Documentation for D-FLAT 0.2

Bernhard Bliem
(bliem@dbai.tuwien.ac.at)

March 9, 2013

1 Introduction

Many practically relevant problems are infeasible for large instances. However, often they become tractable when only instances where a certain parameter is bounded by a constant are considered [Downey and Fellows, 1999, Flum and Grohe, 2006, Niedermeier, 2006]. This research area has attracted a lot of attention lately because it allows for a more fine-grained analysis of the complexity of computational problems than classical complexity theory, which only considers the size of the input as the quantity of interest. Especially *treewidth* has proven to be an attractive parameter because it applies to many different problems [Robertson and Seymour, 1984, Courcelle, 1990]. Bounded treewidth leading to tractability can frequently be exploited by *dynamic programming* on a *tree decomposition* of the original instance [Niedermeier, 2006, Bodlaender, 1997]. Until now, implementing such algorithms, however, has usually been quite intricate which is due to the lack of supporting tools that offer an adequate language for conveniently specifying such algorithms.

In [Bliem et al., 2012], we have introduced a framework called D-FLAT that makes rapid prototyping of algorithms on tree decompositions possible. The acronym abbreviates the full name *Dynamic Programming Framework with Local Execution of ASP on Tree Decompositions*. Its purpose is to take care of everything that surrounds dynamic programming algorithms on tree decompositions and to leave the user only with the responsibility of providing the problem-specific parts in *Answer Set Programming* (ASP) – a logic programming formalism which supports a programming paradigm called *Guess & Check* and is recognized for its capability to express hard problems quite succinctly [Brewka et al., 2011, Lifschitz, 2008]. Using Answer Set Programming as a language to specify the problem-specific parts of dynamic programming algorithms, D-FLAT benefits from efficient solvers as well as from a rich language that allows for easily readable and maintainable code.

Because instances in practical applications often exhibit small treewidth [Thorup, 1998, Agarwal et al., 2011, Gramm et al., 2008, Huang and Lai, 2007, Latapy and Magnien, 2006, Melançon, 2006], D-FLAT has great practical relevance. For many problems

that are hard in general, D-FLAT is thus a promising candidate for solving large instances which have so far been out of reach for existing Answer Set Programming systems.

The system can be obtained as free software from `http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/`. It is written in C++ and can be compiled for many platforms. Since we introduced version 0.1 of D-FLAT in [Bliem et al., 2012], we have made several changes resulting in the release of D-FLAT version 0.2. The current report serves as documentation for D-FLAT 0.2 and is based on [Bliem, 2012], where the underlying problem solving methodology is inspected in more detail.

In Section 2, we will introduce the architecture of D-FLAT and describe the interplay of its components. Section 3 gives information about the main data structures that D-FLAT offers to the user for solving problems via dynamic programming on a tree decomposition of the input instance. In Section 4, we outline the role of the user-specified ASP program and present which input facts D-FLAT provides to it. This is complemented by Section 5, where we describe how the answer sets of that program are used to populate the data structures associated with the tree decomposition nodes. The final materialization of solutions is presented in Section 6. Section 7 describes the input format for problem instances, and Section 8 gives details about the tree decompositions that D-FLAT can automatically generate from those. A complete listing of D-FLAT's command-line arguments can be found in Section 9, and Section 10 provides a reference of all input and output predicates that serve as the interface between the user-specified program and D-FLAT. Finally, Section 11 presents a simple example of a D-FLAT encoding by means of the 3-Col problem.

2 System Overview

A key property of D-FLAT is that it associates a *table* with each tree decomposition node. The computation of these tables proceeds in a bottom-up way by executing a user-provided, problem-specific ASP program for each node, having access to the already computed child tables.

We now give an overview of D-FLAT 0.2 by describing its organization and the interplay of its components. Figure 1 depicts the control flow during the execution of an algorithm with D-FLAT and illustrates the interplay of the system's components. The system consists of the following basic elements:

- The D-FLAT core; it coordinates the data flow between all other components and takes care of parsing the input (see Section 7), storing and processing the tables (see Sections 3 and 5), as well as materializing solutions (see Section 6).

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

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

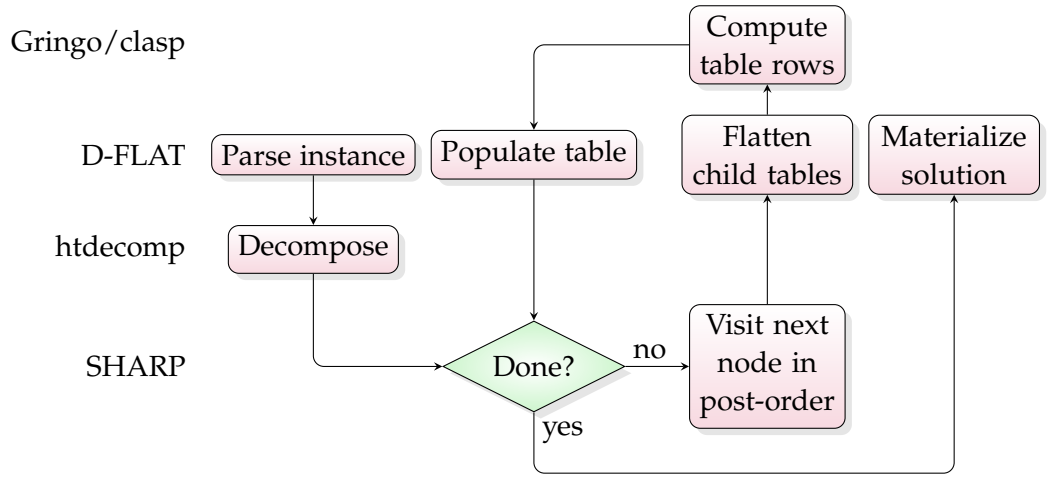


Figure 1 A flowchart illustrating the (simplified) interplay of D-FLAT's components

language for the algorithms.

- An ASP solving system; currently we use *Gringo* for grounding and *clasp* for solving. These programs are part of the Potsdam Answer Set Solving Collection [Gebser et al., 2011] and are currently the generally most efficient ASP solving tools available [Denecker et al., 2009]. For the interface between the user's ASP program and these tools, see Section 10.
- A tree decomposition library; we currently employ *htdecomp* [Dermaku et al., 2008] for this purpose. For details about the decomposition, see Section 8.

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

3 Basic Data Structures: Tables and Rows

Each table row corresponds to a set of partial solutions and consists of a *characteristic* and *additional information*. The remainder of this section will describe the structure of characteristics and give an overview of the additional information stored in table rows. Section 5 then shows how tables can be computed from the answer sets of the user-specified program.

Characteristics of table rows. The information stored in characteristics is comprised of *items* in a way we describe in this section. An item can be any ground term the user pleases. By means of terms containing uninterpreted function symbols, the user can even store structured data in an item.

In the case where we are dealing with problems that are, in general, members of NP, it is usually sufficient to consider a characteristic as just an *item set*, i.e., a set of arbitrary items. With this, all examples presented in [Bliem et al., 2012] can be implemented. However, algorithms on tree decompositions for problems higher in the polynomial hierarchy than NP usually require a more involved table row structure than that, due to the polynomial hierarchy's characterization by quantifier alternation.

In order to allow handling also such problems in a natural way, D-FLAT uses a more general notion of a table row's characteristic: Any item set can possess arbitrarily many *subsidiary item sets*.This can be recursively utilized to obtain a *tree of item sets* within a single table row.

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

Additional information in table rows. A table row contains *additional information* beside its characteristic, depending on the problem type.

- For *decision problems*, no additional information is stored.
- In algorithms for *counting problems*, a table row contains the number of partial solutions it represents.
- When solving *enumeration problems*, each table row r of a node that has n children contains a set of n-tuples of *extension pointers*, where the ith element of any such tuple references a row in the ith child table that has given rise to r. These pointers will be followed when materializing solutions.

  When dealing with multi-level characteristics, extension pointers are usually not only present for enumeration problems but also for other problem types. Moreover, in these cases, *each item set* can possess extension pointers – not merely

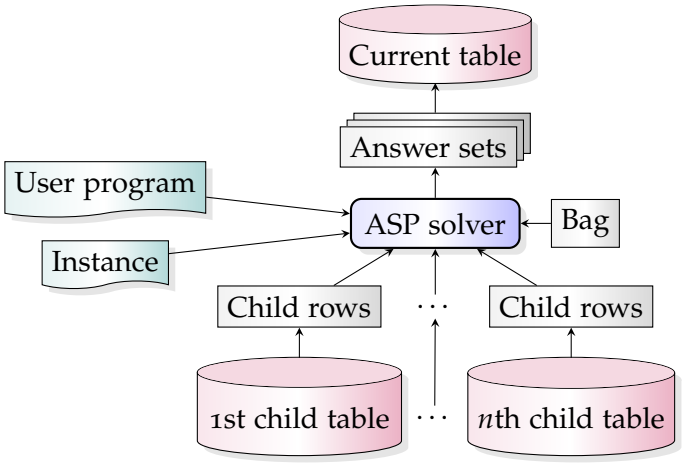


Figure 2 Data flow while processing a node with n children

the row (or top-level item set) itself. The reason is that when the characteristic is a tree of item sets, these pointers are usually required for reconstructing the intended structure of this tree from the answer sets, as is described in Section 5.

- In order to determine the *minimum cost* among all solutions, a table row contains the cost of the cheapest partial solution it represents.
- For *counting optimal solutions*, a row is required to contain the minimum cost of all the partial solutions it represents, as well as the number of represented partial solutions having that cost.

4 Executing the Problem-specific ASP Program

Figure 2 depicts the data flow when D-FLAT visits a node. D-FLAT first flattens the table of each child node, i.e., it builds a set of facts which describes all the rows in that table. Then, to compute the new table, D-FLAT invokes the integrated ASP solver with the following input:

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

The answer sets resulting from this constitute the new table. The user's dynamic programming algorithm must instruct D-FLAT which values it should store in a table row by means of special predicates that are used for communication between D-FLAT and the user's program (cf. Section 10). D-FLAT scans the answer sets for these predicates and inserts rows into the table accordingly.

Figure 3c depicts an example for the Minimum Vertex Cover problem.[2] Consider as the current node the one corresponding to the bag $\{a, b\}$. D-FLAT describes the current bag as just the two facts:[3]

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

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

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

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

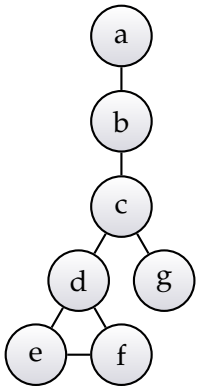
```
-introduced(X) ← childBag(_,X).
introduced(X) ← current(X), not -introduced(X).
removed(X) ← childBag(_,X), not current(X).
```

For some problems it is necessary to perform final actions once all tables have been computed. For cases like this, it is reasonable to use tree decompositions with an empty root node. Having empty roots even enables us to perform additional post-processing steps that are different from what happens when vertices are removed and thus *only* occur at the end. The user's ASP program is provided with the input fact `root` if the currently processed node is the empty root.

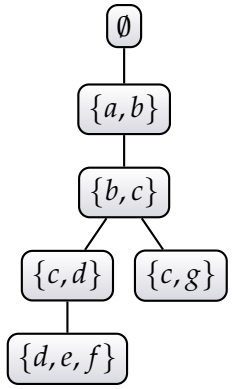
If there are rows in the root table, it is ensured that the solution count, cost and extension pointers only refer to optimal solutions because suboptimal solutions are discarded when merging rows with equal characteristics as described above.

[2] A concrete D-FLAT encoding for this problem can be found in [Bliem et al., 2012].

[3] To emphasize that these are input predicates provided by D-FLAT, we print them in color (see Section 10).



(a) A Minimum Vertex Cover instance



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

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

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

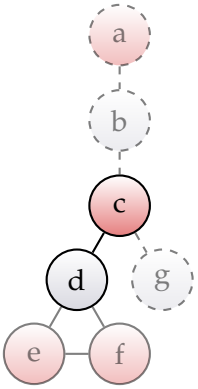
i	items	cost	extend
0	$\{b\}$	4	$\{(1,1)\}$
1	$\{c\}$	3	$\{(0,0),(2,0)\}$
2	$\{b,c\}$	4	$\{(0,0),(2,0)\}$

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

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

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

(c) The dynamic programming tables



(d) Visualization of a partial solution at the parent of the left leaf

Figure 3 Dynamic programming for Minimum Vertex Cover

5 Populating a Table from Answer Sets

Each answer set encodes a complete characteristic together with additional information (like extension pointers, a count or a cost). Therefore, when an answer set is reported that encodes a characteristic not already stored in any row, D-FLAT inserts a new row into the table and fills it with the encoded characteristic and additional information. When another answer set arrives specifying the same characteristic, D-FLAT looks up the existing table row and performs the following actions for the additional information:

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

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

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

   (a) If the problem type demands enumeration of solutions, D-FLAT inserts the encoded extension pointers into the corresponding sets of extension pointer tuples.

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

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

Reconstructing the intended trees from the branches described by answer sets is, however, a relatively intricate task. To accomplish this, an answer set must state, for each node of its specified path, extension pointers referring to the predecessor of the respective node if ambiguities should be avoided.

Just as it is required that there is at most one table row with a given characteristic, D-FLAT makes sure that for each node in a characteristic it holds that there are no two equal subtrees rooted at a child of that node. Equal subtrees are therefore merged, similar to merging rows with equal characteristics.

To summarize, when working with multi-level characteristics, the user can not only specify extension pointers for table rows but also for subsidiary item sets. If such pointers are present, D-FLAT uses this information to construct the characteristics

from the answer sets accordingly. In either case, it ensures that there is at most one table row for each characteristic, and that within a characteristic there are no duplicate subtrees at the same level.

6 Materializing the Solutions

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

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

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

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

table can be considered as bounded by a constant (which is the case if it is bounded by a function only depending on the treewidth).

7 Input Format

As we are using tree decompositions as a way to decompose problem instances, we require the input to be a graph.[4] D-FLAT makes this input available to the user's ASP program because the graph contains the problem-specific information and is needed to compute tables.

For instance, the following input describes the graph from Figure 3a:

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

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

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

8 Constructing a Tree Decomposition

The graph that was obtained as described in Section 7 is now decomposed. D-FLAT leaves the details of this to an external library that is concerned with heuristically constructing a tree decomposition that has near-optimal width (cf. Section 2). In the literature, algorithms for dynamic programming on tree decompositions can often be found to use a restricted class of decompositions, viz. normalized or semi-normalized tree decompositions.[5] A *semi-normalized tree decomposition* is a tree decomposition where each node falls under one of the following types:

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

[4] To be precise, D-FLAT can also handle hypergraphs, but here we only speak of graphs for the sake of simplicity.

[5] Normalized tree decompositions are sometimes also called *nice*. There is also the concept of *semi-nice tree decompositions* [Dorn and Telle, 2009] which are, however, different from our semi-normalized ones and are not supported by D-FLAT at this time.

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

At the moment, the only available possibility for controlling the tree decomposition that D-FLAT generates is specifying on the command-line which normalization should be performed. The correct choice, of course, depends on what the particular dynamic programming algorithm expects as a tree decomposition.

When the dynamic programming phase begins, a bottom-up traversal of the tree decomposition is performed to compute the tables. This way, upon reaching a tree decomposition node, the child tables are already computed. In each node, D-FLAT invokes the ASP solver to compute the new table, which is described next. Note that D-FLAT constructs a tree decomposition such that bag of the root is empty, as depicted in Figure 3b.[6]

9 Command-Line Interface

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

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

Table 1 summarizes the command-line options that control D-FLAT's behavior. Executing the D-FLAT binary `dflat` is illustrated by the following example call, presupposing a 3-Col instance with the file name `instance.lp` and an encoding with the file name `3col.lp` (for an example see Section 11), instructing D-FLAT to enumerate all optimal solutions (which also prints their number and costs):

```
dflat -p opt-enum -e edge 3col.lp < instance.lp
```

10 Interface to ASP Programs: Reserved Predicates

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

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

[6] It should be noted that when D-FLAT is asked to construct a semi-normalized tree decomposition, it also provides for empty leaf nodes because the algorithms for semi-normalized tree decompositions presented in [Bliem et al., 2012] consist of two separate encodings – one for exchange nodes and one for join nodes.

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

Table 1 Command-line options for D-FLAT

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

Table 2 Predicates supplied to the user's program by D-FLAT

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

These predicates are summarized in Tables 2 and 3, respectively.

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

11 Example: Graph Coloring

The D-FLAT encoding in Listing 1 serves as an example of how to solve the 3-colorability problem of graphs (3-Col for short) using D-FLAT. If the treewidth of instances is bounded, this encoding together with D-FLAT solves the problem in linear time.

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

Table 3 Predicates in the ASP program's answer sets recognized by D-FLAT

Listing 1 A D-FLAT encoding for the 3-Col problem

```
color(red;green;blue).
% For each child node, guess a child table row to extend
1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
% Keep colors of still-current vertices
item(map(X,C)) ← extend(R), childItem(R,map(X,C)),
    current(X).
% Only join matching colorings
← item(map(V,C0;C1)), C0 ≠ C1.
% Guess color of introduced nodes
1 { item(map(X,C)) : color(C) } 1 ← introduced(X).
% Check that adjacent vertices are colored differently
← edge(X,Y), item(map(X;Y,C)).
```

References

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

[Bliem, 2012] Bliem, B. (2012). Decompose, guess & check: Declarative problem solving on tree decompositions. Master's thesis, TU Wien, Vienna.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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