The D-FLAT System for Dynamic Programming on Tree Decompositions

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

Institute of Information Systems 184/2
Vienna University of Technology
Favoritenstrasse 9–11, 1040 Vienna, Austria
[abseher,bliem,gcharwat,dusberg,hecher,woltran]@dbai.tuwien.ac.at

Abstract. Complex reasoning problems over large amounts of data pose a great challenge for computer science. 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 the logic programming language Answer Set Programming 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 paper, we present the system resulting from these efforts.

Keywords: Answer Set Programming, tree decompositions, treewidth

1 Introduction

Complex reasoning problems over large amounts of data arise in many of today's application domains for computer science. They pose 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 for achieving code that is maintainable and easy to read and write. A more and more popular candidate for such a declarative approach is Answer Set Programming (ASP) [12, 18], for which sophisticated solvers are available that offer high efficiency and rich languages for modeling the problems at hand.

Second, handling computationally complex queries over huge data is an insurmountable obstacle for standard algorithms. One potential solution is to *exploit structure*. A prominent approach to do so is to employ tree decompositions (see, e.g., [10] for an overview). 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 [2, 21–24, 27, 28] 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], first presented in [5], for rapid prototyping of DP algorithms in the declarative language of ASP. Since ASP is well suited for a lot of problems, it is often also well suited for parts of such problems, making it an appealing candidate to work on decomposed problem instances. The key feature of D-FLAT is that the user is only required to write an encoding of the DP algorithm on a tree decomposition in the ASP language, and the system takes care of tedious tasks that are not related to the problem.

The initial prototype of D-FLAT [5] stored partial solutions in tables. It became clear, however, that for problems higher in the polynomial hierarchy than NP a more general data structure is required. We have showed in [6] that using a tree-shaped data structure instead greatly increases applicability.

In this paper we present the D-FLAT system resulting from the major changes since its initial presentation in [5]. Our main contributions are:

1. We introduce *item trees*, which serve as the central data structure in D-FLAT algorithms.
2. We show how item trees allow problems to be solved in the style of Alternating Turing Machines while also taking decomposition into account.
3. We present the special predicates used for communication between the system and the user's encoding.
4. Finally, we show how the system interprets the answer sets of the user's program for constructing item trees and eventually solving the problem.

This work is structured as follows. In Section 2, we provide background on Answer Set Programming, tree decompositions and the original prototype of D-FLAT, as originally presented in [5]. In Section 3, we then present the current version 1.0.0 of D-FLAT and describe its components in detail. Finally, we give a conclusion and an outlook in Section 4.

2 Background

In this section, we first give brief introductions to Answer Set Programming (Section 2.1) and tree decompositions (Section 2.2). Then, in Section 2.3, we

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

mention the main notions behind the D-FLAT system as presented in [5], which we extend in this work.

2.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative language where a *program* Π is a set of *rules*

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n.$$

The constituents of a rule $r \in \Pi$ are $h(r) = \{a_1, \ldots, a_k\}$, $b^+(r) = \{b_1, \ldots, b_m\}$ and $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. We call r a *fact* if $b^+(r) = b^-(r) = \emptyset$, and we omit the $\leftarrow$ symbol in this case. Intuitively, a rule r states that if an answer set contains all of $b^+(r)$ and none of $b^-(r)$, then it contains some element of $h(r)$. A set of atoms I satisfies a rule r if $I \cap h(r) \neq \emptyset$ or $b^-(r) \cap I \neq \emptyset$ or $b^+(r) \setminus I \neq \emptyset$. I is a *model* of a set of rules if it satisfies each rule. I is an *answer set* of a program Π if it is a subset-minimal model of the program $\Pi^I = \{h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$ [19].

ASP programs can be viewed as succinctly representing problem solving specifications following the *Guess & Check* principle. A "guess" can, for example, be performed using disjunctive rules which non-deterministically open up the search space. Constraints (i.e., rules r with $h(r) = \emptyset$), on the other hand, amount to a "check" by imposing restrictions that solutions must obey.

In this paper, we use the language of the grounder *Gringo* [16, 17] (version 4) where programs may contain variables that are instantiated by all ground terms (elements of the Herbrand universe, i.e., constants and compound terms containing function symbols) before a solver computes answer sets according to the propositional semantics stated above.

Example 1. The following program solves the 3-Colorability problem for graphs that are given as facts using the predicates `vertex` and `edge`.

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

Informally, the first rule is shorthand for the three facts `color(red)`, `color(grn)` and `color(blu)`. The second rule states that any vertex from the input graph shall be mapped to one color. The colon controls the instantiation of the variable C such that it is only instantiated with arguments of the predicate `color`. The third rule checks that adjacent vertices never receive the same color. In this rule, `map(X,C;Y,C)` stands for `map(X,C), map(Y,C)`.

2.2 Tree Decompositions

Tree decompositions and treewidth, originally defined in [26], are well known tools for tackling computationally hard problems. Informally, treewidth is a measure of the cyclicity of a graph, and many NP-hard problems become tractable

if the treewidth is bounded. There are several overviews of this topic, such as [9, 7, 4, 25]. The intuition behind tree decompositions is obtaining a tree from a (potentially cyclic) graph by subsuming multiple vertices under one node and thereby isolating the parts responsible for the cyclicity. The definition of tree decompositions can also be extended to hypergraphs, but in this paper we will only consider graphs for the sake of presentation.

Definition 1. *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.*

Figure 1 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. (A graph is a tree iff it has treewidth 1.)

Constructing an optimal tree decomposition is intractable [3]. However, when considering treewidth as a parameter, the problem is fixed-parameter tractable (FPT) [15], i.e., solvable in time $\mathcal{O}(f(w) \cdot n^c)$, where c is a constant, n is the size of the input, w is its treewidth and $f(w)$ depends only on w [8]. Moreover, there are efficient heuristics that produce good tree decompositions [11, 14, 20].

Tree decompositions are prominently used for solving problems with dynamic programming algorithms. These algorithms generally start at the leaf nodes and traverse the tree decomposition to the root. At each node, partial solutions for the subgraph induced by the vertices encountered so far are computed and stored in a data structure corresponding to that tree decomposition node. For computing the partial solutions of a node, the partial solutions of child nodes are usually extended. Typically, the size of each such data structure only depends on the width of the tree decomposition, and the number of tree decomposition nodes is linear in the size of the input graph. Thus, when the width is bounded by a constant, the search space for each subproblem remains constant as well, and the number of subproblems only grows by a linear factor for larger instances.

2.3 The D-FLAT System

D-FLAT [1, 5] is a framework for developing algorithms that solve computational problems by dynamic programming on a tree decomposition of the problem instance. It proceeds in the following way.

1. D-FLAT parses a representation of the problem instance and automatically constructs a tree decomposition of it using heuristic methods.

2. It provides a data structure 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 populate the data structure associated with a tree decomposition node.
3. D-FLAT traverses the tree decomposition in post-order and calls an ASP system at each tree decomposition node for computing the data structure corresponding to that node by means of the user-specified program.
4. The framework automatically combines the partial solutions and prints all complete solutions. Alternatively, it is also possible to solve decision, counting and optimization problems.

The system is free software and can be downloaded at `http://dbai.tuwien.ac.at/research/project/dflat/system/`.

In our presentation of the initial D-FLAT prototype [5] we were able to successfully apply it to several problems, and we showed in [6] which modifications could further extend its applicability. In the current paper we present the new version of D-FLAT that results from these extensions.

3 The Extended D-FLAT System

This section first gives an overview of the D-FLAT system with emphasis on the extensions made since [5] that allow it to solve any problem expressible in monadic second-order logic in FPT time [6]. This includes many problems from NP but also harder problems in PSPACE. A comprehensive and detailed description is found in [1].

3.1 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 graph representation of this input. Along with the facts describing the instance, the user therefore must specify which predicates therein designate the domain and the edge relation.

Once a graph representation of the input has been built, the framework uses heuristics [14] for constructing a tree decomposition of small width.

It is often convenient to presuppose tree decompositions having a certain normal form (such as the commonly encountered "nice" tree decompositions). This usually makes algorithms easier to specify as fewer cases have to be considered. D-FLAT provides several such normalizations. For details, we refer to [1].

3.2 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) [13]. 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.

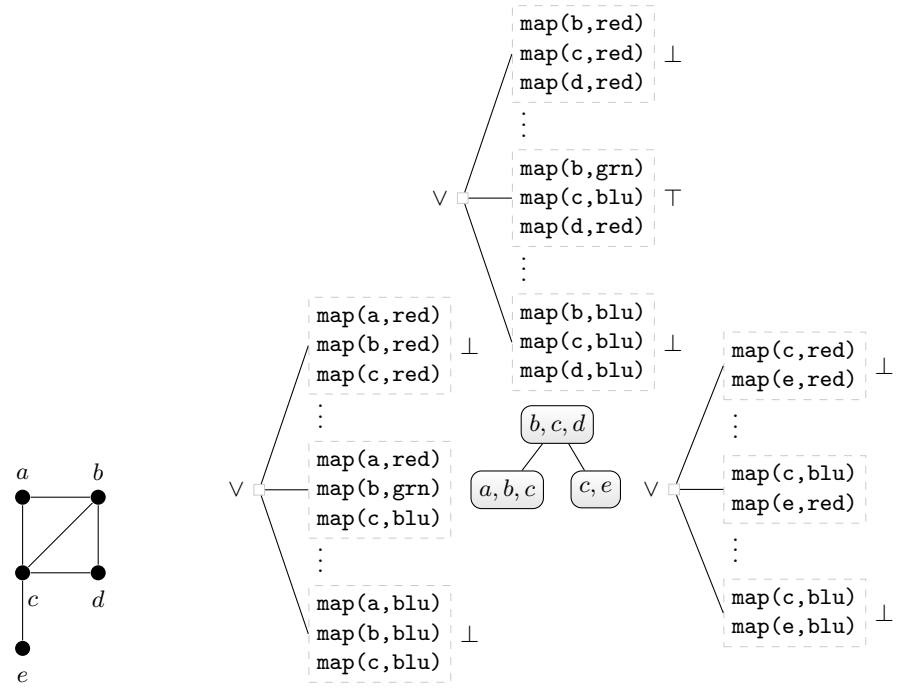
Each branch in an item tree may be associated with a cost value, which allows for optimization problems to be solved. If costs are given, D-FLAT automatically only reports optimal solutions. Details on this are found in [1].

Example 2. Figure 1 shows a graph, one of its tree decompositions and, for each decomposition node, the corresponding item tree that could result from an algorithm for 3-Colorability. 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 later in this section.

Extension Pointers For solving 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 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 $\mathcal{I}_i$, and we say that ν *extends* ν_i.

Example 3. Consider Figure 1b again. In the following examples, let $\mathcal{I}_S$ denote the item tree of the node whose bag is S. In $\mathcal{I}_{\{a,b,c\}}$ and $\mathcal{I}_{\{c,e\}}$, all nodes have



(a) A 3-colorable graph

(b) A tree decomposition of the instance in (a) with item trees for 3-Colorability (not showing extension pointers)

Fig. 1: Item trees for a decomposition of a 3-Colorability instance

the same set of extension pointer tuples: the set consisting of the empty tuple, as those decomposition nodes have no children. The set of extension pointer tuples at the root of $\mathcal{I}_{\{b,c,d\}}$ consists of a single binary tuple – one element references the root of $\mathcal{I}_{\{a,b,c\}}$, the other references the root of $\mathcal{I}_{\{c,e\}}$. For a node ν at depth 1 of $\mathcal{I}_{\{b,c,d\}}$, 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 $\mathcal{I}_{\{a,b,c\}}$ and $\mathcal{I}_{\{c,e\}}$, respectively. Moreover, if an element of the current bag $\{b, c, d\}$ is assigned a color in ν_1 or ν_2, then ν colors it in the same way.

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 depending on the node types. These 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 3.4.

Example 4. The item trees in Figure 1b 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. In $\mathcal{I}_{\{a,b,c\}}$ and $\mathcal{I}_{\{c,e\}}$, 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.

3.3 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. The answer sets of this ASP call specify the item tree that D-FLAT shall construct for the current decomposition node. D-FLAT provides facts about the tree decomposition and child item trees according to Table 1. We have omitted some less frequently used predicates for clarity. A complete list is given in [1].

Each answer set corresponds to a branch in the new item tree. The predicates for specifying this branch are described in Table 2. One should keep in mind, however, that D-FLAT may merge subtrees as described in Section 3.4. Therefore, after merging, one branch in the item tree may comprise information from multiple answer sets.

Example 5. A possible encoding for the 3-Colorability problem is shown in Listing 1.1. We use colors to highlight input and output predicates. Note that it would be more convenient (and faster) to encode this problem using the simplified ASP interface for problems in NP described in [1], which we omit from this paper for space reasons.

Input predicate	Meaning
`final`	The current tree decomposition node is the root.
`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.
`root`(S)	S is the root of an item tree from a child of the current node.
`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.
`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 1: The most commonly used input predicates describing the tree decomposition and 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.
`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.
`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 2: The most commonly used output predicates for constructing the item tree of the current decomposition node

```
1 length(1). or(0).
2 1 { item(1,map(X,red;X,grn;X,blu)) } 1 ← current(X).
3 reject ← edge(X,Y), item(1,map(X,C;Y,C)).
4 extend(0,S) ← root(S).
5 1 { extend(1,S) : sub(R,S) } 1 ← root(R).
6 ← item(1,map(X,C0)), childItem(S,map(X,C1)), extend(_,S),
      C0 ≠ C1.
7 reject ← childReject(S), extend(_,S).
8 accept ← final, not reject.
```

Listing 1.1: D-FLAT encoding for 3-Colorability

Line 1 specifies that each answer set declares a branch of length 1, whose root node has the type "or". Line 2 guesses a color for each current vertex. The "reject" node type is derived in line 3 if this guessed coloring is improper. Lines 4 and 5 guess a branch for each child item tree. Due to line 6, the guessed combination of predecessor branches only leads to an answer set if it does not contradict the coloring guessed in line 2. 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 7. Finally, line 8 derives the "accept" node type if no conflict has occurred.

3.4 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 so-called *uncompressed item tree* of the current node. Subsequently D-FLAT prunes subtrees of that tree that can never be part of a solution 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. The item tree so far is called uncompressed because it may contain redundancies that are eliminated in the final step.

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.

Propagation of Acceptance Statuses and Pruning of Item Trees In Section 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. 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. The acceptance status of nodes that are neither accepting nor rejecting is *undefined*.

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 3.5. 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 entire tree in memory.)

Example 6. Consider again Figure 1b. Because $\mathcal{I}_{\{b,c,d\}}$ 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 $\mathcal{I}_{\{b,c,d\}}$ 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 six leaves, each encoding a proper coloring of the vertices b, c and d, and storing extension pointers that let us extend the respective coloring of these vertices to proper colorings of all the other vertices, too.

Propagation of Optimization Values in Item Trees For optimization problems, after an uncompressed item tree has been computed, an additional step is done. 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 such that the optimization value of a leaf node is its cost, and the optimization value of an "or" or "and" node is the minimum or maximum, respectively, among the optimization values of its children.

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. 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.

For problems in NP, this redundancy elimination can be done on the fly [1].

3.5 Materializing Complete Solutions

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

In the item tree at the root of the decomposition there are only accepting nodes left after the pruning described in Section 3.4. 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 n' with parent n in this way, however, D-FLAT takes care to only pick an extension pointer tuple of n' 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 the optimization value c. This is because, due to D-FLAT's propagation of optimization values (cf. Section 3.4), 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 3.4). Limiting the materialization depth saves us from potentially materializing exponentially many solutions when all we are interested in is 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.

3.6 Debugging Support

As it can be hard to find causes of erroneous results of dynamic programming algorithms on tree decompositions, we have developed a debugging tool that visualizes the generated tree decomposition and the item trees produced by the D-FLAT encoding. It allows the user to inspect how solutions came to be and thus greatly simplifies debugging. A detailed presentation is given in [1].

4 Conclusion

In this paper we have presented the D-FLAT system for solving problems by means of dynamic programming on a tree decomposition of the instance. The key feature is that D-FLAT allows the user to specify the problem-specific computations in the logic programming language of Answer Set Programming, and takes care of the tasks not related to the actual problem.

We have discussed the most significant changes made since the initial publication of D-FLAT in [5]. In particular, these extensions allow D-FLAT to solve any problem expressible in monadic second-order logic [6]. This significantly extends its range of applicability. As reported in [1], this extension of D-FLAT allowed us to apply it to various problems, which demonstrates the usability of the method. Furthermore, we provide a debugging tool that facilitates development of algorithms for D-FLAT.

Future work. In the future, we would like to focus on improving the performance of D-FLAT. There are also plans to apply it in particular to problems from bioinformatics and description logics. Furthermore, we plan to investigate more general notions of decomposition than tree decompositions.

Acknowledgments. This work is supported by the Austrian Science Fund (FWF) projects P25518, P25607 and Y698.

References

1. Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. D-FLAT: Progress report. Technical Report DBAI-TR-2014-86, Vienna University of Technology, 2014.
2. 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.
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, April 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. 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.
6. Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. Declarative dynamic programming as an alternative realization of courcelle's theorem. In Gregory Gutin and Stefan Szeider, editors, *IPEC*, volume 8246 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 2013.
7. Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22, 1993.
8. Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.
9. Hans L. Bodlaender. Discovering treewidth. In *Proc. SOFSEM*, volume 3381 of *LNCS*, pages 1–16. Springer, 2005.
10. Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, 51(3):255–269, 2008.
11. Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.
12. Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
13. Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.
14. 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.
15. Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.
16. 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.
17. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.
18. Michael Gelfond and Nicola Leone. Logic programming and knowledge representation – the A-Prolog perspective. *Artif. Intell.*, 138(1-2):3–38, 2002.

19. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.
20. Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.
21. Jens Gramm, Arfst Nickelsen, and Till Tantau. Fixed-parameter algorithms in phylogenetics. *Comput. J.*, 51(1):79–101, 2008.
22. Xiuzhen Huang and Jing Lai. Parameterized graph problems in computational biology. In *Proc. IMSCCS*, pages 129–132. IEEE, 2007.
23. Matthieu Latapy and Clémence Magnien. Measuring fundamental properties of real-world complex networks. *CoRR*, abs/cs/0609115, 2006.
24. Guy Melançon. Just how dense are dense graphs in the real world? A methodological note. In *Proc. BELIV*, pages 1–7. ACM Press, 2006.
25. Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press, 2006.
26. Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.
27. Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998.
28. Atsuko Yamaguchi, Kiyoko F. Aoki, and Hiroshi Mamitsuka. Graph complexity of chemical compounds in biological pathways. *Genome Inform.*, 14:376–377, 2003.