is the responsibility of the user to exclude such situations.

we can thus conclude that the problem instance is positive iff the remaining tree is non-empty. For enumeration problems, we can follow the extension pointers down to the leaves of the tree decomposition in order to obtain complete solutions by combining all item sets and counter values (for leaf nodes) – if specified – along the way. This is described in more detail in Section 3.6. Recursively extending all item sets in this way would yield a (generally very big) item tree that usually corresponds to the accepting part of a computation tree that an ATM would have when solving the complete problem instance. (But of course D-FLAT does not materialize this tree in memory.)

Example 7. *Consider again Figure 6. Because $\mathcal{I}_{\{b,d,e\}}$ is the final item tree in the decomposition traversal, D-FLAT would subsequently remove all nodes with undefined types (but there are none in this case). Then it would prune all rejecting nodes and conclude that the root of $\mathcal{I}_{\{b,d,e\}}$ is accepting because it has an accepting child. Therefore the problem instance is positive. At the decomposition root, we are then left with an item tree having only three leaves, each encoding a proper coloring of the vertices b, d and e, and storing extension pointers that let us extend the respective coloring for these vertices to proper colorings on all the other vertices, too.*

3.5.3 Propagation of Optimization Values in Item Trees

For optimization problems, after an uncompressed item tree has been computed, an additional step is done in addition to what we described in Section 3.5.2. Each leaf in the item tree stores an optimization value (or "cost") that has been specified by the user's program. D-FLAT now propagates these optimization values from the leaves toward the root of the current uncompressed item tree in the following way:

- The optimization value of a leaf node is its cost.
- The optimization value of an "or" node is the minimum among the optimization values of its children.
- The optimization value of an "and" node is the maximum among the optimization values of its children.

Note that D-FLAT currently raises an error if costs are specified for a leaf that has some ancestor with undefined node type.

3.5.4 Compressing the Item Tree

The uncompressed item tree obtained in the previous step may contain redundancies that must be eliminated in order to avoid an explosion of memory and runtime. The following two situations can arise where D-FLAT eliminates redundancies:

- There are two isomorphic sibling subtrees where all corresponding nodes have equal item sets, counter values (for leaf nodes), node types and (when solving an optimization problem) optimization values.

In this case, D-FLAT merges these subtrees into one and unifies their sets of extension pointers.

- An optimization problem is being solved and there are two isomorphic sibling subtrees where all corresponding nodes have equal item sets, counter values (for leaf nodes) – if specified – and node types, but the root of one of these sibling subtrees is "better" than the root of the other subtree. In this context, a node n_1 is "better" than one of its siblings, n_2, if the parent of n_1 and n_2 either has type "or" and the cost of n_1 is less than that of n_2, or their parent has type "and" and the cost of n_1 is greater than that of n_2.

  In this case, D-FLAT retains only the subtree rooted at the "better" node.

Note that in table mode this redundancy elimination can be done on the fly. That is, every time an answer set is reported, it can be detected right away if this leads to a new table row or if a table row with the same item set and the same counter values – if specified – already exists. This way, D-FLAT does not need to build an uncompressed item tree that can only be compressed once all answer sets have been processed. However, if item trees have a height greater than 1, this is not possible, as multiple answer sets constitute a subtree in general.

3.6 Materializing Complete Solutions

After all item trees have been computed, it remains to materialize complete solutions. We first describe how D-FLAT does this in the case of enumeration problems.

In the item tree at the root of the decomposition there are only accepting nodes left after the pruning described in Section 3.5.4. Starting with the root of this item tree, D-FLAT extends each of the nodes recursively by following the extension pointers, as we will describe next.

To obtain a complete extension of an item tree rooted at a node n, we first pick one of the extension pointer tuples of n. We then extend the item set of n by unifying it with all items in the nodes referenced by these extension pointers. Then we again pick one extension pointer tuple for each of the nodes that we have just used for extending n and repeat this procedure until we have reached a leaf of the tree decomposition. This gives us one of the possible extensions of n. For each of the children of n we also perform this procedure and add all possible extensions of that child to the list of children of the current extension of n. When extending a node m with parent n in this way, however, D-FLAT takes care to only pick an extension pointer tuple of m if every node that is being pointed to in this tuple is a child of a node that is used for the current extension of n.

When an optimization problem is to be solved, D-FLAT only materializes optimal solutions. That is, if n is an "or" node with optimization value c, D-FLAT only extends those children of n that actually have cost c. This is because, due to D-FLAT's propagation of optimization values (cf. Section 3.5.3), the optimization value of an "or" node is the minimum of the optimization values of its children. For "and" nodes this is symmetric.

D-FLAT allows the depth until which the final item tree is to be extended to be limited by the user. This is useful if, e.g., only existence of a solution shall be determined. In such a case, we could limit the materialization depth to 0. This would lead to only the root of the final item tree being extended. If D-FLAT yields an extension, a solution in fact exists. This is because

the final item tree would have no root in case no solutions existed (cf. Section 3.5.4). Limiting the materialization depth in this case saves us from potentially materializing exponentially many solutions when all we are only interested in knowing if there is a solution.

Moreover, limiting the materialization depth is also helpful for counting problems. If the user limits this depth to d and in the final item tree there is a node at depth d having children, D-FLAT prints for each extension of this node how many extended children would have been materialized. This behavior can be disabled to increase performance, but for the most common case, where the materialization depth is limited to 0, it is not required to disable this feature. The reason is that in such cases D-FLAT is able to calculate the number of possible extensions while doing the main bottom-up traversal of the decomposition for computing the item trees. Hence, for classical counting problems, D-FLAT offers quite efficient counting.

3.7 Further Functionality

3.7.1 Default Join

In weakly normalized, semi-normalized and normalized tree decompositions, nodes with more than one child are called join nodes. Such nodes have the same bag as each of its children (cf. Section 3.2). For join nodes, D-FLAT offers a default implementation for computing the item tree.

This *default join* implementation combines the item trees from the child nodes in the following way. For each child node, an item tree branch is selected such that for each pair of selected branches and each pair of nodes at the same depth in these branches it holds that the item sets, counter values (for leaf nodes) – if specified – and node types are equal. This is done for all possible combinations of item tree branches.

Once D-FLAT finds a combination of joinable branches, it inserts a branch whose item sets and counter values – if specified – are equal to those of the selected branches into the new item tree. In case costs (or counters) have been specified, D-FLAT first sets the cost (or the values of the specified counters) of this new branch to the sum of costs of the selected predecessors, but – similar to the inclusion-exclusion principle – costs (or counter values) that have been counted multiple times must be subtracted. This is why D-FLAT recognizes the output predicate `currentCost`/1 (and `currentCounter`/2, `currentCounterInc`/3+, (cf. Section 3.7.2)), which is used for quantifying the part of the cost (or counter), specified by `cost`/1 (or `counter`/2, `counterInc`/3+), that is due to current bag elements. If there are n child nodes in the decomposition, D-FLAT subtracts $n-1$ times the current cost (or counter) value from the sum of the predecessor costs (or counters).[12]

Example 8. *Consider again the* GRAPH COLORING *problem and the encoding from Listing 3. Suppose we only want to consider weakly normalized decompositions (or even stronger forms of normalizations) and use the default join implementation for all nodes with more than one child. Then we could remove line 4 from Listing 3 because the default join takes care of only combining item tree nodes (i.e., in this case, table rows) with equal item sets. The resulting encoding in combination with the default join would always yield the same results as Listing 3.*

[12]D-FLAT assumes that equality of item sets implies the same current cost (or counter) value, so each branch in a joinable combination features the same current cost (or counter) value.

Sometimes it is desired to join two branches even though they only agree on *some* part of the contained information. That is, we would like to express that some items are critical in determining whether two branches are joinable while some other items are irrelevant for this because they are some kind of auxiliary information.

For such situations, D-FLAT offers the possibility of using *auxiliary items*. Every item tree node thus contains, beside the ordinary item set, an auxiliary item set (that is disjoint from the ordinary one). Whenever we want to store an item in a node without preventing this node to be joined with nodes that do not contain this item, we can declare it as an auxiliary item. D-FLAT joins two branches even if their auxiliary item sets are different as long as their ordinary item sets, counter values – if specified – and item tree node types are equal. The auxiliary item sets in the branch resulting from the join is then set to the union of the respective auxiliary item sets of the nodes of the joined branches. For an example, see Section 4.3.

The same behavior could also be achieved in an ASP program. The default join implementation is, however, much more efficient than using an ASP solver. Firstly, of course the default join implies less overhead as it is implemented directly in D-FLAT using C++. Secondly, it can make use of the fact that D-FLAT stores item trees using a certain order that allows for very quick joining. This is similar to the sort-merge join algorithm in relational database management systems.

3.7.2 Built-in Counters

Starting with version 1.2.0, D-FLAT offers also so-called counters [2], which extend the functionality offered for tracking the cost of each solution to any kind of counter, and with which the user can outsource certain computations from ASP to D-FLAT. Keeping track of costs or other elements which need to be counted no longer requires using aggregate functions, leading to ample groundings and long run-times. Instead, D-FLAT offers the means to store the value for any defined counter, not only for the cost of a solution and also to increment these values instead of recalculating them in every ASP call.

First, the user can utilize the output predicates `counter`/2 and `currentCounter`/2, which both take a string and an integer value as parameters. The first sets the value of the counter denominated by the first argument, in the current table row, or the leaf of the current branch, respectively. If a counter with this denomination does not exist yet, it is first created. The second is needed when using the option `--default-join` and sets only the value of the counter corresponding to the current node, in order to be used by D-FLAT when applying the inclusion-exclusion principle to automatically merge counter values from different branches at join nodes. The use of these two predicates enhances higher readability and maintainability of the code and makes post-processing nodes with identical bag elements above the join node superfluous. For the user to be able to compute counter values in the current node, the input predicate `childCounter`/3 is printed for each counter, at each node of the tree decomposition, and passed as input for the parent node. Its arguments are the extended table row of the child node, or the extended item tree leaf of the child node, respectively, the denomination of the counter, and its value.

Further, D-FLAT offers the output predicates `counterInc`/3+ and `currentCounterInc`/3+, which can be used alternatively to `counter`/2 and `currentCounter`/2. With the former we can

specify via its second argument by how much a certain counter or current counter denominated by the first argument will be incremented (or decremented, when using negative values). The values to be incremented are taken by D-FLAT from the extended table row in the child node, or the extended item tree leaf in the child node, respectively. The rest of the arguments is used to make each instantiation of the predicate unique and their number depends on how many of them are needed for this purpose. Again, if the counter does not exist yet, it is created with the first occurrence of the predicate `counterInc`/3+. Using these predicates improves run-time dramatically in average, as they make the use of aggregate functions superfluous for keeping track of counters, otherwise having their values computed in ASP at every node of the tree decomposition or even handled using items. The greater the values of the counters are, the higher is the gain in running time.

For removing counters which will not be necessary anymore, D-FLAT provides the output predicate `counterRem`/1, whose argument is the denomination of the counter to be removed. Further, when the denomination of the counter is "cost", it acts as a cost declaration and D-FLAT optimizes the set of solutions specifically on the values stored in this counter. Further, if counters other than the one denominated by "cost" have different values for identical items sets, the partial answer sets are not merged and the counters are printed as if they were items. However, when using the option `--default-join`, the merge of different table rows is not done on the equality of counters, which are in fact recalculated based on the values in the child nodes and the current counter value. In this way, counters work rather like auxiliary items. For performance results on using built-in counters, please refer to [2]. Examples illustrating the use of built-counters can be found in Section 4.4.

3.7.3 Lazy Evaluation

Starting with version 1.2.0, D-FLAT also puts an implementation for "lazy" dynamic programming [14] at the user's disposal, when working in table mode. When the user opts for lazy evaluation by specifying the option `--lazy`, D-FLAT does not compute every row of a table before advancing to the next node of the tree decomposition but rather tries to deliver solutions by computing as few table rows as possible: Whenever a new row r is inserted into a table T, D-FLAT tries to extend r to a new row in the parent table. A new row is computed at T if all attempts of extending r have failed. Thus, in the best case, only one row per table needs to be computed in order to find a solution. By default, D-FLAT uses an ASP technology called "multi-shot solving" in order to avoid invoking the grounder whenever a new row has been computed. The user may turn off this behavior (with the option `--reground`), but experiments showed regrounding to be clearly less efficient. Lazy evaluation is especially useful for optimization problems. It may not only lead to better performance but also allows for anytime computations, that is, the possibility to abort the execution at any point and report the best solution found so far. The user may specify the option `--print-provisional`, which prints solutions that improve the current optimum as they are found.

Next, we will outline how lazy evaluation works on the whole.[13] D-FLAT runs many ASP systems in parallel – one for each tree decomposition node. These interact during the lazy evalua-

[13] For the detailed algorithms behind, please refer to [14].

tion, as we will describe in the following. After D-FLAT has constructed a tree decomposition, it requests a row from the root node. This triggers a recursion as follows:

When a leaf node is requested to produce a new row, its ASP solver is asked to find the next answer set (or, in the beginning, the first one). If this is impossible, the leaf reports that it is exhausted, that is, no new rows can be produced at its table.

When an exchange node is requested to produce a new row but the child table is empty, it first asks its child to produce a row. Then the exchange node invokes its ASP solver in order to compute a row at its own table by extending that child row. If this is impossible, it asks its child for the next row and repeats this process until it can produce a row. Subsequently, whenever the exchange node is requested to produce a new row, it first tries to obtain a new answer set from the last computed child row, and if this fails it asks the child for a new row. If at any point the child is exhausted, the exchange node is also exhausted.

For join nodes, the same idea is extended to more than one child. Here, D-FLAT first tries out all *combinations of existing child rows* and tries to compute a new row extending that combination. Only when all possible such combinations have been considered, a new row is requested from one of the child nodes in a round-robin fashion: A child node is only asked to produce a new row when all other nodes have been asked before (or are already exhausted). D-FLAT follows this round-robin strategy in the hope of obtaining solutions more quickly than by filling one child table completely before proceeding to the next one. Once a child table comes up with a new row, D-FLAT iterates over all combinations of this new row with existing rows of the other child nodes. For each such combination, the join node invokes its ASP solver in order to find all rows that extend that combination.

D-FLAT's lazy mode also incorporates some more optimizations that can be deactivated by the user if wanted. First, when working with the default join, D-FLAT keeps tables sorted; now whenever a child produces a new row, D-FLAT uses binary search in order to quickly find matching rows that already exist in the tables of the other children. By using the option `--no-binary-search` the user can deactivate this binary search and instead have all combinations be tried out sequentially. (Experiments showed, however, that binary search usually gives better performance.) Second, when solving optimization problems, D-FLAT uses the branch-and-bound strategy by default, in order to reject partial solution candidates that will for sure lead to solutions that are more expensive than the best solution found so far. For this, D-FLAT carries a global variable, which contains the cost of the best solution found so far. A newly found row candidate is inserted into a table only if its cost does not exceed this upper bound. If only this simple branch-and-bound procedure is wanted, it can be indicated using the option `--bb basic`. By default, however, D-FLAT implements a refined branch-and-bound strategy, which not only discards solution candidates when their cost exceeds the cost of the best solution so far, but it also tries to determine lower bounds and uses these for finding better upper bounds as follows: Suppose that node n has children n_1 and n_2, and that n_1 is exhausted, which intuitively means that we "know everything" about the subgraph induced by the vertices that have been forgotten up to n_1 (i.e., they appear in a bag of a node below n_1 but not in the bag of n_1 itself). Suppose further that c is the minimum cost of all partial solutions of the latter subgraph. Then, we can reject a candidate row at n_2 already when its cost exceeds $k - c$, where k is the cost of the best solution so far. This behavior is explicitly

chosen with the option `--bb full` and it is in fact the default. For the case that the user deals with a problem that does not exhibit monotone costs (i.e., the cost of a row is possibly lower than the sum of the costs of the rows it extends), the option `--bb none` turns off branch and bound.

As grounding is done only once at each node before having any information from rows in child tables, the user must declare atoms that pass on information from child rows (i.e., `childItem`/1, `childAuxItem`/1, `childCost`/1, `childCounter`/2 atoms) as `#external` in order for the grounder not to assume these atoms to be false even though they cannot be derived by any rule. Each time the ASP solver is called, the truth values of these atoms are temporarily frozen according to the child rows currently considered. Note that the arity of the former predicates is smaller than in the normal "eager" mode of D-FLAT. The reason is that the first argument, which would denote a child row in eager mode, is omitted in lazy mode. Moreover, the `extend`/1 predicate is not used at all in lazy mode. The reason is that D-FLAT now invokes the ASP solver once for every combination of child rows (instead of once per tree decomposition node), so it is already clear which combination of child rows is to be extended. Starting with version 1.2.5, D-FLAT's lazy evaluation mode also supports built-in counters and optimization on costs declared by the user. An example of such an encoding is provided in Section 4.5.

3.8 Command-Line Usage

In this section we present how D-FLAT can be used from the command line.

General Options

- `-h`: Print usage information and exit.
- `--version`: Print version number and exit.
- `-p <program>`: Use `<program>` as the ASP encoding that is to be called at each tree decomposition node. This option may be used multiple times, in which case all specified programs are jointly used.
- `-e <edge>`: Predicate `<edge>` declares (hyper)edges in the input facts.
  (This is explained in Section 3.2.)
- `-f <file>`: Read the problem instance from `<file>`. By default, standard input is used.
- `--seed <n>`: Initialize the random number generator with seed `<n>`.
- `--tables`: Use the table mode described in Section 3.4.2, i.e., the simplified ASP interface when a table suffices for storing problem-specific data. If this option is absent, D-FLAT uses the general interface described in Section 3.4.1.
- `--ignore-modelines`: Do not scan the encoding files for modelines, which are explained below.

- `--stats`: Print *Clasp* statistics.
- `--default-join`: Use the built-in default implementation for join nodes instead of invoking the ASP solver there. This is described in Section 3.7.1.

Options for Constructing a Tree Decomposition

- `-d <decomposer>`: Use decomposition method `<decomposer>`, which is one of the following.
  - `dummy`: Do not decompose (place all vertices into only one node).
  - `td`: Create a tree decomposition using bucket elimination. (This is the default.)
  - `graphml`: Use the decomposition specified in the GraphML file provided by the user.
- `-n <normalization>`: Use the normal form `<normalization>`, which is one of the following.
  - `none`: No normalization. (This is the default.)
  - `weak`: Weak normalization.
  - `semi`: Semi-normalization.
  - `normalized`: Normalization.
- `--elimination <h>`: Use the heuristic `<h>` for the bucket elimination algorithm. The following values are supported.
  - `min-degree`: Minimum degree ordering. (This is the default.)
  - `min-fill`: Minimum fill ordering.
- `--fitness <criterion>`: From several generated tree decompositions, choose the one with the `<criterion>` with the minimal value, where `<criterion>` can be one of the following.
  - `width`: Maximum bag size. (This is the default.)
  - `join-bag-avg`: Average join node bag size (defaulting to 0 if there is no join node in the tree decomposition).
  - `join-bag-median`: Median join node bag size (defaulting to 0 if there is no join node in the tree decomposition).
  - `num-joins`: Number of join nodes.
- `--iteration-count <i>`: Use `<i>` iterations for finding the fittest tree decomposition. (The default value is 1.)

  The decomposition strategies mentioned above are briefly described in Section 3.2.

- `--no-empty-root`: By default, D-FLAT constructs tree decompositions with an empty bag at the root. This option disables that behavior.
- `--no-empty-leaves`: By default, D-FLAT constructs tree decompositions with an empty bag at every leaf. This option disables that behavior.
- `--post-join`: Tree decompositions contain above each join node a parent node with identical bag. This can be useful especially when using default join and some post-processing needs to be done on the bag elements of the join node after merging them.
- `--path-decomposition`: Generate a path decomposition, which is a tree decomposition that does not contain any join nodes.

Solving Options

- `-s <solver>`: In order to compute partial solutions, use `<solver>`, which is one of the following.
  - `dummy`: Do not perform solving and always report the empty item tree.
  - `clasp`: Use Answer Set Programming solver *Clasp*. (This is the default.)
- `--no-optimization`: Do not optimize on costs.
- `--cardinality-cost`: Use item set cardinality as costs.
- `--depth <d>`: Limit materialization of final item tree to `<d>`.
  (This is explained in Section 3.6.)
- `--no-counting`: Do not count the number of solutions.
- `--no-pruning`: Prune rejecting subtrees only in the decomposition root (for instance for debugging).

Printing Options

- `--print-decomposition`: Print the generated decomposition.
- `--output <module>`: Print information during the run using `<module>`, which is one of the following.
  - `quiet`: Only print the final result of the computation.
  - `progress`: Print a progress report. (This is the default.)
  - `human`: Human-readable debugging output.

  - `machine`: Machine-readable debugging output that includes information about extension pointers (for instance for a debugging tool such as the D-FLAT Debugger for D-FLAT version 1.1.0, as described in [1]).
  - `count-rows`: Print the number of level-1 item tree nodes (or rows in table mode) for each decomposition node.

- `--print-solver-events`: Print events occurred during solving (for instance finding answer sets).
- `--print-solver-events`: Print the input passed on to the solver on its invocation.
- `--print-uncompressed`: Print item trees before compression. The mechanism for the latter is described in Section 3.5.4.

Options for Using the GraphML Format

- `--graphml-in <file>`: Read the decomposition in GraphML format from `<file>`.
- `--graphml-out <file>`: Write the decomposition in the GraphML format to `<file>`.
- `--add-empty-root`: When using a decomposition read in GraphML format from a file, add an empty root to it.
- `--add-empty-leaves`: When using a decomposition read in GraphML format from a file, add empty leaves to it.

Options for Lazy Evaluation

- `--lazy`: When in table mode, use lazy evaluation to find at least one solution. This is described in Section 3.7.3.
- `--no-binary-search`: Disable binary search on table rows to be matched at join nodes when using default join and lazy evaluation.
- `--bb <s>`: Using branch and bound strategy `<s>` when using lazy evaluation and optimizing on costs. The following values are permitted.
  - `none`: Do not use branch and bound.
  - `basic`: Prevent rows not cheaper then the current provisional solution.
  - `full`: Improve bounds for partial solution costs by using optimal cost values of partial solutions for forgotten subgraphs. (This is the default.)
- `--print-provisional`: Report possibly non-optimal solutions before determining the optimal solution(s) when using lazy evaluation (for instance when stopping the computation after a fixed amount of time).

- `--reground`: Reground every time a row is passed on from a child node instead of using external atoms when using lazy evaluation.

Modelines Usually a D-FLAT encoding only produces meaningful results if it is used in combination with certain command-line options. For instance, usually an encoding relies on a specific name of the (hyper)edge predicate or maybe a certain normalization of the tree decomposition. Because it is tedious to always specify these options, D-FLAT supports so-called *modelines* in the ASP encodings.

D-FLAT scans the first line of each encoding specified via `-p`. If this line starts with `%dflat:` (followed by a space), the rest of the line is treated as if it were supplied to the command line.

For instance, an encoding that requires the table mode ASP interface (cf. Section 3.4.2), seminormalized tree decompositions and an edge predicate called `edge` (as well as `vertex` for including isolated vertices in the decomposition – cf. Section 3.2), might start with the following modeline.

```
1 %dflat: --tables -e vertex -e edge -n semi
```

This way, D-FLAT only needs to be called with `-p` together with the encoding's filename.

4 D-FLAT in Practice

In this section we illustrate the applicability of D-FLAT to different types of computational problems. We will exemplify its functionality upon encodings of different variants of the DOMINATING SET problem, which we also refered to in Section 2.2.2. In Section 4.1 encodings for some variants of the problem in NP are described in detail. The same is done in Section 4.2 for a variant harder than NP. In Sections 4.3, 4.4, and 4.5 we illustrate how to use default join, built-in counters and lazy evaluation, using furhter D-FLAT encodings. Note that for actually running these encodings, `#show` statements should be added in order to avoid getting a warning (cf. Section 3.4). We omitted these statements for the sake of brevity.

For implementations of further problems in NP, as well as beyond NP, please refer to the first Progress Report from 2014 [1] and the project website[14]. There you can find also ecodings for the enumeration counterparts of NP-hard problems, such as MINIMUM VERTEX COVER, NP-complete problems like SAT, FEEDBACK VERTEX SET, HAMILTONIAN CYCLE, INDEPENDENT SET, MAXIMUM CUT, ODD CYCLE TRANSVERSAL, STEINER TREE, and even Σ_2^{P}-complete problems such as SECURE SET or $\subseteq$-MINIMAL SAT, and PSPACE-complete ones, like QSAT and MSO MODEL CHECKING.

4.1 Problems in NP

4.1.1 Minimum Dominating Set

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

Task: Compute all cardinality-minimal dominating sets of G. A subset X of V is a dominating set of G if for each $v \in V$ the vertex is part of X or v is adjacent to at least one $u \in X$ (i.e. is dominated by at least one $u \in X$).

ASP allows a very succinct modeling of the problem and also formulating it for D-FLAT yields a very simple encoding. The encoding (see Listing 4) represents one possibility of how to encode the optimization variant of DOMINATING SET in D-FLAT and implements a simple *Guess & Check* approach, following the ideas of the dynamic programming algorithm we introduced in Section 2.2.2. The code defines the recipe of how the information has to be handled in a node of the tree decomposition. We have to guess rows to be extended (line 5) and check whether these rows coincide on vertices contained in the dominating set (line 7). Information contained in a child row has to be retained in case that the corresponding vertex is still present in the current decomposition node (lines 9–10). In case a vertex is removed that is neither selected nor dominated, the respective solution candidate is discarded (line 12). For each introduced vertex we guess if it is selected to be part of a dominating set (line 14), and we derive which vertices are now dominated (line 16).

Up to now, there is no optimization involved. This task is done by employing the output predicate `cost`/1 in lines 17–22 of the encoding. For leaf nodes, the cost simply coincides with

[14] See `http://dbai.tuwien.ac.at/research/project/dflat/system/examples/encodings-dbai-tr-2017-107.tar.gz`

```
1  % Make explicit that edges are undirected.
2  edge(X,Y) ← current(X;Y), edge(Y,X).
3  % Make explicit that a vertex is not selected in a row.
4  out(R,X) ← childRow(R,N), bag(N,X), not childItem(R,sel(X)).
5  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
6  % Only join rows that coincide on selected vertices.
7  ← extend(R1;R2), childItem(R1,sel(X)), out(R2,X).
8  % Retain relevant information.
9  item(sel(X)) ← extend(R), childItem(R,sel(X)), current(X).
10 item(dom(X)) ← extend(R), childItem(R,dom(X)), current(X).
11 % Ensure that removed vertices are selected or dominated.
12 ← removed(X), extend(R), not childItem(R,sel(X)), not childItem(R,dom(X)).
13 % Guess selected vertices.
14 { item(sel(X)) : introduced(X) }.
15 % A vertex is dominated if it is adjacent to a selected vertex.
16 item(dom(Y)) ← item(sel(X)), edge(X,Y), current(X;Y).
17 % Leaf node costs.
18 cost(NC) ← NC = #count{ X : item(sel(X)) }, initial.
19 % Exchange node costs.
20 cost(CC + NC) ← extend(R), childCost(R,CC),
       NC = #count{ X : item(sel(X)), introduced(X) }, numChildNodes == 1.
21 % Join node costs.
22 cost(CC - (LC * (EP - 1))) ← CC = #sum { C,R : extend(R), childCost(R,C) },
       LC = #count { X : item(sel(X)) }, EP = numChildNodes, EP >= 2.
```

Listing 4 D-FLAT encoding for MINIMUM DOMINATING SET

the number of selected vertices. For exchange nodes we add the number of selected introduced vertices to the extended child row's cost. For join nodes we have to compute the sum of all the child rows' costs and subtract $\texttt{numChildNodes}-1$ times the current costs following the inclusion-exclusion principle. Note that we require the tree decomposition to be weakly normalized, as we assume in our cost function that the bag of a join node coincides with the bags of its child nodes.

Further note that this encoding only requires item trees of height 1, which allows for the use of the table-based D-FLAT interface. As described in Section 3.4.2, the input predicates $\texttt{childNode}/1$, $\texttt{childRow}/2$ specify the child nodes in the decomposition and the rows in their tables, and $\texttt{current}/1$, $\texttt{introduced}/1$ and $\texttt{removed}/1$ denote the vertices that are in the current bag, have been newly introduced in it or have appeared in a child node's bag but no longer are contained in the current bag, respectively. The output predicate $\texttt{item}/1$ (and $\texttt{auxItem}/1$) defines the (auxiliary) item set of the current row, and $\texttt{cost}/1$ defines the cost value of the current row. $\texttt{extend}/1$ determines for each child node the row to be extended by the current row. The input predicates $\texttt{childItem}/2$ (and $\texttt{childAuxItem}/2$), $\texttt{childCost}/2$ specify the item sets (and auxiliary item sets) of the child nodes and their cost values. $\texttt{initial}/0$ denotes that the current node is a leaf and $\texttt{numChildNodes}/0$ denotes the number of children of the current node.

4.1.2 Connected Dominating Set

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

Task: Compute all connected dominating sets of G. A subset X of V is a connected dominating set of G if the sub-graph G' of G induced by X is connected and for each $v \in V$ the vertex is part of X or v is adjacent to at least one $u \in X$.

As seen before, D-FLAT allows one to implement dynamic programming algorithms in a very succinct way. To underline that D-FLAT is also suitable for solving problems where the internal data structure, i.e., the information that is propagated through the decomposition nodes, is more complicated, we show how the CONNECTED DOMINATING SET problem can be solved with D-FLAT.

The encoding is given in Listing 5. Observe that lines 1–16 are the same as in the encoding for MINIMUM DOMINATING SET. We additionally use the function symbols $\texttt{stop}/0$, $\texttt{con}/2$ and $\texttt{est}/2$. For any pair of selected vertices, $\texttt{con}/2$ is derived if the vertices are connected (lines 18–20), while predicate $\texttt{est}/2$ denotes that there has not been any evidence of their connectedness yet (lines 22–23). Once a vertex x is removed, we check if x still needs to be connected to another selected vertex y in the current bag and if no connection to another selected vertex z in the current bag exists, the solution candidate is eliminated (line 24).

To avoid solutions containing isolated components, we denote by the function symbol $\texttt{stop}/0$ that the last selected vertex is removed from the bag (line 28). We then remove every solution candidate where a new vertex is introduced after `stop` was derived (line 29) or two rows are to be joined that both contain `stop` (line 30).

```
1  % Make explicit that edges are undirected.
2  edge(X,Y) ← current(X;Y), edge(Y,X).

3  % Make explicit that a vertex is not selected in a row.
4  out(R,X) ← childRow(R,N), bag(N,X), not childItem(R,sel(X)).

5  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).

6  % Only join rows that coincide on selected vertices.
7  ← extend(R1;R2), childItem(R1,sel(X)), out(R2,X).

8  % Retain relevant information.
9  item(sel(X)) ← current(X), extend(R), childItem(R,sel(X)).
10 item(dom(X)) ← current(X), extend(R), childItem(R,dom(X)).

11 % Ensure that removed vertices are selected or dominated.
12 ← removed(X), extend(R), not childItem(R,sel(X)), not childItem(R,dom(X)).

13 % Guess selected vertices.
14 { item(sel(X)) : introduced(X) }.

15 % A vertex is dominated if it is adjacent to a selected vertex.
16 item(dom(Y)) ← item(sel(X)), edge(X,Y), current(X;Y).

17 % Connectedness.
18 item(con(X,Y)) ← current(X;Y), extend(R), childItem(R,con(X,Y)).
19 item(con(X,Y)) ← current(X;Y), item(sel(X;Y)), edge(X,Y).
20 item(con(X,Z)) ← current(X;Y;Z), item(con(X,Y)), item(con(Y,Z)).

21 % If no connection between two selected vertices exists, it has to be established later.
22 item(est(X,Y)) ← current(X;Y), X ≠ Y, item(sel(X;Y)), not item(con(X,Y)).
23 item(est(X,Y)) ← current(X;Y), extend(R),
       childItem(R,est(X,Y)), not item(con(X,Y)).
24 ← removed(X), extend(R), childItem(R,est(X,Y)),
       not childItem(R,con(X,Z)) : current(Z).

25 % 'stop' is used to track and avoid getting several isolated components.
26 item(stop) ← extend(R), childItem(R,stop).
27 ok(X) ← removed(X), current(Y), extend(R), childItem(R,con(X,Y)).
28 item(stop) ← removed(X), extend(R), childItem(R,sel(X)), not ok(X).

29 ← current(X), item(stop), introduced(X).
30 ← extend(R1;R2), childItem(R1,stop), childItem(R2,stop), R1 < R2.
```

Listing 5 D-FLAT encoding for CONNECTED DOMINATING SET

```
1  length(2).
2  or(0).
3  and(1).

4  % Make explicit that edges are undirected.
5  edge(X,Y) ← current(X;Y), edge(Y,X).

6  % Make explicit that a vertex is not selected in an item set.
7  out(S,X) ← childNode(N), bag(N,X), atNode(S,N), not rootOf(S,N), not
       childItem(S,sel(X)).

8  % Guess item tree nodes to extend.
9  1 { extend(0,R) : rootOf(R,N) } 1 ← childNode(N).
10 1 { extend(L+1,S) : sub(R,S) } 1 ← extend(L,R), L < 2.

11 % Only join item tree nodes that coincide on selected vertices.
12 ← extend(L,R1), extend(L,R2), childItem(R1,sel(X)), out(R2,A), L=1..2.

13 % Retain relevant information.
14 item(L,sel(X)) ← extend(L,R), childItem(R,sel(X)), current(X), L=1..2.
15 item(L,dom(X)) ← extend(L,R), childItem(R,dom(X)), current(X), L=1..2.

16 % Ensure that removed vertices are selected or dominated.
17 ← removed(X), extend(L,R), not childItem(R,sel(X)), not childItem(R,dom(X)),
       L=1..2.

18 % Guess selected vertices, such that set on level 2 is a subset.
19 { item(1,sel(X)) : introduced(X) }.
20 { item(2,sel(X)) } ← introduced(X), item(1,sel(X)).

21 % A vertex is dominated if it is adjacent to a selected vertex.
22 item(L,dom(Y)) ← item(L,sel(X)), edge(X,Y), current(X;Y), L=1..2.

23 % Store if a proper subset has been guessed for the second level.
24 item(2,smaller) ← extend(2,S), childItem(S,smaller).
25 item(2,smaller) ← item(1,sel(X)), not item(2,sel(X)).

26 % Ensure subset-minimality.
27 reject ← final, item(2,smaller).
28 accept ← final, not reject.
```

Listing 6 D-FLAT encoding for $\subseteq$-MINIMAL DOMINATING SET

4.2 A Problem beyond NP: Subset-Minimal Dominating Set

> Input: An undirected graph $G = (V, E)$.
>
> Task: Compute all subset-minimal dominating sets of G. A subset X of V is a dominating set of G if for each $v \in V$ the vertex is part of X or v is adjacent to at least one $u \in X$.

An example of a problem above NP is $\subseteq$-MINIMAL DOMINATING SET. As the problem is located at the second level of the polynomial hierarchy, we can rely on the capabilities of D-FLAT which allow an easy representation of these levels. The increased complexity is reflected

in D-FLAT by the necessity of adding an additional level to the item trees. Listing 6 shows that this is done by specifying the predicates `length/2`, `and/1` and `or/1`. Via the predicate `length/2`, D-FLAT is instructed to create item tree branches of length 2. The facts `or(0)` and `and(1)` are used to inform D-FLAT that the problem instance is positive if there is an accepting node at depth 1 such that all its children are accepting. Similar to other problems of the same complexity, also $\subseteq$-MINIMAL DOMINATING SET is tackled by guessing a solution candidate for the basic problem variant on the first level. On the second level, for each subset of the selected vertices of level 1 it has to hold that it is no dominating set unless the subset and the original selection coincide.

In the first lines of the encoding, we specify item tree branches of length 2, and we define that the first level is, notionally, existentially quantified, while the second one is universally quantified. This way we can capture problems in Σ_2^P. The use of item tree branches of length greater than 1 requires slight modifications to the original encoding for the MINIMUM DOMINATING SET problem. Syntactically, the arity of some predicates (among them are `extend`, `item` and `auxItem`) increases by one because we need to specify at which item tree branch level we want to store, e.g., an item. Clearly, we also need additional code for checking the subset-minimality of a guess on the first level. As the tasks we want to compute at both levels are very similar, we could simply copy most of the rules from the encoding of MINIMUM DOMINATING SET and then adapt them. Here, the modeling language of Gringo comes into play and allows us to simply reuse almost all existing rules without the need for duplicated code. In this example, this is achieved by adding `L=1..2` to the relevant rules where a level specification is needed. The variable `L` is then instantiated to both constants `1` and `2` by the grounder, allowing us to capture both levels by writing only one rule.

In detail, the algorithm presented in Listing 6 works as follows. Similar to the encoding for the MINIMUM DOMINATING SET problem, we guess for each level which item tree node is extended, and via a constraint we ensure that the selected vertices coincide. Afterwards, we guess for each introduced vertex if it is selected at the first level and if this is the case, we guess if it is also selected at the second level. This way, it is ensured that the selection on the second level is always a subset of the selection on the first level and we derive the atom `item(2,smaller)` when it is a strict subset. To remove all solution candidates that are not subset-minimal, we derive `reject` in the final node of the tree decomposition whenever there is evidence of a smaller dominating set. All remaining candidates are then known to be indeed valid solutions and we derive `accept` for them.

As it turns out that DP algorithms for problems of the second level of the polynomial hierarchy often follow recurring patterns as sketched above, there is also a "spin-off" of D-FLAT, called D-FLAT^2, that provides dedicated functionality for problems of this kind. As an example D-FLAT^2 generates the DP algorithm for $\subseteq$-MINIMAL DOMINATING SET directly from an encoding for the standard DOMINATING SET problem. Describing this approach requires additional formal machinery, and thus, we decided to not discuss D-FLAT^2 in this report. The reader is referred to [13] for further reading.

4.3 Default Join in Practice

In this section we will present the use of the `--default-join` option presented in Section 3.7.1, by comparing an encoding of the MINIMUM DOMINATING SET problem which uses default join,

```
1  % Make explicit that edges are undirected.
2  edge(X,Y) ← current(X;Y), edge(Y,X).
3  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
4  % Retain relevant information.
5  item(sel(X)) ← extend(R), childItem(R,sel(X)), current(X).
6  auxItem(dom(X)) ← extend(R), childAuxItem(R,dom(X)), current(X).
7  % Ensure that removed vertices are selected or dominated.
8  ← removed(X), extend(R), not childItem(R,sel(X)), not childAuxItem(R,dom(X)).
9  % Guess selected vertices.
10 { item(sel(X)) : introduced(X) }.
11 % A vertex is dominated if it is adjacent to a selected vertex.
12 auxItem(dom(Y)) ← item(sel(X)), edge(X,Y), current(X;Y).
13 % Leaf node costs.
14 cost(NC) ← NC = #count{ X : item(sel(X)) }, initial.
15 % Exchange node costs.
16 cost(CC + NC) ← extend(R), childCost(R,CC),
       NC = #count{ X : item(sel(X)), introduced(X) }, numChildNodes == 1.
17 % Costs corresponding only to current node.
18 currentCost(CC) ← CC = #count{ W,X : item(sel(X)), weight(X,W) }.
```

Listing 7 D-FLAT encoding for MINIMUM DOMINATING SET with default join

in Listing 7 to the one in Listing 4, which does not. Both are designed for weakly normalized tree decompositions and, as we only need item trees of height 1, they use D-FLAT's simplified interface presented in Section 3.4.2. Using default join makes the rules in lines 4 and 7 from the encoding in Listing 4 superfluous, as the merge on specific items is done implicitly at join nodes. Nevertheless, we do need to specify on which types of items the merge should be done. As the final solutions must be consistent in the vertices belonging to the dominating sets, also the merge must be done only on items of `sel`/1. This is why `dom`/1 is an auxiliary item in Listing 7. Further, instead of calculating the cost value in join nodes ourselves (line 22 in Listing 4), we just need to calculate the current cost, as seen in line 18 in Listing 7.

4.4 Built-in Counters in Practice

Next, we will present the use of built-in D-FLAT counters in practice, based on encodings of variants of the MINIMUM DOMINATING SET problem for weakly normalized tree decompositions, using D-FLAT's simplified interface presented in Section 3.4.2 and the option `--default-join` presented in Section 3.7.1. We will show how to use the output predicates `counterInc`/3+ and `currentCounterInc`/3+ by comparing an encoding of the MINIMUM WEIGHTED DOMINATING SET problem which uses counter incrementation for calculating costs with one that does not. Further, we will illustrate the use of the output predicates `counter`/2 and `currentCounter`/2 by comparing encodings of the GENERALIZED MINIMUM WEIGHTED DOMINATING SET problem with and without built-in counters for every vertex to keep track of the number of dominating

```
1  % Make explicit that edges are undirected.
2  edge(X,Y) ← current(X;Y), edge(Y,X).
3  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
4  % Retain relevant information.
5  item(sel(X)) ← extend(R), childItem(R,sel(X)), current(X).
6  auxItem(dom(X)) ← extend(R), childAuxItem(R,dom(X)), current(X).
7  % Ensure that removed vertices are selected or dominated.
8  ← removed(X), extend(R), not childItem(R,sel(X)), not childAuxItem(R,dom(X)).
9  % Guess selected vertices.
10 { item(sel(X)) : introduced(X) }.
11 % A vertex is dominated if it is adjacent to a selected vertex.
12 auxItem(dom(Y)) ← item(sel(X)), edge(X,Y), current(X;Y).
13 % Leaf node costs.
14 cost(NC) ← NC = #sum{ W,X : item(sel(X)), weight(X,W) }, initial.
15 % Exchange node costs.
16 cost(CC + NC) ← extend(R), childCost(R,CC),
       NC = #sum{ W,X : item(sel(X)), introduced(X), weight(X,W) }.
17 % Costs corresponding only to current node.
18 currentCost(CC) ← CC = #sum{ W,X : item(sel(X)), weight(X,W) }.
```

Listing 8 D-FLAT encoding for MINIMUM WEIGHTED DOMINATING SET

vertices.

We define the MINIMUM WEIGHTED DOMINATING SET problem as follows:

> Input: An undirected graph $G = (V, E)$ with weighted vertices.
>
> Task: Compute all weight-minimal dominating sets of G. A subset X of V is a dominating set of G if for each $v \in V$ the vertex is part of X or v is adjacent to at least one $u \in X$.

In Listing 8 we see a possibility to encode the optimization variant of MINIMUM DOMINATING SET in D-FLAT, where we minimize on the sum of vertex weights instead of the cardinality of the dominating set. As we only need item trees of height 1, we work again in table mode, and with default join, like in Listing 7. Compared to the latter, the encoding in Listing 8 uses a `sum` instead of a `count` aggregate, as the costs represent the sum of the weights of the selected vertices.

In Listing 9 we replace lines 13–18 from Listing 8 in order to illustrate the technique of incrementing counter values, in our case costs, instead of recalculating them in every ASP call. The rule in line 14 will be instantiated by the grounder as many times as many selected vertices there are in the current partial solution candidate and D-FLAT will add each weight value to the cost of the latter. As we use the `--default-join` option we also increment the value of the current counter by the weights of newly introduced vertices (line 16) and decrement it by the weights of the removed vertices (line 18).

```
1  % Make explicit that edges are undirected.
2  edge(X,Y) ← current(X;Y), edge(Y,X).
3  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
4  % Retain relevant information.
5  item(sel(X)) ← extend(R), childItem(R,sel(X)), current(X).
6  auxItem(dom(X)) ← extend(R), childAuxItem(R,dom(X)), current(X).
7  % Ensure that removed vertices are selected or dominated.
8  ← removed(X), extend(R), not childItem(R,sel(X)), not childAuxItem(R,dom(X)).
9  % Guess selected vertices.
10 { item(sel(X)) : introduced(X) }.
11 % A vertex is dominated if it is adjacent to a selected vertex.
12 auxItem(dom(Y)) ← item(sel(X)), edge(X,Y), current(X;Y).
13 % Increment the cost by the weights of newly introduced vertices.
14 counterInc(cost,W,X) ← introduced(X), item(sel(X)), weight(X,W).
15 % Increment the current cost by the weights of newly introduced vertices.
16 currentCounterInc(cost,W,X) ← introduced(X), item(sel(X)), weight(X,W).
17 % Decrement the current cost by the weights of newly removed vertices.
18 currentCounterInc(cost,-W,X) ← removed(X), extend(R), childItem(R,sel(X)),
       weight(W,X).
```

Listing 9 D-FLAT encoding for MINIMUM WEIGHTED DOMINATING SET using counter incrementation for cost calculation

Next, we will show how to use counters based on the GENERALIZED MINIMUM WEIGHTED PERFECT DOMINATING SET problem, which we define as follows:

> Input: An undirected graph $G = (V, E)$ with weighted vertices and upper and/or lower bounds defining for a vertex by how many other vertices it may be dominated.
>
> Task: Compute all weight-minimal dominating sets of G such that the defined bounds are respected. A subset X of V is a dominating set of G if for each $v \in V$ the vertex is part of X or v is adjacent to at least one $u \in X$ (i.e. is dominated by at least one $u \in X$).

Listing 10 shows the D-FLAT encoding for the GENERALIZED MINIMUM WEIGHTED PERFECT DOMINATING SET problem, which implements the changes needed for the vertices to respect the given bounds, as compared to the encoding in Listing 8. Instead of using an auxiliary item `dom`/1, which tells that the vertex given as its only argument is dominated, we need one that takes also the number of vertices our vertex is dominated by and the name of the current node as arguments and in fact tells for each vertex by how many vertices it is dominated (in case it is not dominated, the value of the second argument is 0). Storing the name of the current node is necessary as we need a supplementary node above each join node bearing an identical bag (option `--post-join`). There we can adjust the values of `dom`/3 items, as the default join just passes

```
1  % Make explicit that edges are undirected.
2  edge(X,Y) ← current(X;Y), edge(Y,X).
3  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
4  % Retain which vertices have already been selected.
5  item(sel(X)) ← extend(R), childItem(R,sel(X)), current(X).
6  % Ensure that removed vertices are selected and respect domination bounds.
7  ← removed(X), extend(R), not childItem(R,sel(X)),
       childAuxItem(R,dom(X,N,CR)), lowerBound(X,B), N<B.
8  ← removed(X), extend(R), not childItem(R,sel(X)),
       childAuxItem(R,dom(X,N,CR)), upperBound(X,B), N>B.
9  % Guess selected vertices.
10 { item(sel(X)) : introduced(X) }.
11 % Deduct for each vertex by how many other vertices it is dominated.
12 auxItem(dom(Y,N,CR)) ← extend(R), currentNode(CR), introduced(Y),
       not item(sel(Y)), 1 #count { CH: childAuxItem(R,n(CH)) } 1,
       N = #count { X : item(sel(X)), edge(X,Y) }.
13 auxItem(dom(Y,N1+N0,CR)) ← extend(R), currentNode(CR), current(Y),
       1 #count { CH: childAuxItem(R,n(CH)) } 1, childAuxItem(R,dom(Y,N1,CH1)),
       N0 = #count { X : item(sel(X)), edge(X,Y), introduced(X) }.
14 auxItem(dom(Y,N1+N2-N12,CR)) ← extend(R), currentNode(CR), current(Y),
       childAuxItem(R,dom(Y,N1,CH1)), childAuxItem(R,dom(Y,N2,CH2)), CH1 ≠ CH2,
       N12 = #count { X : item(sel(X)), edge(X,Y) }.
15 auxItem(n(CR)) ← currentNode(CR).
16 % Leaf node costs.
17 cost(NC) ← NC = #sum{ W,X : item(sel(X)), weight(X,W) }, initial.
18 % Exchange node costs.
19 cost(CC + NC) ← extend(R), childCost(R,CC),
       NC = #sum{ W,X : item(sel(X)), introduced(X), weight(X,W) }.
20 % Costs corresponding only to current node.
21 currentCost(CC) ← CC = #sum { W,X : item(sel(X)), weight(X,W) }.
```

Listing 10 D-FLAT encoding for GENERALIZED MINIMUM WEIGHTED PERFECT DOMINATING SET

auxiliary items through without merging them and we need to do this explicitly. In line 12 we count for each vertex which is newly introduced into an exchange node, by how many other vertices it is dominated, in line 13 we calculate the latter value for vertices which are not newly introduced by adding the number of dominating vertices in the child node to the number of newly introduced dominating vertices and in line 14 we merge the values coming from different branches below the join node, by the inclusion-exclusion principle. In line 15 we define the `n`/1 predicate which is necessary when checking that we are not dealing with a post-processing node. Further, instead of line 8 from Listing 8, we now have two constraint rules in lines 7–8 which also check for each vertex the lower and upper bound of dominating vertices, respectively.

Listing 11 illustrates how such an encoding can be simplified by means of counters. Instead of `dom`/3 auxiliary items we now use counters which implicitly keep track of these values and let the

```
1  % Make explicit that edges are undirected.
2  edge(X,Y) ← current(X;Y), edge(Y,X).
3  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
4  % Retain which vertices have already been selected.
5  item(sel(X)) ← extend(R), childItem(R,sel(X)), current(X).
6  % Ensure that removed vertices are selected and respect domination bounds.
7  ← removed(X), extend(R), not childItem(R,sel(X)), childCounter(R,X,N),
       lowerBound(X,B), N<B.
8  ← removed(X), extend(R), not childItem(R,sel(X)), childCounter(R,X,N),
       upperBound(X,B), N>B.
9  % Guess selected vertices.
10 { item(sel(X)) : introduced(X) }.
11 % Keep track for each vertex by how many other vertices it is dominated, using counters.
12 counter(Y,N) ← introduced(Y), not item(sel(Y)),
       N = #count { X : item(sel(X)), edge(X,Y) }.
13 counter(Y,N0+N1) ← extend(R), current(Y), childCounter(R,Y,N1),
       N0 = #count { X : item(sel(X)), edge(X,Y), introduced(X) }.
14 currentCounter(Y,N) ← current(Y), not item(sel(Y)),
       N = #count { X: item(sel(X)), edge(X,Y) }.
15 counterRem(Y) ← removed(Y).
16 % Leaf node costs.
17 cost(NC) ← NC = #sum{ W,X : item(sel(X)), weight(X,W) }, initial.
18 % Exchange node costs.
19 cost(CC + NC) ← extend(R), childCost(R,CC),
       NC = #sum{ W,X : item(sel(X)), introduced(X), weight(X,W) }.
20 % Costs corresponding only to current node.
21 currentCost(CC) ← CC = #sum { W,X : item(sel(X)), weight(X,W) }.
```

Listing 11 D-FLAT encoding for GENERALIZED MINIMUM WEIGHTED PERFECT DOMINATING SET with counters

default join do the merge such that a post-processing node is not necessary anymore. Lines 12–13 have the same role as before, yet they are simpler because there is no need to check whether we are in a post-processing node and we can make use of the input predicate `childCounter`/3. Next, instead of having to explicitly merge values in the post-processing node, we just need to keep track of the value of the counter corresponding to the current tree decomposition node, which is done in line 14. In order to not carry around unnecessary counters in the system, in line 15 we remove a counter if the corresponding vertex has been removed from the tree decomposition. Further, in lines 7–8 we can now also use `childCounter`/3, instead of `childAuxItem`/2.

4.5 Lazy Evaluation in Practice

In order to illustrate how to employ D-FLAT with lazy evaluation, we adapted the D-FLAT encoding from Listing 8 of the previously defined MINIMUM WEIGHTED DOMINATING SET problem

```
1  % Make explicit that edges are undirected.
2  edge(X,Y) ← current(X;Y), edge(Y,X).
3  % Declare atoms passing on information from child rows.
4  #external childItem(sel(X)) : childNode(N), bag(N,X).
5  #external childAuxItem(dom(X)) : childNode(N), bag(N,X).
6  #external childCost(0..N) : maxSumOfWeights(N).
7  % Retain relevant information.
8  item(sel(X)) ← childItem(sel(X)), current(X).
9  auxItem(dom(X)) ← childAuxItem(dom(X)), current(X).
10 % Ensure that removed vertices are selected or dominated.
11 ← removed(X), not childItem(sel(X)), not childAuxItem(dom(X)).
12 % Guess selected vertices.
13 { item(sel(X)) : introduced(X) }.
14 % A vertex is dominated if it is adjacent to a selected vertex.
15 auxItem(dom(Y)) ← item(sel(X)), edge(X,Y), current(X;Y).
16 % Leaf node costs.
17 cost(NC) ← NC = #sum{ W,X : item(sel(X)), weight(X,W) }, initial.
18 % Exchange node costs.
19 cost(CC + NC) ← childCost(CC),
       NC = #sum{ W,X : item(sel(X)), introduced(X), weight(X,W) }.
20 % Costs corresponding only to current node.
21 currentCost(CC) ← CC = #sum { W,X : item(sel(X)), weight(X,W) }.
```

Listing 12 D-FLAT encoding for solving MINIMUM WEIGHTED DOMINATING SET with lazy evaluation

for this purpose, resulting in Listing 12, which also makes use of the `--default-join` option. Compared to the former encoding, the one for lazy evaluation does not need the rule in line 3 anymore since grounding is done as an initial step at each table without information about any child row and the truth values of the atoms that depend on child rows are successively set to those corresponding to the child row currently considered. Instead, we need lines 4–5 so the grounder will not assume an atom `childItem(sel(X))` or `childAuxItem(dom(X))` to be false, where `X` is a vertex in the bag of the child node. Furthermore, in line 6, we declare which are the possible values for `childCost`/1. For this, we expect that the input contains a fact `maxSumOfWeights(W)`, where `W` is the maximum cost that a partial solution may have. Everything else stays the same as in Listing 8, except for the fact that there is no reference to any child table row and the `extend`/1 predicate is not used anymore.

5 Conclusion

In this report we presented the D-FLAT system (as of version 1.2.5) for developing algorithms that employ dynamic programming on tree decompositions. The key feature of D-FLAT is that it lets the user specify such algorithms in the declarative language of ASP. This makes it well suited for rapid prototyping.

After presenting the background on ASP and tree decompositions, we explained the different components of D-FLAT in detail, as well as the interface that is used for communication between the system and the problem-specific encodings. We discussed new features that have been added over time, such as built-in counters and lazy evaluation, and we presented the improvements brought by using the *htd* framework. Then we showed how D-FLAT can be applied to a selection of problems to underline the usability of our approach.

The fact that we have been able to come up with relatively simple D-FLAT encodings for various different problems shows that our system is indeed well suited for rapid prototyping of decomposition-based algorithms. Even more, D-FLAT can be a competitive solver in certain settings, in particular if the treewidth of the instances to be solved is small. The proposed ASP interface is general enough to accommodate quite different kinds of problems.

There are a number of related works that concern ASP and tree decomposition techniques, some of which originate from the experience we gained in developing D-FLAT. DYNASP [34, 35] is a new ASP solver for ground answer-set programs that employs dynamic programming on tree decompositions. Such a system can be beneficial if the (incidence graph of a) ground program has rather small treewidth. As has been shown recently, for certain classes of ASP encodings the treewidth of the grounded program remains small given that the instance also has small treewidth [15]. Finally, we would like to mention the lpopt tool [11] that aims at decomposing non-ground rules.

References

[1] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. D-FLAT: Progress report. Technical Report DBAI-TR-2014-86, DBAI, Vienna University of Technology, 2014.

[2] Michael Abseher, Marius Moldovan, and Stefan Woltran. Providing built-in counters in a declarative dynamic programming environment. In *Proc. KI*, volume 9904 of *LNCS*, pages 3–16. Springer, 2016.

[3] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd–a free, open-source framework for tree decompositions and beyond. Technical Report DBAI-TR-2014-96, DBAI, Vienna University of Technology, 2016.

[4] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd - A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond. In *Proc. CPAIOR*, volume 10335 of *LNCS*, pages 376–386. Springer, 2017.

[5] Michael Abseher, Nysret Musliu, and Stefan Woltran. Improving the efficiency of dynamic programming on tree decompositions via machine learning. *Journal of Artificial Intelligence Research*, 58:829–858, 2017.

[6] Rachit Agarwal, Philip Brighten Godfrey, and Sariel Har-Peled. Approximate distance queries and compact routing in sparse graphs. In *Proc. INFOCOM*, pages 1754–1762. IEEE, 2011.

[7] Mario Alviano, Francesco Calimeri, Wolfgang Faber, Giovambattista Ianni, and Nicola Leone. Function symbols in ASP: Overview and perspectives. In *NMR – Essays Celebrating Its 30th Anniversary*, pages 1–24. College Publications, London, 2011.

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

[9] Markus Aschinger, Conrad Drescher, Georg Gottlob, Peter Jeavons, and Evgenij Thorstensen. 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, 2011.

[10] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[11] Manuel Bichler, Michael Morak, and Stefan Woltran. lpopt: A rule optimization tool for answer set programming. In *Proc. LOPSTR*, volume 10184 of *LNCS*, pages 114–130. Springer, 2017.

[12] Bernhard Bliem. Decompose, Guess & Check: Declarative problem solving on tree decompositions. Master's thesis, Vienna University of Technology, 2012.

[13] Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran. D-FLATˆ2: Subset minimization in dynamic programming on tree decompositions made easy. *Fundamenta Informaticae*, 147(1):27–61, 2016.

[14] Bernhard Bliem, Benjamin Kaufmann, Torsten Schaub, and Stefan Woltran. ASP for anytime dynamic programming on tree decompositions. In *Proc. IJCAI*, pages 979–986. IJCAI/AAAI Press, 2016.

[15] Bernhard Bliem, Marius Moldovan, Michael Morak, and Stefan Woltran. The impact of treewidth on ASP grounding and solving. In *Proc. IJCAI*, pages 852–858. ijcai.org, 2017.

[16] Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. Declarative dynamic programming as an alternative realization of Courcelle's theorem. In *Proc. IPEC*, volume 8246 of *LNCS*, pages 28–40. Springer, 2013.

[17] Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. Implementing Courcelle's Theorem in a declarative framework for dynamic programming. *Journal of Logic and Computation*, 2017. Early access available (DOI 10.1093/logcom/exv089), to appear.

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

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

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

[21] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, 51(3):255–269, 2008.

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

[23] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[24] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: Theory and implementation. In *Proc. ICLP*, volume 5366 of *LNCS*, pages 407–424. Springer, 2008.

[25] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[26] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Springer, 1977.

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

[28] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[29] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artif. Intell.*, 113(1–2):41–85, 1999.

[30] Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The first parameterized algorithms and computational experiments challenge. In *Proc. IPEC*, volume 63 of *LIPIcs*, pages 30:1–30:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[31] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *Proc. MICAI*, volume 5317 of *LNCS*, pages 1–11. Springer, 2008.

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

[33] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997.

[34] Johannes Klaus Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Answer set solving with bounded treewidth revisited. In *Proc. LPNMR*, volume 10377 of *LNCS*, pages 132–145. Springer, 2017.

[35] Johannes Klaus Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Dynasp2.5: Dynamic programming on tree decompositions in action. *CoRR*, abs/1706.09370, 2017.

[36] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo. Second Edition. Available at `http://potassco.sourceforge.net`, 2015.

[37] Martin Gebser, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Sven Thiele. On the input language of ASP grounder gringo. In *Proc. LPNMR*, volume 5753 of *LNCS*, pages 502–508. Springer, 2009.

[38] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The Potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

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

[40] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proc. ICLP/SLP*, pages 1070–1080. The MIT Press, 1988.

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

[42] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[43] Jens Gramm, Arfst Nickelsen, and Till Tantau. Fixed-parameter algorithms in phylogenetics. *Comput. J.*, 51(1):79–101, 2008.

[44] Xiuzhen Huang and Jing Lai. Parameterized graph problems in computational biology. In *Proc. IMSCCS*, pages 129–132. IEEE, 2007.

[45] Matthieu Latapy and Clémence Magnien. Measuring fundamental properties of real-world complex networks. *CoRR*, abs/cs/0609115, 2006.

[46] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

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

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

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

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

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

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

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

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

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

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

[57] Atsuko Yamaguchi, Kiyoko F. Aoki, and Hiroshi Mamitsuka. Graph complexity of chemical compounds in biological pathways. *Genome Inform.*, 14:376–377, 2003.