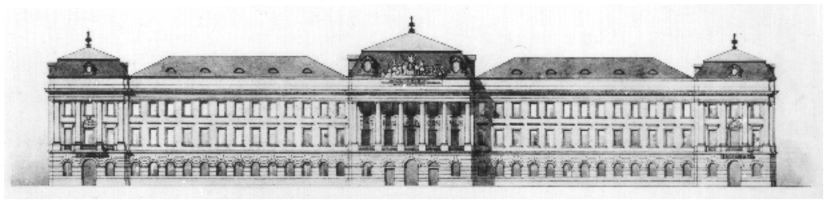
dbai

TECHNICAL REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

D-FLAT: Progress Report

DBAI-TR-2014-86

Michael Abseher **Bernhard Bliem**
Günther Charwat **Frederico Dusberger**
Markus Hecher **Stefan Woltran**

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
sekret@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

DBAI TECHNICAL REPORT
2014

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI TECHNICAL REPORT

DBAI TECHNICAL REPORT DBAI-TR-2014-86, 2014

D-FLAT: Progress Report

Michael Abseher [1] **Bernhard Bliem** [1] **Günther Charwat** [1]
Frederico Dusberger [1] **Markus Hecher** [1] **Stefan Woltran** [1]

Abstract. Complex reasoning problems over large amounts of data pose a great challenge for Artificial Intelligence and Knowledge Representation. To cope with it, it is desirable to use declarative approaches, as these offer high maintainability and are often easy to use for domain experts. Especially Answer Set Programming (ASP) has become prominent due to the existence of powerful solvers that offer high efficiency and rich modeling languages. To overcome the obstacle of high computational complexity, exploiting structure by means of tree decompositions has proved to be effective in many cases. However, the implementation of suitable efficient algorithms is often tedious. D-FLAT is a software system that combines ASP with problem solving on tree decompositions and can serve as a rapid prototyping tool for such algorithms. Since we initially proposed D-FLAT, we have made major changes to the system, improving its range of applicability and its usability. In this report, we give a comprehensive overview of our software and provide many examples to illustrate its versatility.

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

Acknowledgements: This work has been supported by the Austrian Science Fund (FWF) under grant P25607, 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) [16, 31] 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 complex problems often 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 fact that, in reality, 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., [14] for an overview). This is particularly attractive because it allows to decompose any problem for which a graph representation can be found. Even more important, via Courcelle's famous result [20] it is nowadays 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, roughly, 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 runtime 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 [1, 35, 37, 38, 44, 50, 51] 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. Since ASP is well suited for a lot of problems, it

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

is often also well suited for parts of such problems, making it an appealing candidate to work 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 a great number of problems.

D-FLAT is free software written in C++ and internally uses the answer set solving systems *Gringo* and *Clasp* [30], as well as the *htdecomp* library for heuristically generating a tree decomposition of the input [23].

Since we initially proposed D-FLAT in [9], we have made major changes to the system. Most importantly, we have generalized the system in such a way that it can now solve any problem expressible in monadic second-order logic, which significantly extends its range of applicability [10]. We have used the tool for implementing decomposition-based algorithms for various problems from diverse application areas, which demonstrates the usability of the method. Furthermore, for assisting users of D-FLAT, we have developed a debugging tool that allows one to inspect the intermediate results of decomposition-based algorithms implemented in D-FLAT.

This work 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 1.0.0 of D-FLAT and describe its components in detail. Subsequently, we turn to practical applications in Section 4, where we present D-FLAT encodings for several problems. In Section 5 we focus on the D-FLAT Debugger and introduce its visualization and debugging capabilities. Finally, we give a conclusion and an outlook in Section 6.

2 Background

This section describes the underlying concepts of the D-FLAT system. We largely follow the presentation in [8]. 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 [5, 31, 39]. *Answer Set Programming* (ASP) denotes a 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 [16, 40]. In [46, 43], 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.

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*

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

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

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

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 [32]. 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* [30].

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 [21]. We will now mention results that are especially important for our purposes.

The unrestricted use of function symbols leads to undecidability in general [17, 2]. 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 [42].

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 [49], and also every search problem in NP can be expressed by a uniform ASP program [41]. 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 [33]) yields higher expressiveness at the cost of Σ_2^{P}-completeness for the problem of deciding answer set existence for ground programs [25, 26].

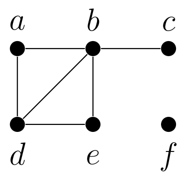


Figure 1 A 3-colorable graph

```
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 [30, 39] 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 [29, 28] 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 propositional satisfiability problem (SAT). An instance in CNF can be given by facts using the predicates atom/1 and clause/1, as well as pos(C, A) and neg(C, A), denoting that the atom A occurs positively or, respectively, negatively in the clause C. The following ASP program then solves the SAT problem.

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

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

2.2 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 [47]. 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 [13, 11, 4, 45].

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.)

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

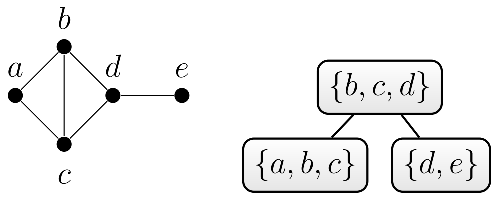


Figure 2 A graph with treewidth 2 and an (optimal) tree decomposition for it

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., [50, 1, 35, 37, 38, 44]). We have taken a look at some publicly available datasets[2] (from domains like co-authorship among scientists, electric power networks or biological networks) and found that indeed often the treewidth is quite small. This appears to be very promising, since it indicates that the 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 [3]. 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 [12]. 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.

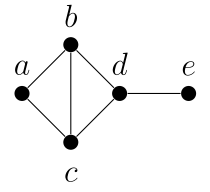
If no such bound on the treewidth can be given a priori, which is the case if we want to be able to process problems even if their treewidth is large, we are not necessarily doomed. Although finding an optimal tree decomposition is intractable in this case, there are efficient heuristics that produce a reasonably good tree decomposition [15, 23, 34]. 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

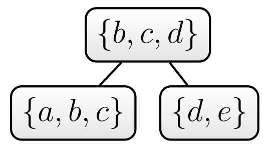
Figure 3 shows how dynamic programming can be applied to a tree decomposition of a GRAPH COLORING instance. Each of the tree decomposition nodes in Figure 3b has a corresponding table in Figure 3c where there is a column for each bag element. Additionally, we have a column i that is used to store an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

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

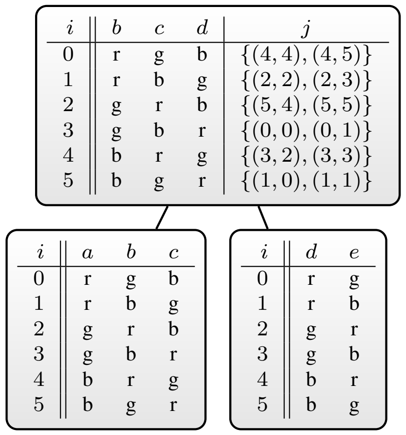
[2]See, for instance, `http://wiki.gephi.org/index.php/Datasets`



(a) A GRAPH COLORING instance (colors are "r", "g" and "b")



(b) A tree decomposition of the instance



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

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

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

(c) The computed dynamic programming tables

Figure 3 Dynamic programming for GRAPH COLORING on a tree decomposition

violate a constraint, i.e., where adjacent vertices have the same color. For instance, the combination of row 0 from the left child with row 2 from the right child is invalid because the adjacent vertices b and d are colored with "g"; the left row 0 combined with the right row 0 is valid, however, and gives rise to the row 3 in the root table. We store the identifiers of these child rows as a pair in the j column. Note that the entry of j in row 3 not only contains $(0,0)$ but also $(0,1)$ because joining these rows produces the same row as we project onto the current bag elements b, c and d. Storing *all* predecessors of a row like this allows us to enumerate *all* 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.

3 The D-FLAT System

This section first gives an overview of the D-FLAT system and then describes its components in more detail. The system is free software and can be downloaded at `http://dbai.tuwien.ac.at/research/project/dflat/system/`. Some examples of D-FLAT encodings can be found in Section 4.

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.

[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.

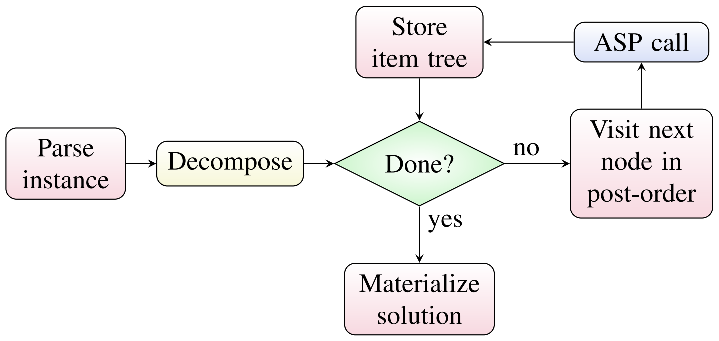


Figure 4 Control flow in D-FLAT

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 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 [10] 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 4 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. 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*

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

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.*

Once a hypergraph representation of the input has been built, the framework uses an external library for heuristically constructing a tree decomposition of small width.[6] This library relies on a bucket elimination algorithm [22] 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.

Maximum Cardinality Search The first vertex is chosen randomly. Afterwards, the vertex having the greatest number of already selected neighbors is selected. This step is repeated 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.

[6]The library currently used is called SHARP which internally uses a software called htdecomp [23].

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) [18]. 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 5 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 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.

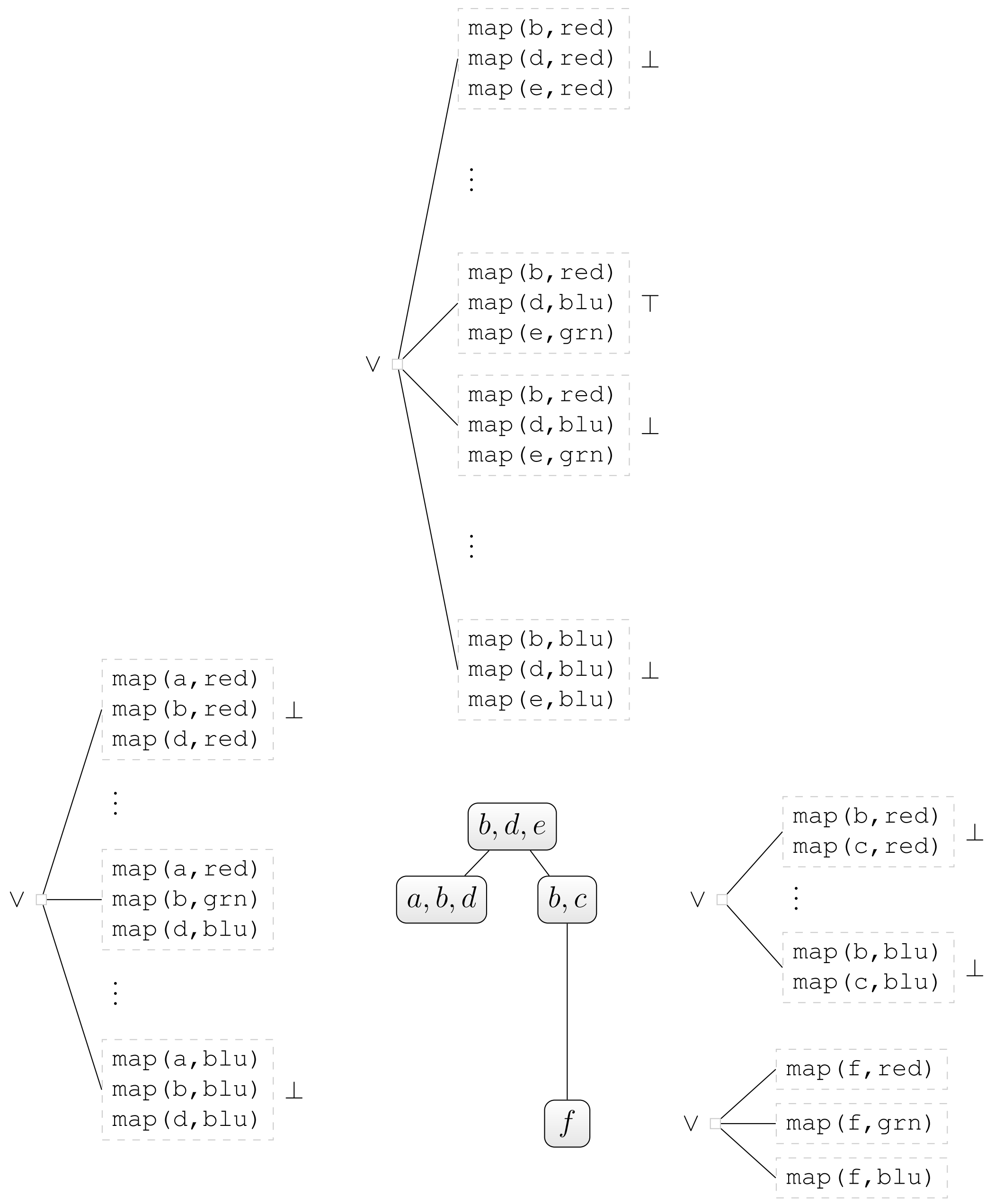


Figure 5 A tree decomposition with item trees for GRAPH COLORING (not showing extension pointers)

Example 3. *Consider Figure 5 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 5 all have roots of type "or", denoted by the symbol $\vee$. This is because an ATM for deciding graph corolability 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 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 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

[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.

nodes whose type has still been undefined, whereas the leaf marked with $\perp$ 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 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.

[8]Note that this means that the input in each ASP solver invocation has always at least linear size in the current implementation of D-FLAT. For really obtaining fixed-parameter linear (instead of quadratic) algorithms, however, the input of each call must not depend on the problem instance size. It is left for future work to only supply the bag-specific part of the problem instance to the ASP solver. We suspect that this is, however, only a minor improvement in practice, as we have observed that the main performance bottlenecks lie elsewhere.

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 5.

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

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.
`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.
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

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

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

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.
- 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.

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.*

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.*

[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.

Input predicate	Meaning
`childRow`(R, N)	R is a table row belonging to decomposition node N.
`childItem`(R, I)	The item set of table row R contains I.
`childAuxItem`(R, I)	The auxiliary item set (for the default join) of table row R contains I.
`childCost`(R, C)	C is the cost value corresponding to the table row R.

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.

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

```
%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

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.

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, 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, 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 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 5. 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.*

[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.

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, 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 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 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 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 (see 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 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 are equal to those of the selected branches into the new item tree. In case costs have been specified, D-FLAT first sets the cost of this new branch to the sum of costs of the selected predecessors, but – similar to the inclusion-exclusion principle – costs that have been counted multiple times must be subtracted. This is why D-FLAT recognizes the output predicate `currentCost`/1, which is used for quantifying the part of the cost (specified by `cost`/1) 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 value from the sum of the predecessor costs.[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) 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.*

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 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.

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.

3.8 Command-Line Usage

In this section we present how D-FLAT can be used from the command line. We first describe the most important command-line options. For a complete list of options along with descriptions, the

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

`-h` option can be passed to D-FLAT for displaying a help message.

General options

- `-h`: Print usage information and exit.
- `--depth <d>`: Print only item sets of depth at most `<d>`.
  (This is explained in Section 3.6.)
- `-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.
- `--no-pruning`: Prune rejecting subtrees only in the decomposition root (for instance for debugging).
- `--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 for the debugging tool described in Section 5.

Options for Constructing a Tree Decomposition

- `-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.

  The different kinds of normalization are described in Section 3.2.
- `--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.
  - `mcs`: Maximum cardinality search.

  These heuristics 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.

Options for the ASP Solver Invocation

- `-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.

- `--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.

- `--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.

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), semi-normalized 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.

```
%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 demonstrate the applicability of D-FLAT to a wide variety of computational problems. Table 6 shows a non-exhaustive listing of problems (grouped by their domain) encoded so far with D-FLAT.[13] For each problem, the computational complexity[14] is given. Furthermore, we give the maximum height of the item tree that we used for solving the problem. The column *Normalization* gives the degree of normalization of the tree decompositions (*Normalized*, *Semi*, *Weak* or *None*) for which the encoding is designed. Finally – if available – a reference to an FPT algorithm for the problem is given. In case the algorithm is presented in this report, a reference to the section is given in the respective column.

In Section 4.1 some representative problems in NP are described in detail. The same is done in Section 4.2 for problems harder than NP. For each problem, a D-FLAT encoding is listed and explained. 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.

4.1 A Selection of Problems in NP

In this section, we present D-FLAT encodings for selected problems in NP.

4.1.1 Boolean Satisfiability

The Boolean Satisfiability problem (SAT) is the prototypical NP-complete problem and is defined as follows.

Input:	A formula ϕ in CNF.
Task:	Compute all models of ϕ.

ASP allows a very succinct modeling of the problem and also formulating it for D-FLAT yields a very simple encoding. In order to solve the problem with D-FLAT, we construct the *incidence graph* of ϕ: Each atom A and each clause C is represented by a vertex of the graph, denoted by facts `atom`(A) and `clause`(C) respectively. An edge between an atom and a clause means that the atom appears positively or negatively in the clause. A positive (negative) occurrence of atom A in clause C is represented by the fact `pos`(C, A) (`neg`(C, A)).

The encoding (see Listing 4) implements a simple *Guess & Check* approach, guessing the truth values for the atoms introduced to the current bag (line 13), and checking if the clauses that are removed from the bag are satisfied (line 11). Note that due to the connectedness condition of tree

[13]A package containing the respective encodings is available at `http://dbai.tuwien.ac.at/proj/dflat/system/examples/encodings-dbai-tr-2014-86.tar.gz`.

[14]Note that the complexity illustrated in the table not necessarily coincides with the complexity of the respective decision problem (i.e., asking whether a solution exists). For instance, deciding if a subset-minimal model for a formula exists is clearly not harder than deciding whether the formula is satisfiable, but reasoning over subset-minimal models can raise the complexity to Σ_2^P.

Problem	**Complexity**	**Levels**	**Normalization**	**Ref.**
Logic				
SAT	NP-c.	1	None	S. 4.1.1
⊆-MINIMAL SAT	Σ_2^P-c.	2	None	S. 4.2.1
QSAT	PSPACE-c.	unbounded	None	-
MSO MODEL CHECKING	PSPACE-c.	unbounded	Semi	[10]
Graph Theory				
GRAPH COLORING	NP-c.	1	None	S. 3.4.1
LIST COLORING	NP-c.	1	None	[27]
MINIMUM VERTEX COVER	NP-hard	1	None	-
CONNECTED VERTEX COVER	NP-c.	1	Weak	-
MINIMUM DOMINATING SET	NP-hard	1	Weak	S. 4.1.2
CONNECTED DOMINATING SET	NP-c.	1	Weak	S. 4.1.3
⊆-MINIMAL DOMINATING SET	Σ_2^P-c.	2	Weak	S. 4.2.2
PERFECT DOMINATING SET	NP-c.	1	Semi	-
STEINER TREE	NP-c.	1	Weak	-
INDEPENDENT SET	NP-c.	1	Semi	-
HAMILTONIAN CYCLE	NP-c.	1	Semi	-
FEEDBACK VERTEX SET	NP-c.	1	Semi	-
ODD CYCLE TRANSVERSAL	NP-c.	1	Semi	-
MAX-CUT	NP-c.	1	Weak	-
DAG PARTITIONING	NP-hard	1	Normalized	[7]
DISJOINT PATHS	NP-c.	1	Semi	[48]
MIN-DEGREE DELETION	NP-hard	1	None	[6]
Bioinformatics				
MIC DETECTION	D^P-c.	3	Semi	-
METAB. NETWORK COMPLET.	?	2	Semi	-
Argumentation				
ADMISSIBLE SETS	NP-c.	1	None	[19, 24, 36]
STABLE EXTENSIONS	NP-c.	1	None	[19, 36]
COMPLETE EXTENSIONS	NP-c.	1	None	[19, 36]
PREFERRED EXTENSIONS	Π_2^P-c.	2	None	[19, 24, 36]

Figure 6 The (non-exhaustive) list of problems encoded for D-FLAT by the authors

```
1  % Make explicit that a row interprets an atom as false or a clause as unsatisfied.
2  false(R,A) ← childRow(R,N), bag(N,A), atom(A), not childItem(R,A).
3  unsat(R,C) ← childRow(R,N), bag(N,C), clause(C), not childAuxItem(R,C).
4  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
5  % Only join rows that coincide on common atoms.
6  ← extend(R1;R2), atom(A), childItem(R1,A), false(R2,A).
7  % True atoms and satisfied clauses remain so unless removed.
8  item(A) ← extend(R), childItem(R,A), current(A).
9  auxItem(C) ← extend(R), childAuxItem(R,C), current(C).
10 % A child row cannot be extended if it contains an unsatisfied clause that is removed.
11 ← clause(C), removed(C), extend(R), unsat(R,C).
12 % Guess truth value of introduced atoms.
13 { item(A) : atom(A), introduced(A) }.
14 % Through the guess, clauses may become satisfied.
15 auxItem(C) ← current(C;A), pos(C,A), item(A).
16 auxItem(C) ← current(C;A), neg(C,A), not item(A).
```

Listing 4   D-FLAT encoding for the SAT problem

decompositions, a removed clause can never occur again in the following nodes' bags and thus can indeed not be satisfied.

Observe that in this encoding only item trees of height 1 are needed, which allows for the use of the table-based D-FLAT interface. As described in Section 3.4.2, the input predicates `childNode`/1, `childRow`/2 specify the child nodes in the decomposition and the rows in their tables, and `current`/1, `introduced`/1 and `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 `item`/1 (and `auxItem`/1) defines the (auxiliary) item set of the current row, and `extend`/1 determines for each child node the row to be extended by the current row. The additional predicate `false/2` denotes the atoms that got assigned the truth value "false". Line 6 then assures that only those rows are joined which agree on the truth assignments of common atoms. Furthermore, `unsat/2` is used to make explicit that a clause was not satisfied in a child row.

Note that due to the separation of item set (containing only the atoms that are set to true) and auxiliary item set (containing the satisfied clauses), the default join (cf. Section 3.7) can be applied to this encoding. In this case the join constraint in line 6 will never fire as the join is done directly in D-FLAT.

```
 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)), introduced(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 5 D-FLAT encoding for the MINIMUM DOMINATING SET problem

4.1.2 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 $u \in X$.

The encoding (see Listing 5) represents one possibility of how to encode the optimization variant of DOMINATING SET in D-FLAT. Note that this encoding only requires item trees of height 1. Again, the code defines the recipe of how the information has to be handled in a node of the tree decomposition. Similar to the previous encoding, 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 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 `numChildNodes` $- 1$ times the current costs due to the inclusion-exclusion principle. Here, `numChildNodes` is a constant provided as input by D-FLAT (cf. Section 3.4). 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.

4.1.3 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 6. Observe that lines 1–16 are the same as in the encoding for MINIMUM DOMINATING SET. We additionally use the function symbols `stop`/0, `con`/2 and `est`/2. For any pair of selected vertices, `con`/2 is derived if the vertices are connected (lines 18–20), while predicate `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 `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).

```
% Make explicit that edges are undirected.
edge(X,Y) ← current(X;Y), edge(Y,X).
% Make explicit that a vertex is not selected in a row.
out(R,X) ← childRow(R,N), bag(N,X), not childItem(R,sel(X)).
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
% Only join rows that coincide on selected vertices.
← extend(R1;R2), childItem(R1,sel(X)), out(R2,X).
% Retain relevant information.
item(sel(X)) ← current(X), extend(R), childItem(R,sel(X)).
item(dom(X)) ← current(X), extend(R), childItem(R,dom(X)).
% Ensure that removed vertices are selected or dominated.
← removed(X), extend(R), not childItem(R,sel(X)), not childItem(R,dom(X)).
% Guess selected vertices.
{ item(sel(X)) : introduced(X) }.
% A vertex is dominated if it is adjacent to a selected vertex.
item(dom(Y)) ← item(sel(X)), edge(X,Y), current(X;Y).
% Connectedness.
item(con(X,Y)) ← current(X;Y), extend(R), childItem(R,con(X,Y)).
item(con(X,Y)) ← current(X;Y), item(sel(X;Y)), edge(X,Y).
item(con(X,Z)) ← current(X;Y;Z), item(con(X,Y)), item(con(Y,Z)).
% If no connection between two selected vertices exists, it has to be established later.
item(est(X,Y)) ← current(X;Y), X ≠ Y, item(sel(X;Y)), not item(con(X,Y)).
item(est(X,Y)) ← current(X;Y), extend(R),
     childItem(R,est(X,Y)), not item(con(X,Y)).
← removed(X), extend(R), childItem(R,est(X,Y)),
     not childItem(R,con(X,Z)) : current(Z).
% 'stop' is used to track and avoid getting several isolated components.
item(stop) ← extend(R), childItem(R,stop).
ok(X) ← removed(X), current(Y), extend(R), childItem(R,con(X,Y)).
item(stop) ← removed(X), extend(R), childItem(R,sel(X)), not ok(X).
← current(X), item(stop), introduced(X).
← extend(R1;R2), childItem(R1,stop), childItem(R2,stop), R1 < R2.
```
Listing 6 D-FLAT encoding for the Connected Dominating Set problem

4.2 A Selection of Problems beyond NP

In this section, we present D-FLAT encodings for selected problems from the polynomial hierarchy beyond NP.

4.2.1 Subset-Minimal Boolean Satisfiability

$\subseteq$-MINIMAL SAT is an extension of the Boolean Satisfiability problem where the aim is not only to decide if the given set of clauses is satisfiable (and to enumerate the respective models) but to compute only the subset-minimal models of it.

Input:	A formula ϕ in CNF.
Task:	Compute all subset-minimal models of ϕ.

The graph is constructed the same way as for the SAT problem. Also the basic idea of guessing a truth assignment for the variables and checking if the clauses are satisfied remains the same. However, to ensure subset-minimality we have to additionally guess all possible subsets of this truth assignment and verify that those do *not* satisfy the clauses. This leads to the increased complexity which is reflected in D-FLAT by the necessity of adding an additional level to the item trees. Listing 7 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.

The use of item tree branches of length greater than 1 requires slight modifications to the original encoding for SAT. 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 we want to compute satisfying truth assignments at both levels, we could simply copy most of the rules from the encoding of SAT 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 order to have only subsets on the second level we rule out every candidate containing an element on the second level that is not on the first level (line 20). The item `smaller` is stored on the second level if the solution candidate on the second level is a strict subset of the assignment of the first level (lines 25–26). Again, we have to eliminate all solution candidates constituting no satisfying truth assignment and, in addition, we now have to take care of subset-minimality. This can be done very easily with D-FLAT. In the final node of the decomposition we simply reject all candidates where a satisfying truth assignment remains on the second level that is a strict subset of the guess on the first level (line 28). Other branches of the item tree are accepted (line 29). As the

```
length(2).
or(0).
and(1).
% Make explicit that an item set interprets an atom as false or a clause as unsatisfied.
false(S,A) ← atNode(S,N), not rootOf(S,N), bag(N,A), atom(A), not
    childItem(S,A).
unsat(S,C) ← atNode(S,N), not rootOf(S,N), bag(N,C), clause(C), not
    childAuxItem(S,C).
% Guess item tree nodes to extend.
1 { extend(0,R) : rootOf(R,N) } 1 ← childNode(N).
1 { extend(L+1,S) : sub(R,S) } 1 ← extend(L,R), L < 2.
% Only join item tree nodes that coincide on common atoms.
← extend(L,X), extend(L,Y), atom(A), childItem(X,A), false(Y,A), L=1..2.
% True atoms and satisfied clauses remain so unless removed.
item(L,A) ← extend(L,S), childItem(S,A), current(A), L=1..2.
auxItem(L,C) ← extend(L,S), childAuxItem(S,C), current(C), L=1..2.
% A child item tree node cannot be extended if it contains an unsat. clause that is removed.
← clause(C), removed(C), extend(L,R), unsat(R,C), L=1..2.
% Guess truth value for introduced atoms.
{item(L,A) : atom(A), introduced(A), L=1..2}.
% Remove (partial) solutions where in level 2 a superset of level 1 has been guessed.
← atom(A), item(2,A), not item(1,A).
% Through the guess, clauses may become satisfied.
auxItem(L,C) ← current(A;C), pos(C,A), item(L,A), L=1..2.
auxItem(L,C) ← current(A;C), neg(C,A), not item(L,A), L=1..2.
% Store if a proper subset has been guessed for the second level.
item(2,smaller) ← extend(2,S), childItem(S,smaller).
item(2,smaller) ← introduced(A), atom(A), item(1,A), not item(2,A).
% Ensure subset-minimality.
reject ← final, item(2,smaller).
accept ← final, not reject.
```

Listing 7 D-FLAT encoding for the ⊆-MINIMAL SAT problem

second level is universally quantified, one rejected branch suffices to remove the solution candidate of the branch's first level. Hence, only subset-minimal models of the formula remain.

4.2.2 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 $u \in X$.

```
length(2).
or(0).
and(1).
% Make explicit that edges are undirected.
edge(X,Y) ← current(X;Y), edge(Y,X).
% Make explicit that a vertex is not selected in an item set.
out(S,X) ← childNode(N), bag(N,X), atNode(S,N), not rootOf(S,N), not
      childItem(S,sel(X)).
% Guess item tree nodes to extend.
1 { extend(0,R) : rootOf(R,N) } 1 ← childNode(N).
1 { extend(L+1,S) : sub(R,S) } 1 ← extend(L,R), L < 2.
% Only join item tree nodes that coincide on selected vertices.
← extend(L,R1), extend(L,R2), childItem(R1,sel(X)), out(R2,A), L=1..2.
% Retain relevant information.
item(L,sel(X)) ← extend(L,R), childItem(R,sel(X)), current(X), L=1..2.
item(L,dom(X)) ← extend(L,R), childItem(R,dom(X)), current(X), L=1..2.
% Ensure that removed vertices are selected or dominated.
← removed(X), extend(L,R), not childItem(R,sel(X)), not childItem(R,dom(X)),
      L=1..2.
% Guess selected vertices, st. set on level 2 is a subset.
{ item(1,sel(X)) : introduced(X) }.
{ item(2,sel(X)) } ← introduced(X), item(1,sel(X)).
% A vertex is dominated if it is adjacent to a selected vertex.
item(L,dom(Y)) ← item(L,sel(X)), edge(X,Y), current(X;Y), L=1..2.
% Store if a proper subset has been guessed for the second level.
item(2,smaller) ← extend(2,S), childItem(S,smaller).
item(2,smaller) ← item(1,sel(X)), not item(2,sel(X)).
% Ensure subset-minimality.
reject ← final, item(2,smaller).
accept ← final, not reject.
```

Listing 8 D-FLAT encoding for the ⊆-MINIMAL DOMINATING SET problem

Another example of a problem above NP is ⊆-MINIMAL DOMINATING SET. As the problem is located at the second level of the polynomial hierarchy, we can again rely on the capabilities of D-FLAT which allow an easy representation of these levels. Similar to ⊆-MINIMAL SAT, also ⊆-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. As we have seen in the previous encoding, this way we can capture problems in Σ_2^P. Note that the tasks which have to be performed in the two levels are very similar, therefore we again use the abbreviation `L=1..2` in some of the rule bodies.

In detail, the algorithm presented in Listing 8 works as follows. Similar to the encoding for the classical 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.

5 The D-FLAT Debugger

As we have seen, the D-FLAT framework supports rapid prototyping of dynamic programming algorithms on tree decompositions. In particular, the user just provides a declarative specification of the problem at hand together with an input instance, and the framework automatically handles decomposition, ASP solver invocation and printing of solutions.

However, it is sometimes desirable to gain a deeper insight into the course of computation in a run of D-FLAT. A visual representation of the decomposed instance and intermediate results computed during the traversal of the decomposition can help in understanding what is going on in the user-specified algorithm. Furthermore, interactive inspection of item trees and their extension pointers (see Section 3.3) allow for a deeper understanding of how erroneous results come about. This is particularly useful when developing new algorithms, since it can be be quite tedious to find errors in the encodings.

These aspects are tackled in the D-FLAT Debugger, a tool for visualization and inspection of algorithms specified in D-FLAT. It is designed to be an easy-to-use, yet powerful, tool which provides support for writing D-FLAT encodings. From our perspective it turned out that this tool can be quite useful in practice. It can be downloaded at `http://dbai.tuwien.ac.at/research/project/dflat/debugger/`.

In the following we present version 0.15 of the D-FLAT Debugger. It is written in Python 2 and it requires GTK+ $\geq$ 2.24. The goal of the tool is to provide support for finding errors in D-FLAT encodings. To this end it can be seen as a static analysis tool and visualizer; it neither interacts directly with D-FLAT, nor performs changes on the decomposition or intermediate results.

In Section 5.1 we show how the debugger can be invoked. Section 5.2 then presents the main features of the tool. Finally, possible future developments are discussed in Section 5.3.

5.1 Command-Line Usage

In order to use the debugger, D-FLAT has to provide machine-readable debugging information. This is achieved by calling D-FLAT with the parameter `--output machine`. For more details about the parameters of D-FLAT, the reader is referred to Section 3.8. A succinct overview is also given in the help output (`-h`) of D-FLAT. The debugger relies only on this output of D-FLAT and runs otherwise independently. That is, one can just use pipes for the inter-process communication. An example call is given in the following.

```
dflat --output machine -p problem.lp < instance.lp | python DflatDebugger.py
```

Of course, it is also possible to store a specific "dump" and use it in a debugging session later. For this, one would usually make two calls as follows.

```
dflat --output machine -p problem.lp < instance.lp > dump.dbg
python DflatDebugger.py < dump.dbg
```

Such a strategy might be useful if, for instance, the call to D-FLAT takes quite some time or one would like to debug several times with the same tree decomposition. In addition, it is also possible to pass option `--seed` to D-FLAT in order to achieve deterministic behaviour. Since a D-FLAT call may take quite some time, one might come across the need for interrupting it. This is possible by pressing `Ctrl+C`. In this case, the debugger will still visualize the data computed so far. Note that this result might not be a tree, but a forest instead.

5.2 Features

In this section we give an overview of the debugging tool's features. We focus on visualization of item trees, search for item tree nodes with specific contents, and the representation of different item tree node types.

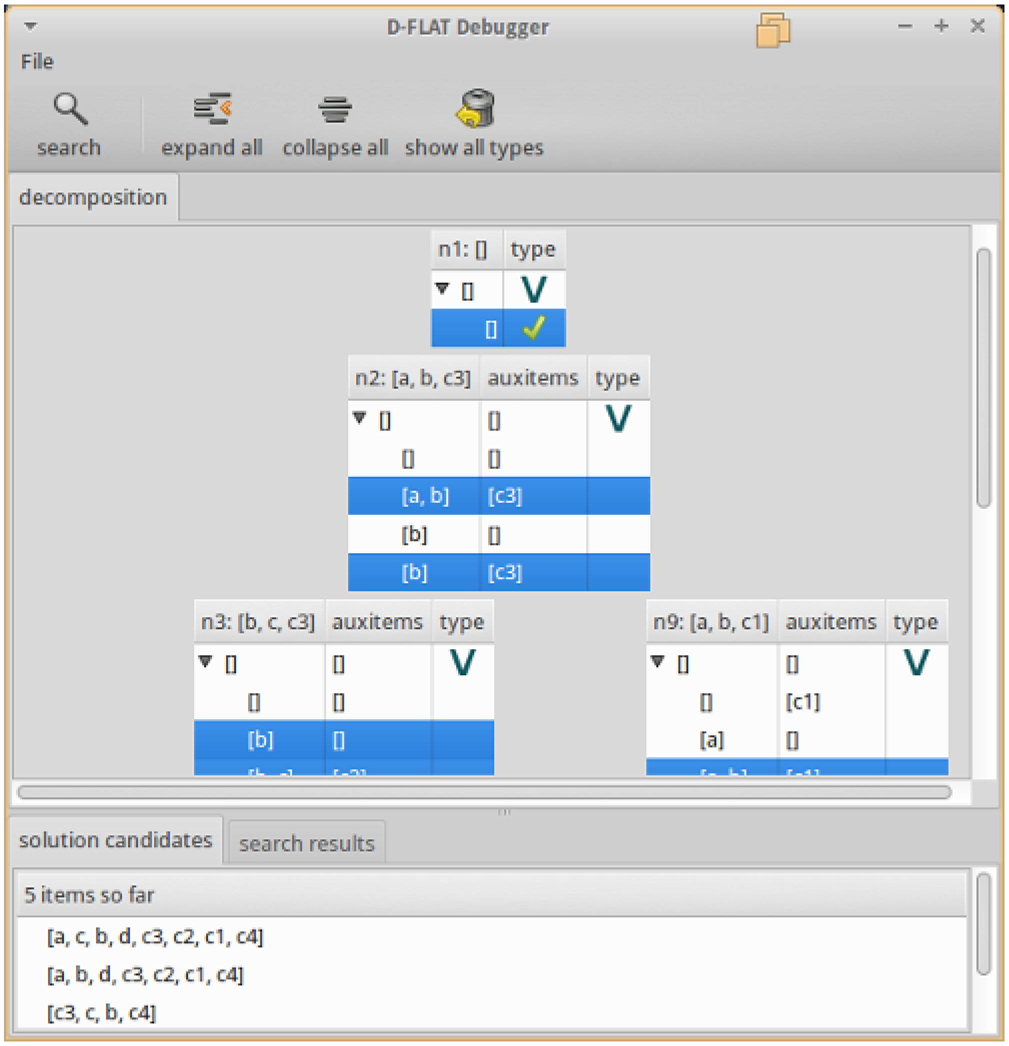
5.2.1 Visualization

One of the most important features of the debugging tool is visualization. It is a lot easier to find flaws within a D-FLAT encoding by looking at the output of the debugger than at the raw data printed to the console. Figure 7 shows a debugging session for the Boolean Satisfiability problem (cf. Section4.1.1), including representation of the tree decomposition and item trees with item sets, auxiliary item sets and item tree node types.

Tree decomposition The main view of the debugging-tool is labeled by the tab "decomposition", which shows the nodes of the used tree decomposition as boxes (tree view); their IDs and bag contents are displayed in the headline of the boxes. The view supports both visualization of programs using the simplified ASP interface (see Section 3.4.2) and the general item-tree-based interface (see Section 3.4.1). Within a node of the decomposition, one can expand the nodes of the item tree by clicking on the "expand" symbol of the respective node. Thereby, for each item tree node at some level, the corresponding child nodes of the next, i.e. higher, level are expanded. The problem visualized in Figure 7 just uses item tree branches of length 1. Note that the root node of each item tree is also represented, reflecting the D-FLAT-internal data structure of item trees. In the example, the item sets of these root nodes are always empty.

Showing extension pointers In order to visualize extension pointers, the user can mark a specific item tree node by clicking on it. The debugger will then (recursively) mark all nodes that are extended by the selected node. In Figure 7 the leaf of the item tree at the root node of the decomposition has been selected, and the debugger recursively marked all the extended nodes. This feature covers the rather frequently occurring use case of finding out how an item tree node came to be. It is quite useful in combination with the visualization of solution candidates (see below). Additionally, the tool makes all marked nodes visible by expanding the relevant parts of the item trees.

Solution candidates At the bottom of the debugger window there is a pane consisting of the tabs "solution candidates" and "search results". The latter one will be discussed in more detail later on; the former one is quite useful in combination with the extension pointer visualization described above. Since the debugging tool allows one to click on any item tree node for marking all the



n1: []	type
▼ []	V
[]	✓

n2: [a, b, c3]	auxitems	type
▼ []	[]	V
[]	[]	
[a, b]	[c3]	
[b]	[]	
[b]	[c3]	

n3: [b, c, c3]	auxitems	type
▼ []	[]	V
[]	[]	
[b]	[]	

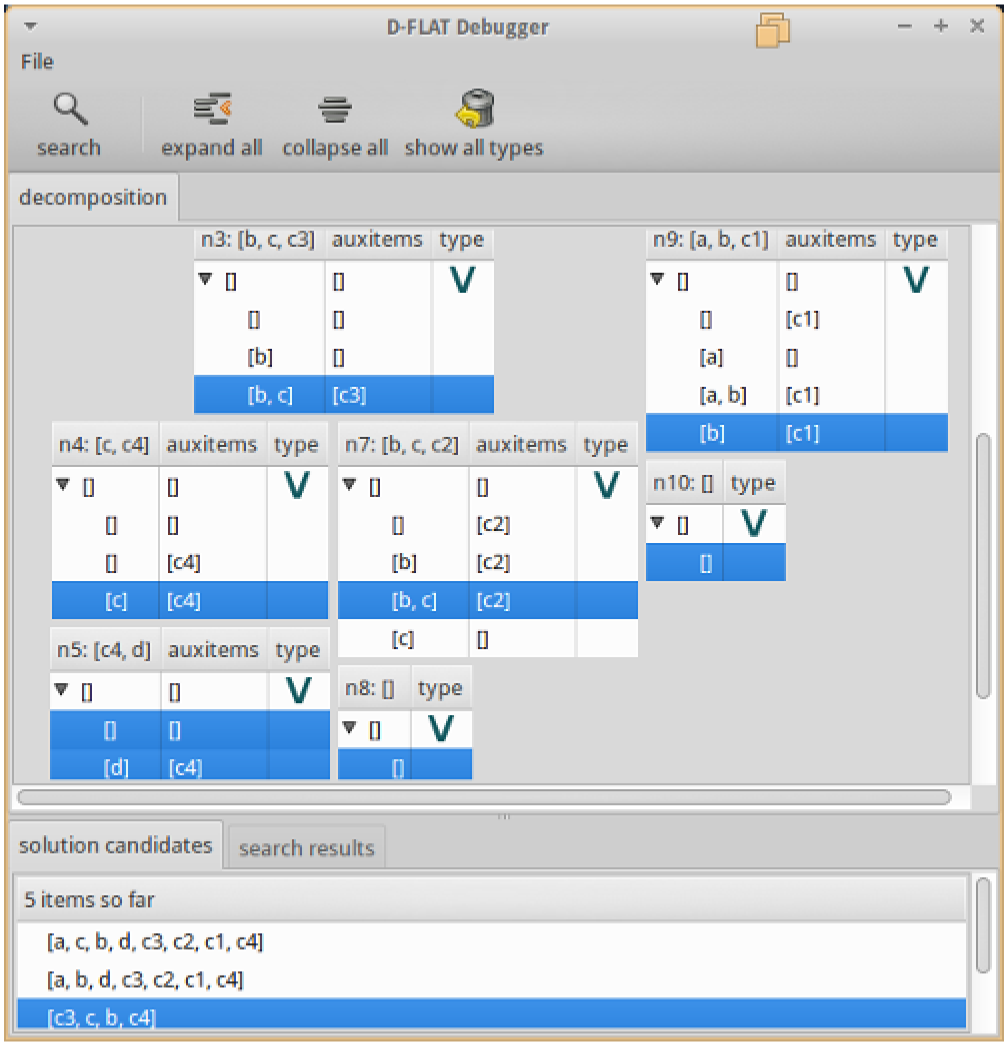
n9: [a, b, c1]	auxitems	type
▼ []	[]	V
[]	[c1]	
[a]	[]	

```
5 items so far
  [a, c, b, d, c3, c2, c1, c4]
  [a, b, d, c3, c2, c1, c4]
  [c3, c, b, c4]
```

Figure 7   Visualization of the decomposition and intermediate results

extended nodes, it is very helpful to also know which solution candidates are represented by the currently selected item tree node. These candidates appear in the first tab of the bottom pane. In Figure 7 the leaf node at the root of the decomposition is selected. This item tree node represents five solution candidates, which are the solutions of the problem instance to be solved in this example. In particular, if the decomposition and the item trees are very large, this feature comes in handy, as it provides the solution candidates at a glance. As seen in Figure 8, the tab "solution candidates" even allows the user to select one specific candidate, for which it then highlights all constituting item tree nodes.

Collapse and expand operations Apart from the already mentioned auto-expand feature when clicking at some specific item tree node, the debugger provides additional expand operations. Global operations are available at the top of the main view, while operations related to a decompo-



n3: [b, c, c3]	auxitems	type
▼ []	[]	V
[]	[]	
[b]	[]	
[b, c]	[c3]	

n9: [a, b, c1]	auxitems	type
▼ []	[]	V
[]	[c1]	
[a]	[]	
[a, b]	[c1]	
[b]	[c1]	

n4: [c, c4]	auxitems	type
▼ []	[]	V
[]	[]	
[]	[c4]	
[c]	[c4]	

n7: [b, c, c2]	auxitems	type
▼ []	[]	V
[]	[c2]	
[b]	[c2]	
[b, c]	[c2]	
[c]	[]	

n10: []	type
▼ []	V
[]	

n5: [c4, d]	auxitems	type
▼ []	[]	V
[]	[]	
[d]	[c4]	

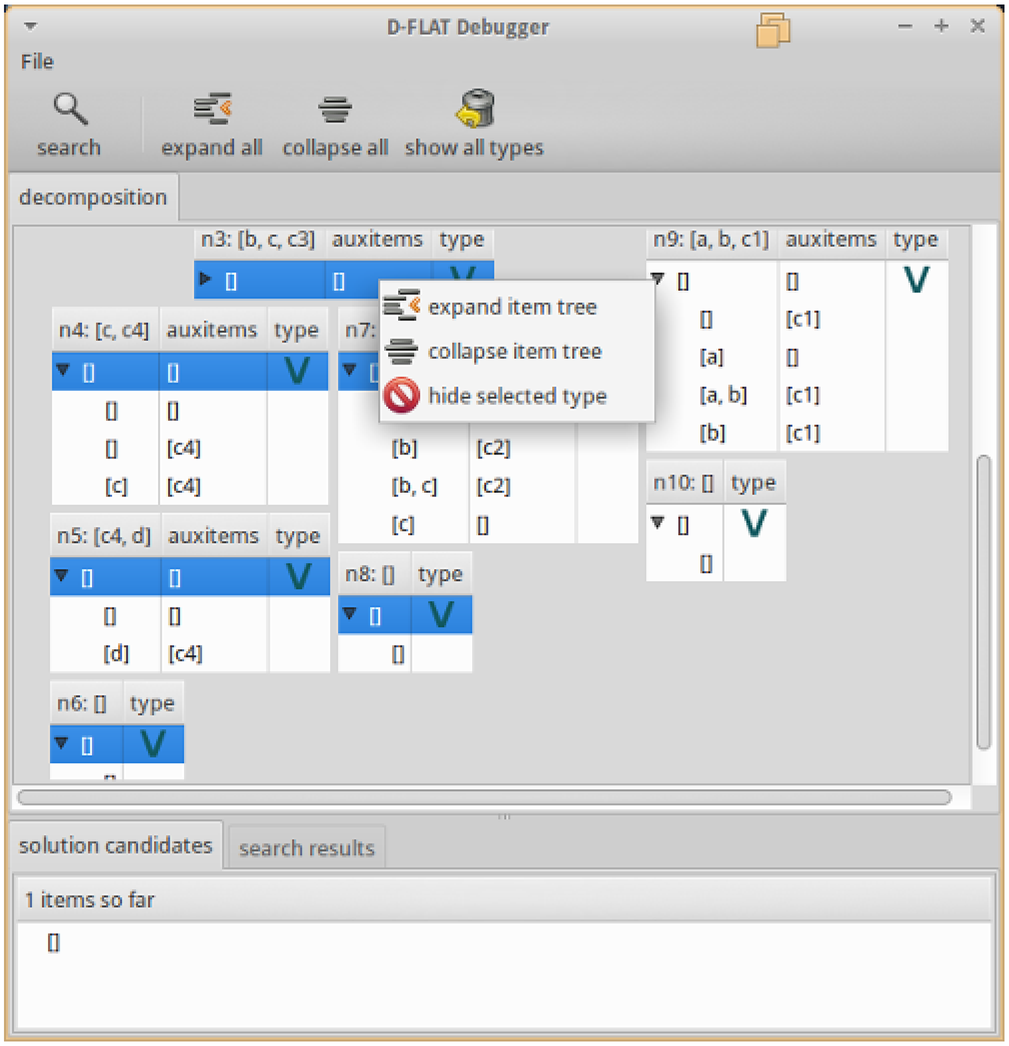
n8: []	type
▼ []	V
[]	

5 items so far
[a, c, b, d, c3, c2, c1, c4]
[a, b, d, c3, c2, c1, c4]
[c3, c, b, c4]

Figure 8 Construction of a solution candidate – A blue item tree node indicates that it was constructed on basis of the blue item tree node(s) in the child nodes of the decomposition

sition node are accessible by right-clicking on the respective node, as shown in Figure 9. There are four collapse and expand operations, which are described below.

- expand all: All item trees are expanded (i.e., all item tree nodes are shown).
- collapse all: All item trees are collapsed (i.e., only the root nodes of item trees are shown).
- expand item tree: The item tree of the currently selected tree decomposition node is expanded.
- collapse item tree: The item tree of the currently selected tree decomposition node is collapsed.



n3: [b, c, c3]	auxitems	type
▸ []	[]	∨

n4: [c, c4]	auxitems	type
▾ []	[]	∨
[]	[]	
[]	[c4]	
[c]	[c4]	

n5: [c4, d]	auxitems	type
▾ []	[]	∨
[]	[]	
[d]	[c4]	

n6: []	type
▾ []	∨

n7:		
▾ [		
[b]	[c2]	
[b, c]	[c2]	
[c]	[]	

n8: []	type
▾ []	∨
[]	

n9: [a, b, c1]	auxitems	type
▾ []	[]	∨
[]	[c1]	
[a]	[]	
[a, b]	[c1]	
[b]	[c1]	

n10: []	type
▾ []	∨
[]	

1 items so far

[]

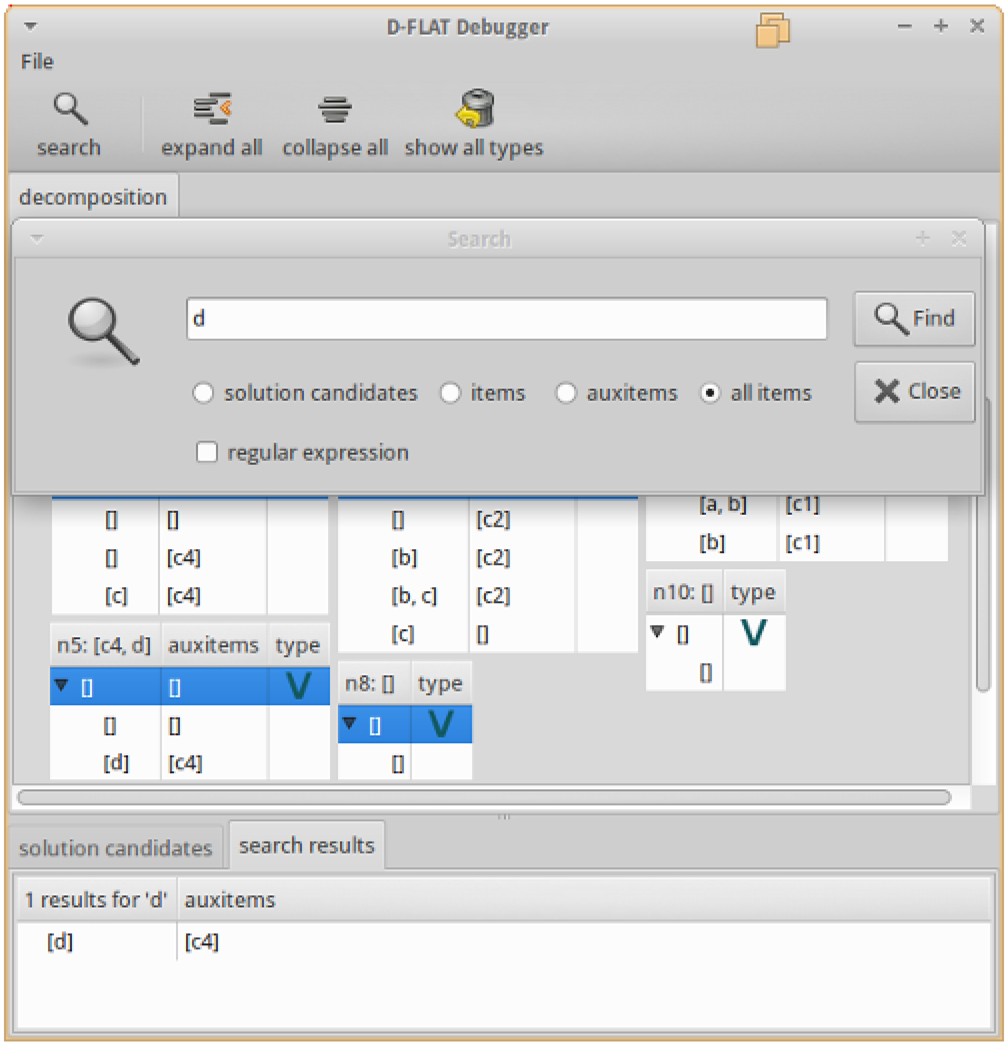
Figure 9 Collapse and expand operations at work

5.2.2 Search

In case there are many solution candidates, items or auxiliary items, the search function might be helpful. It is possible to use regular expressions to search for specific candidates or nodes. As seen in Figure 10, the results are listed in the tab “search results”. When searching for solution candidates, current candidates can be filtered and their corresponding item tree nodes are highlighted. For (auxiliary) items, the respective matching item tree nodes are highlighted. In Figure 11, one can see how regular expressions can be used in order to find specific item sets.

5.2.3 Item Tree Node Types

If the used encoding assigns some type (i.e., “accept”, “reject”, “and” or “or”; cf. Section 3.3.2) to an item tree node, this is represented in the debugger by a corresponding icon (✓, $\times$, $\wedge$ or $\vee$,



Search: d
( ) solution candidates ( ) items ( ) auxitems [✓ all items]
[ ] regular expression

n5: [c4, d]	auxitems	type
▼ []	[]	V
[]	[]	
[d]	[c4]	

n8: []	type
▼ []	V
[]	

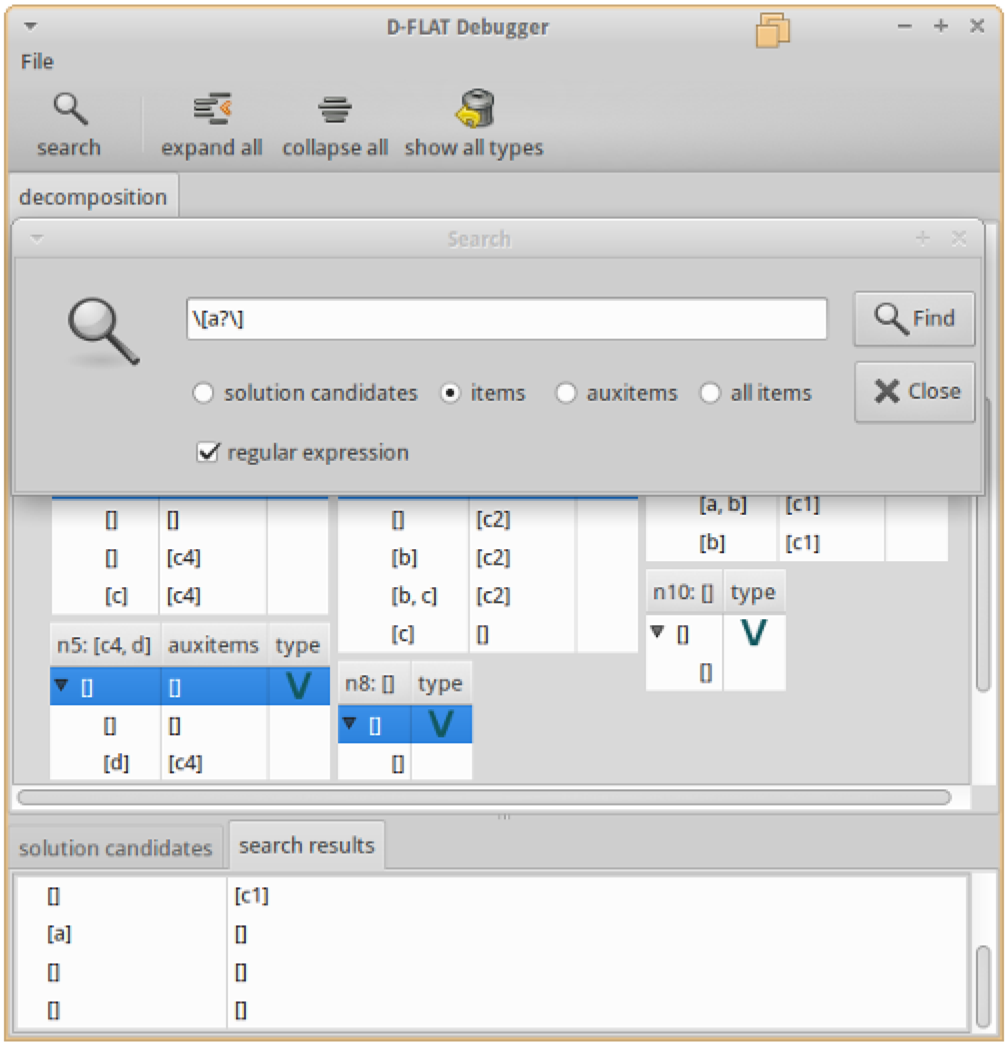
n10: []	type
▼ []	V
[]	

1 results for 'd'	auxitems
[d]	[c4]

Figure 10 Searching for all items

respectively) next to the (auxiliary) items. By selecting some item tree node of a specific type and selecting “hide selected type”, all item tree nodes of the chosen type will become invisible. By choosing “show all types”, this filter gets reverted. For example, if in Figure 12 “hide selected type” were selected, all the item tree nodes of type “accept” would be made invisible.

In particular, this feature is useful for debugging when rejecting item tree nodes are not pruned right away. In D-FLAT this can be achieved by specifying the option `--no-pruning`. Complementing the extension pointer visualization for inspecting how reported solutions came into existence, this feature gives indications at which point during the traversal of the decomposition a solution candidate was *discarded*. Note that this is a common and important use case, since errors in encodings often lead to expected solutions *not* being reported. Furthermore, it is often useful to remove nodes of a particular type from the debugging view in order to keep the visualization compact.



\[a?\]

( ) solution candidates [✓ items] ( ) auxitems ( ) all items

[✓] regular expression

[]	[]
[]	[c4]
[c]	[c4]

n5: [c4, d]	auxitems	type
▼ []	[]	V
[]	[]	
[d]	[c4]	

[]	[c2]
[b]	[c2]
[b, c]	[c2]
[c]	[]

n8: []	type
▼ []	V
[]	

[a, b]	[c1]
[b]	[c1]

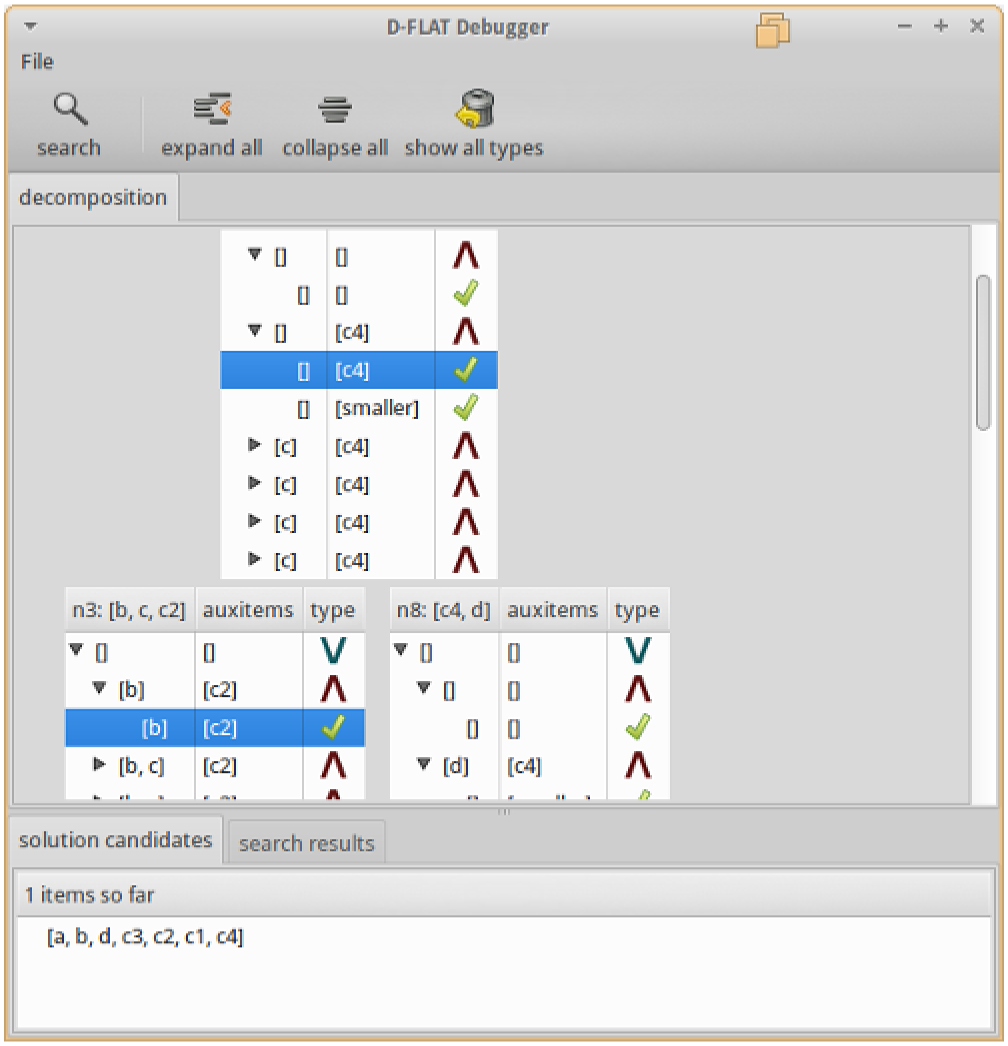
n10: []	type
▼ []	V
[]	

[]	[c1]
[a]	[]
[]	[]
[]	[]

Figure 11 Searching for nodes with regular expressions

5.3 Current Status and Developments

The debugger in its current version supports static analysis of programs. That is, program runs can be retraced but there is no direct interaction with D-FLAT. It gathers debugging information provided by D-FLAT, and uses this to draw an appealing representation of the tree decomposition and item trees. With this, time needed for error detection, and therefore development time, of encodings can be significantly reduced. In a nutshell, the main features of the D-FLAT Debugger are (a) visualization of the tree decomposition and item trees; (b) interactive display of extension pointers to let users easily see where solution candidates originated; (c) diverse collapse and expand operations in order to keep the number of expanded nodes small; (d) search, including support for regular expressions when searching for specific solution candidates or item sets; (e) visualization of (auxiliary) items and item tree node types; (f) identifying reasons why solution candidates are rejected.



▼ []	[]	∧
[]	[]	✓
▼ []	[c4]	∧
[]	[c4]	✓
[]	[smaller]	✓
▶ [c]	[c4]	∧
▶ [c]	[c4]	∧
▶ [c]	[c4]	∧
▶ [c]	[c4]	∧

n3: [b, c, c2]	auxitems	type
▼ []	[]	∨
▼ [b]	[c2]	∧
[b]	[c2]	✓
▶ [b, c]	[c2]	∧

n8: [c4, d]	auxitems	type
▼ []	[]	∨
▼ []	[]	∧
[]	[]	✓
▼ [d]	[c4]	∧

1 items so far

[a, b, d, c3, c2, c1, c4]

Figure 12 Representation of different item tree node types in the D-FLAT Debugger

The debugger can be extended in many promising directions. In order to better cope with large item trees, zooming features and further display options might be helpful. Moreover, dynamic integration with D-FLAT would allow for a “what-would-be-if” analysis, which interacts with D-FLAT and allows to change specific data of some item tree nodes already *during computation*. This would help in understanding the reasons behind unexpected behavior of the ASP encoding. Similar to common debugging tools, it could be helpful to provide the possibility of going through the computation step-by-step. With such a feature we could support breakpoints, which, in this context, can bee seen as special events like the introduction of a specific item tree node.

6 Conclusion

In this report we presented the D-FLAT system in version 1.0.0 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. Then we showed how D-FLAT can be applied to a selection of problems to underline the usability of our approach. Furthermore, we introduced a tool for debugging D-FLAT encodings that visualizes the intermediate results of the user's algorithm and can help to explain the presence of certain undesired results.

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. The proposed ASP interface is general enough to accommodate quite different kinds of problems, and the debugging tool greatly improves usability.

Future work This work shows that D-FLAT has by now reached a level of maturity that allows it to be effectively applied to many problems. In the future, we would like to work on the performance of our approach. In previous performance tests (see, e.g., [9, 36]) we have observed that the overhead of repeatedly calling the ASP system poses a problem if we are not only interested in a rapid prototyping system but also wish to use D-FLAT in competitive settings. Therefore we would like to investigate whether and how this can be improved. Furthermore, we plan to apply D-FLAT to problems from bio-informatics and description logics in order to take advantage of decomposition-based approaches also in these areas.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] Nadja Betzler, Hans L. Bodlaender, Robert Bredereck, Rolf Niedermeier, and Johannes Uhlmann. On making a distinguished vertex of minimum degree by vertex deletion. *Algorithmica*, 68(3):715–738, 2014.

[7] René Bevern, Robert Bredereck, Morgan Chopin, Sepp Hartung, Falk Hüffner, André Nichterlein, and Ondřej Suchý. Parameterized complexity of DAG partitioning. In *Proc. Algorithms and Complexity*, volume 7878 of *LNCS*, pages 49–60. Springer, 2013.

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

[9] Bernhard Bliem, Michael Morak, and Stefan Woltran. D-FLAT: Declarative problem solving using tree decompositions and answer-set programming. *TPLP*, 12(4-5):445–464, 2012.

[10] 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.

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

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

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

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

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

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

[17] 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.

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

[19] Günther Charwat. Tree-decomposition based algorithms for abstract argumentation frameworks. Master's thesis, Vienna University of Technology, 2012.

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

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

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

[23] 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.

[24] Wolfgang Dvořák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37, 2012.

[25] 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.

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

[27] Michael R. Fellows, Fedor V. Fomin, Daniel Lokshtanov, Frances A. Rosamond, Saket Saurabh, Stefan Szeider, and Carsten Thomassen. On the complexity of some colorful problems parameterized by treewidth. *Inf. Comput.*, 209(2):143–153, 2011.

[28] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo. Preliminary Draft. Available at `http://potassco.sourceforge.net`, 2010.

[29] 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.

[30] 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.

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

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

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

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

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

[36] Markus Hecher. Abstract argumentation with D-FLAT: Encodings & experimental results. Bachelor's thesis, Vienna University of Technology, 2013.

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

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

[39] 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.

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

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

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

[43] 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.

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

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

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

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

[48] Petra Scheffler. A practical linear time algorithm for disjoint paths in graphs with bounded tree width. *Fachbereich Mathematik - Report*, 396, 1994.

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

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

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