D-FLAT: Declarative Problem Solving Using Tree Decompositions and Answer-Set Programming

BERNHARD BLIEM, MICHAEL MORAK and STEFAN WOLTRAN

Institute of Information Systems 184/2
Vienna University of Technology
Favoritenstrasse 9–11, 1040 Vienna, Austria
(*e-mail:* `[surname]@dbai.tuwien.ac.at`)

Abstract

In this work, we propose Answer-Set Programming (ASP) as a tool for rapid prototyping of dynamic programming algorithms based on tree decompositions. In fact, many such algorithms have been designed, but only a few of them found their way into implementation. The main obstacle is the lack of easy-to-use systems which (i) take care of building a tree decomposition and (ii) provide an interface for declarative specifications of dynamic programming algorithms. In this paper, we present D-FLAT, a novel tool that relieves the user of having to handle all the technical details concerned with parsing, tree decomposition, the handling of data structures, etc. Instead, it is only the dynamic programming algorithm itself which has to be specified in the ASP language. D-FLAT employs an ASP solver in order to compute the local solutions in the dynamic programming algorithm. In the paper, we give a few examples illustrating the use of D-FLAT and describe the main features of the system. Moreover, we report experiments which show that ASP-based D-FLAT encodings for some problems outperform monolithic ASP encodings on instances of small treewidth.
To appear in *Theory and Practice of Logic Programming* (TPLP).

1 Introduction

Computationally hard problems can be found in almost every area of computer science, hence the quest for general tools and methods that help designing solutions to such problems is one of the central research challenges in our field. One particularly successful approach is Answer-Set Programming (Niemelä 1999; Marek and Truszczyński 1999; Brewka et al. 2011)—ASP, for short—where highly sophisticated solvers (Leone et al. 2006; Gebser et al. 2011) provide a rich declarative language to specify the given problem in an intuitive and succinct manner. On the other hand, the concept of dynamic programming (Larson 1967; Wagner 1995) denotes a general recursive strategy where an optimal solution to a problem is defined in terms of optimal solutions to its subproblems, thus constructing a solution "bottom-up", from simpler to more complex problems.

One particular application of dynamic programming is the design of algorithms which proceed along tree decompositions (Robertson and Seymour 1984) of the problem at hand, or rather of its graph representation. The significance of this approach is highlighted by Courcelle's seminal result (see, e.g., (Courcelle 1990)) which states that every problem definable in monadic second-order logic can be solved in linear time on structures of bounded treewidth; formal definitions of these concepts will be provided in Section 2.2. This suggests a two-phased methodology for problem solving, where first a tree decomposition of the given problem instance is obtained and subsequently used in a second phase to solve the problem by a specifically designed algorithm (usually employing dynamic programming) traversing the tree decomposition; see, e.g., (Bodlaender and Koster 2008) for an overview of this approach. Such tree decomposition based algorithms have been successful in several applications including constraint satisfaction problems such as MAX-SAT (Koster et al. 2002), and also bio-informatics (Xu et al. 2005; Cai et al. 2008).

While good heuristics to obtain tree decompositions exist (Dermaku et al. 2008; Bodlaender and Koster 2010) and implementations thereof are available, the actual implementation of dynamic programming algorithms that work on tree decompositions had, as yet, often to be done from scratch. In this paper, we present a new method for declaratively specifying the dynamic programming part, namely by means of ASP programs. As mentioned, dynamic programming relies on the evaluation of subproblems which themselves are often combinatorial in nature. Thanks to the Guess & Check principle of the ASP paradigm, using ASP to describe the treatment of subproblems is thus a natural choice and also separates our approach from existing systems (which we will discuss in the conclusion section of this paper). Using ASP as a language in order to describe the constituent elements of a dynamic programming algorithm obviously suggests to also employ sophisticated off-the-shelf ASP systems as internal machinery for evaluating the specified algorithm. Thus, our approach not only takes full advantage of the rich syntax ASP offers to describe the dynamic programming algorithm, but also delegates the burden of local computations to highly efficient ASP systems.

We have implemented our approach in the novel system D-FLAT[1] which takes care of the computation of tree decompositions and also handles the "bottom-up" data flow of the dynamic programming algorithm.[2] All that is required of users is an idea for such an algorithm. Hence, D-FLAT serves as a tool for rapid prototyping of dynamic programming algorithms but can also be considered for educational purposes. As well, ASP users are provided with an easy-to-use interface to decompose problem instances—an issue which might allow large instances of practical importance to be solved, which so far could not be handled by ASP systems. To summarize, D-FLAT provides a new method for problem solving, combining the advantages of ASP and tree decomposition based dynamic programming. Some attempts to ease the specification of dynamic programming algorithms already exist (see Section 5), but we are not aware of any other system that employs ASP for dynamic programming on tree decompositions.

The structure of this paper is as follows: In Section 2, we first briefly recall logic programming under the answer-set semantics and provide the necessary background for (hyper)tree decompositions and dynamic programming. In Section 3, we introduce the features of D-FLAT step-by-step, providing ASP specifications of several standard (but also some novel) dynamic programming algorithms. Section 4.1 gives some system specifics, and Section 4.2 reports on a preliminary experimental evaluation. In the conclusion of the paper, we address related work and give a brief summary and an outlook.

[1] (D)ynamic Programming (F)ramework with (L)ocal Execution of (A)SP on (T)ree Decompositions; available at `http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/`.

[2] Concerning the decomposition, we employ the htdecomp library of (Dermaku et al. 2008) making our system amenable to hypertree decompositions, as well. This, in fact, allows to decompose arbitrary finite structures thus extending the range of applicability for our system.

2 Preliminaries

2.1 Logic Programs and Answer-Set Semantics

The proposed system D-FLAT uses ASP to specify a dynamic programming algorithm. In this section, we will therefore briefly introduce the syntax and semantics of ASP. The reader is referred to (Brewka et al. 2011) for a more thorough introduction.

A *logic program* Π consists of a set of *rules* of the form

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

We call $a_1, \ldots, a_k$ and $b_1, \ldots, b_n$ *atoms*. A *literal* is either an atom or its default negation which is obtained by putting not in front. For a rule $r \in \Pi$, we call $h(r) = \{a_1, \ldots, a_k\}$ its *head* 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*, $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. Note that the head may be empty, in which case we call r an *integrity constraint*. If the body is empty, r is called a *fact*, and the $\longleftarrow$ symbol can be omitted.

A rule r is satisfied by a set of atoms I (called an *interpretation*) iff $I \cap h(r) \neq \emptyset$ or $b^-(r) \cap I \neq \emptyset$ or $b^+(r) \setminus I \neq \emptyset$. I is a *model* of a set of rules iff it satisfies each rule. I is an *answer set* of a program Π iff it is a subset-minimal model of the program $\Pi^I = \{h(r) \longleftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$, called the *Gelfond-Lifschitz reduct* (Gelfond and Lifschitz 1991) of Π w.r.t. I.

In this paper, we use the input language of Gringo (Gebser et al. 2010) for logic programs. Atoms in this language are predicates whose arguments can be either variables or ground terms. Such programs can be seen as abbreviations of variable-free programs, where all the variables are instantiated (i.e., replaced by ground terms). The process of instantiation is also called *grounding*. It results in a propositional program which can be represented as a set of rules which adhere to the above definition.

As an example, an instance for the 3-COL problem[3] (covered in Section 3.1) consists of a set of vertices and a set of edges, so the following set of facts represents a valid instance:

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

The following logic program solves 3-COL for instances of this form:

```
1 color(red;green;blue).
2 1 { map(X,C) : color(C) } 1 :- vertex(X).
3 :- edge(X,Y), map(X,C), map(Y,C).
```

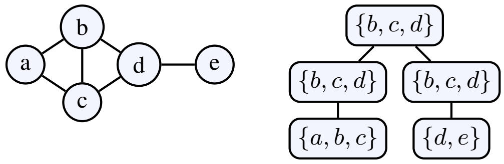
Solving 3-COL for an instance amounts to grounding this encoding together with the instance as input and then solving the resulting ground program. Line 1 of the encoding is expanded by the grounder to three facts which state that “red”, “green” and “blue” are colors. Line 2 contains a cardinality constraint in the head and is conceptually expanded to

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

before the grounder expands this rule by substituting ground terms for `X`. Roughly speaking, a cardinality constraint $l\{L_1, \ldots, L_n\}u$ is satisfied by an interpretation I iff at least l and at most u of the literals $L_1, \ldots, L_n$ are true in I. Therefore, the rule in question

[3] 3-COL is defined as the problem of deciding whether, for a given graph (V, E), there exists a 3-coloring, i.e., a mapping $f : V \to \{\mathrm{red}, \mathrm{green}, \mathrm{blue}\}$ s.t. for each edge $(a, b) \in E$ it holds that $f(a) \neq f(b)$.

Figure 1: A graph and a corresponding semi-normalized tree decomposition



expresses a choice of exactly one of `map(X,red)`, `map(X,green)`, `map(X,blue)`, for any vertex `X`. Finally, the integrity constraint in line 3 of the encoding ensures that no answer set maps the same color to adjacent vertices. For space reasons, we refer the reader to (Gebser et al. 2010) for more details on the input language.

2.2 Hypertree Decompositions

Tree decompositions and treewidth, originally defined in (Robertson and Seymour 1984), are a well known tool to tackle computationally hard problems (see, e.g., (Bodlaender 1993; Bodlaender 2005) for an overview). Informally, treewidth is a measure of the cyclicity of a graph and many **NP**-hard problems become tractable if the treewidth is bounded. 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.

Several problems are better represented as hypergraphs for which the concept of tree decomposition can be generalized, see, e.g., (Gottlob et al. 2002). In the following, we therefore define hypertree decompositions, of which tree decompositions are a special case.

A *hypergraph* is a pair $H = (V, E)$ with a set V of vertices and a set E of hyperedges. A hyperedge $e \in E$ is itself a set of vertices with $e \subseteq V$. A *hypertree decomposition* of a hypergraph $H = (V, E)$ is a pair $HD = \langle T, \chi \rangle$, where $T = (N, F)$ is a (rooted) tree, with N being its set of nodes and F its set of edges, and χ is a labeling function such that for each node $n \in N$ the so-called *bags* $\chi(n) \subseteq V$ of *HD* meet the following requirements:

1. for every $v \in V$ there exists a node $n \in N$ such that $v \in \chi(n)$,
2. for every $e \in E$ 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.

A hypertree decomposition $\langle T, \chi \rangle$ is called *semi-normalized* if each node n in T has at most two children—nodes with zero (resp. one, two) children are called leaf (resp. exchange, join) nodes—and for a join node n with children n_1 and n_2, $\chi(n) = \chi(n_1) = \chi(n_2)$ holds.[4] Figure 1 depicts a semi-normalized tree decomposition for the example graph given by the problem instance in Section 2.1.

For defining the *width* of a hypertree decomposition $\langle (N, F), \chi \rangle$ of a hypergraph (V, E), we need an additional labeling function $\lambda : N \to 2^E$ such that, for every $n \in N$,

[4] A few terminological remarks: The reason we use the term "*semi*-normalized" is that the (more restrictive) concept of *normalized* (also called *nice*) tree decompositions appears in the literature. Normalized tree decompositions are also semi-normalized. Moreover, hypertree decompositions generalize the notion of tree decompositions to the case of hypergraphs. Therefore, we often use both terms interchangeably if we are dealing with ordinary graphs, even though, strictly speaking, D-FLAT always produces a hypertree decomposition.

$\chi(n) \subseteq \bigcup_{e \in \lambda(n)} e$. A hypertree decomposition is called *complete* when for each hyperedge $e \in E$ there exists a node $n \in N$ such that $e \in \lambda(n)$. The *width* of a hypertree decomposition is then the maximum λ-set size over all its nodes. The minimum width over all possible hypertree decompositions is called the *generalized hypertree width*. The idea of this parameter is to capture the inherent difficulty of the given problem, such that the smaller the generalized hypertree width (i.e., the level of cyclicity), the easier the problem is to solve, see, e.g., (Gottlob et al. 2002). If $G = (V, E)$ is an ordinary graph, then the width of a tree decomposition $\langle T, \chi \rangle$ is defined differently, namely as the maximum bag size $|\chi(n)|$, minus one. For a given hypergraph, it is **NP**-hard to compute a hypertree decomposition of minimum width. However, efficient heuristics have been developed that offer good approximations (cf. (Dermaku et al. 2008; Bodlaender and Koster 2010)). It should be noted that in general there exist many possible hypertree decompositions for a given hypergraph, and there always exists at least one: the degenerated hypertree decomposition consisting of just a single node which covers the entire hypergraph. D-FLAT expects the dynamic programming algorithms provided by the user to work on any semi-normalized hypertree decomposition, so users do not have to worry about (nor can they currently determine) which decomposition is generated. This is up to the heuristic methods of the htdecomp library, which attempt to construct decompositions of small width.

2.3 Dynamic Programming on Hypertree Decompositions

The value of hypertree decompositions is that their size is only linear in the size of the given graph. Moreover, they provide a suitable structure to design dynamic programming algorithms for a wide range of problems. These algorithms generally start at the leaf nodes and traverse the tree to the root, whereby at each node a set of partial solutions is generated by taking those solutions into account that have been computed for the child nodes. The most difficult part in constructing such an algorithm is to identify an appropriate data structure to represent the partial solutions at each node: On the one hand, this data structure must contain sufficient information so as to compute the representation of the partial solutions at each node from the corresponding representation at the child node(s). On the other hand, the size of the data structure should only depend on the size of the bag (and not on the total size of the instance to solve). For this purpose, at each tree decomposition node we maintain a data structure which we call *table*. A table contains rows which we call *tuples* (i.e., mappings that assign some value to each current bag element), in which we store the partial solutions. Additionally, each tuple in non-leaf nodes may be associated with tuples from child nodes by means of pointers. These pointers determine which child tuples a tuple has originated from and are used to construct complete solutions out of the respective partial solutions when the computation of all tables is finished. [5]

We illustrate this idea for the 3-COL problem. Consider a given graph G and a corresponding tree decomposition. For each node n in the decomposition, we basically compute the valid colorings of the subgraph of G induced by $\chi(n)$, i.e., by the current bag, and store these colorings in the rows of the table associated with n. Hence each tuple of node n's

[5] If only the decision problem needs to be solved, these pointers are not necessary; in general, the algorithm and the structure of the tables depend on the problem type (cf. (Greco and Scarcello 2010; Gottlob et al. 2009)).

Figure 2: The 3-COL tuple tables for the instance and tree decomposition given in Figure 1

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

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

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

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

table assigns either "r", "g" or "b" to each vertex that is contained in the current bag $\chi(n)$. So far, the steps taken by the dynamic programming algorithm in n do not differ from a general Guess & Check approach to 3-COL. However, when computing the colorings for an exchange node n, we start from the colorings for the child node n' of n and adapt them with respect to the new vertices $\chi(n) \setminus \chi(n')$. In addition, we also keep track of which tuples of n' give rise to which newly calculated tuples of n. Figure 2 depicts the respective tables of each node of the semi-normalized tree decomposition of Figure 1. In each table, column i contains an index for each tuple, that can be used in column j of a potential parent node to reference it. For exchange nodes, an entry in column j is a set of pointers to such child tuples. For join nodes, an entry in column j is a set of pairs (l,r), where l (resp. r) references a tuple of the left (resp. right) child node. Using these pointers, it is possible to enumerate all complete solutions to the entire problem by a final top-down traversal.

Note that the size of the tables only depends on the width of the tree decomposition, and the number of tables 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.

3 D-FLAT by Example

In this section, we introduce the usage of the D-FLAT system by means of specific example problems. We begin with a relatively simple case to illustrate the basic functionality of the system and then continue with more complex applications. In the course of this, we will gradually introduce the features of D-FLAT and the special predicates responsible for the communication between D-FLAT and the user's programs. These predicates are summarized in Table 1 for future reference. A general description of the system and its possible applications is delegated to Section 4.1.

3.1 Graph Coloring

As a first example of how to solve an **NP**-complete problem using D-FLAT, we consider 3-COL, the 3-colorability problem for graphs. We have already given an encoding for 3-COL

Table 1: Reserved predicates for the communication of the user's programs with D-FLAT

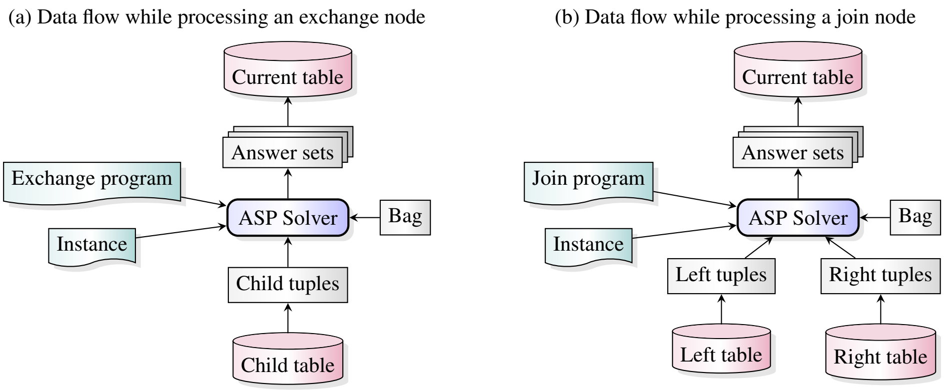
Input predicate	Meaning
`current(V)`	V is an element of the current bag.
`introduced(V)`	V was introduced in the current bag.
`removed(V)`	V was removed in the current bag.
`childTuple(I)`	I is the identifier of a child tuple. (Only in exchange nodes)
`childTupleL(I)`, `childTupleR(I)`	I is a tuple from the left resp. right child node. (Only in join nodes)
`mapped(I,X,Y)`	Child tuple I assigns to X the value Y.
`childCost(I,C)`	C is the total cost of the partial solution corresponding to the child tuple I.
`root`	Indicates that the current node is the root node. (Only in exchange nodes)
Output predicate	**Meaning**
`map(X,Y)`	Assign the value Y to the current bag element X.
`chosenChildTuple(I)`	Declare I to be the identifier of the child tuple that is the predecessor of the currently described one. (Only in exchange nodes; not necessary for decision problems)
`chosenChildTupleL(L)`, `chosenChildTupleR(R)`	Declare L, R to be the identifiers of the preceding child tuples from the left resp. right child node. (Only in join nodes; not necessary for decision problems)
`cost(C)`	Let C be the total cost of the current partial solution. (Only required when solving an optimization problem)
`currentCost(C)`	Let C be the local cost of the current tuple. (Only required in exchange nodes when solving an optimization problem and using the default join implementation)

in Section 2.1. Since this program, together with the instance as input, solves the whole problem, we call it a *monolithic encoding*. As 3-COL is fixed-parameter tractable w.r.t. the treewidth, we can take advantage of low treewidths by solving the problem with a dynamic programming algorithm that operates on a tree decomposition of the original input graph. We have sketched the dynamic programming algorithm for 3-COL in Section 2.3.

D-FLAT now provides a means to specify this dynamic programming algorithm in the ASP language. It reads the instance, stores its graph representation, and from this it constructs a semi-normalized tree decomposition. In order for D-FLAT to obtain a graph representation from the instance, the user only needs to specify on the command line which predicates indicate (hyper)edges in the graph representation (in this case only `edge/2`).

Following the idea of dynamic programming, we wish to compute a table for each tree decomposition node, such that a table can be computed using the tables of the child nodes;

Figure 3: Workflow for calculating a tree decomposition node's table



i.e., we perform the calculations in a bottom-up manner. Since semi-normalized tree decompositions only allow for two kinds of nodes—*exchange nodes*, which have one child, and *join nodes*, which have exactly two children with the same contents as the join node—it suffices to provide D-FLAT with an ASP program that computes the table for exchange nodes (*exchange program* for short) and with a program that computes the table for join nodes (*join program* for short).[6] These two programs are all that is required of the user to solve a problem. D-FLAT traverses the tree decomposition in post-order and, for each node, invokes the ASP solver with the proper program, i.e., with the exchange program for exchange nodes and with the join program for join nodes. An illustration of such single steps is given in Figure 3. The input for the exchange or join program consists of

- the original problem instance as supplied by the user,
- the tuples from the child nodes as a set of facts,
- the current bag, i.e., the current, introduced and removed vertices, as a set of facts.

The answer sets then constitute the current node's table. D-FLAT takes care of processing them and filling the appropriate data structures. Once all new tuples have thus been stored, it proceeds to the next node. This procedure continues until the root has been processed.

We now present an exchange program for 3-COL. Regarding the input that is supplied by D-FLAT, the set of current, introduced and removed vertices is given by predicates `current/1`, `introduced/1` and `removed/1`, respectively. Each child tuple has some identifier I and is declared by the fact `childTuple(I)`. The corresponding mapping is given by facts of the form `mapped(I,X,C)`, which signifies that in the tuple I the vertex X is assigned the color C. (The predicate name "mapped" was chosen in analogy to the present tense term "map".)

[6] In fact, D-FLAT produces tree decompositions where all leaves as well as the root node have an empty bag. Empty leaf nodes have the advantage that they do not involve any problem-specific computation (at least in the use cases we considered so far) but just deliver the empty tuple (with cost 0) to its parent node. (This default behavior is mirrored in D-FLAT in the sense that it currently only supports user-specified ASP programs for exchange and join nodes). On the other hand, having an empty root node guarantees that final actions of the dynamic programming algorithm can always be specified in the program for the exchange nodes.

```
1 color(red;green;blue).
2 1 { chosenChildTuple(I) : childTuple(I) } 1.
3 map(X,C) :- chosenChildTuple(I), mapped(I,X,C), current(X).
4 1 { map(X,C) : color(C) } 1 :- introduced(X).
5 :- edge(X,Y), map(X,C), map(Y,C).
```

The program above is based on the intuition that in an exchange node each child tuple gives rise to a set of new tuples, such that the new tuples coincide on the coloring of common vertices (line 3), and the coloring of introduced vertices is guessed (line 4), followed by a check (line 5) which uses the `edge/2` predicate of the problem instance.

Each answer set here constitutes exactly one new tuple in the current node's table. The output predicate `map/2` is used to specify the partial coloring of the new tuple, just like it was used in the monolithic encoding. It must assign a color to each vertex in the current node. Another output predicate recognized by D-FLAT is `chosenChildTuple/1`. It indicates which child tuple a partial coloring (characterized by an answer set) corresponds to and must, of course, only have one child tuple identifier as its extension. This predicate is required for the reconstruction of complete solutions after all tables have been computed and can therefore be omitted if just the decision problem should be solved.

It still remains to provide D-FLAT with a program for processing the other kind of nodes, viz., the join nodes. A join program for this purpose could look like this:

```
1 1 { chosenChildTupleL(I) : childTupleL(I) } 1.
2 1 { chosenChildTupleR(I) : childTupleR(I) } 1.
3 :- chosenChildTupleL(L), chosenChildTupleR(R),
      mapped(L,X,A), mapped(R,X,B), A != B.       % Only join equal tuples
4 map(X,V) :- chosenChildTupleL(L), mapped(L,X,V).
```

Here an answer set obviously must state *for each* of the two child nodes the chosen preceding tuple (indicated by `chosenChildTupleL` and `chosenChildTupleR`). In the case of 3-COL, two tuples match (i.e., are joined to a new tuple) iff their partial assignments coincide (line 3). The tuple resulting from such a join is then also equal to each of the two matching child tuples (line 4).

This particular join behavior is so basic that it also applies to other common problems. Therefore, D-FLAT offers a (quite efficient) default implementation of this behavior, which users can resort to (if it suffices) instead of writing their own join programs.

We would like to point out here the difference between the role of exchange and join nodes. In particular, one might wonder why for join nodes there is a default implementation that can often be used, whereas there is none for exchange nodes. This difference is due to the idea of dynamic programming which, on the one hand, involves solving partial problems and, on the other hand, requires combining partial solutions (which is clearly separated, thanks to the concept of semi-normalized tree decompositions). Obviously, the combining is explicitly done in join nodes, whereas usually the solving of partial problems is performed in exchange nodes, which therefore require problem-specific code. Exchange nodes *always* require problem-specific code since the entire problem can be seen as a single degenerated exchange node. Hence, parts of exchange programs resemble monolithic encodings very much, even though these are technically irrelevant to D-FLAT. Indeed, exchange programs can be seen as more general than monolithic encodings since a correct exchange program must in principle also solve the entire problem. In some cases however, knowledge of the problem is also needed in join nodes, as the next subsection will show.

3.2 Boolean Satisfiability

Instances for the satisfiability problem (SAT) are given by the predicates `pos(C,V)` and `neg(C,V)`, denoting that the propositional variable V occurs positively resp. negatively in the clause C. Each clause and variable is declared using the predicates `clause/1` and `atom/1`, respectively. The following monolithic encoding solves such instances:

```
1 { map(A,true), map(A,false) } 1 :- atom(A).
sat(C) :- pos(C,A), map(A,true).
sat(C) :- neg(C,A), map(A,false).
:- clause(C), not sat(C).
```

We can obtain a graph representation of a SAT instance by constructing its incidence graph, i.e, the graph obtained by considering the clauses and variables as vertices and connecting a clause vertex with a variable vertex by an edge iff the respective variable occurs in the respective clause. Again, in order for D-FLAT to transform the input into a graph representation, the user is only required to state that `pos/2` and `neg/2` indicate edges.

A dynamic programming algorithm for the model counting problem working on tree decompositions of the incidence graph is given in (Samer and Szeider 2010). We now present a possible ASP encoding for the exchange node that follows that work's general idea of such an algorithm and generalizes it for semi-normalized tree decompositions. It should be noted that we primarily assign truth values to current propositional atoms like in the monolithic encoding, but as a consequence of this we also assign true or false to each current clause depending on whether or not it is satisfied by the partial interpretation represented by the tuple. We need this information on the status of a clause because, when a clause is removed, all tuples not satisfying this clause must be eliminated.

```
1 1 { chosenChildTuple(I) : childTuple(I) } 1.
2 :- clause(C), removed(C), chosenChildTuple(I), not mapped(I,C,true).
3 map(X,true) :- chosenChildTuple(I), mapped(I,X,true), current(X).
4 1 { map(A,true), map(A,false) } 1 :- atom(A), introduced(A).
5 map(C,true) :- pos(C,A), current(C), map(A,true).
6 map(C,true) :- neg(C,A), current(C), map(A,false).
7 map(X,false) :- current(X), not map(X,true).
```

The mentioned elimination of non-satisfying truth assignments is performed by the check in line 2. The child tuple's partial interpretation is retained (lines 3 and 7) and extended by a guess on introduced atoms (line 4). If a clause was satisfied at some point before, it remains so due to line 3, whereas lines 5 and 6 mark clauses as satisfied by the current partial assignment.

For the SAT problem we cannot use D-FLAT's default implementation for join nodes. The reason is that, in order to be joined, two tuples do not need to coincide on the values they assign to clauses, but only on the assignments for atoms. If, for a given truth assignment on the current atoms, a clause is satisfied by either child tuple, it is also satisfied by the tuple resulting from joining the two. The following join program follows this idea:

```
1 { chosenChildTupleL(I) : childTupleL(I) } 1.
1 { chosenChildTupleR(I) : childTupleR(I) } 1.
:- mapped(L,A,true), mapped(R,A,false), atom(A), chosenChildTupleL(L),
    chosenChildTupleR(R).
:- mapped(L,A,false), mapped(R,A,true), atom(A), chosenChildTupleL(L),
    chosenChildTupleR(R).
map(X,true) :- chosenChildTupleL(I), mapped(I,X,true).
map(X,true) :- chosenChildTupleR(I), mapped(I,X,true).
map(X,false) :- current(X), not map(X,true).
```

3.3 Minimum Vertex Cover

Encodings as outlined above suffice to instruct D-FLAT to solve the respective enumeration problems (“What are the solutions?”), counting problems (“How many solutions exist?”) and decision problems (“Is there a solution?”). Often it is also desired to solve optimization problems, for which additional information regarding the cost of a (partial) solution must be supplied in the encodings. As an example, we briefly introduce how the “drosophila” of fixed-parameter algorithms, the MINIMUM VERTEX COVER problem, can be solved. Instances are again given by `vertex/1` and `edge/2`. As before, we begin with a monolithic encoding for comparison:

```
1 { map(X,in), map(X,out) } 1 :- vertex(X).
:- edge(X,Y), map(X,out), map(Y,out).
cost(C) :- C = #count{ map(X,in) }.
#minimize[ cost(C) = C ].
```

By means of the `#minimize` statement, we leave it to the ASP solver to filter out suboptimal solutions. However, it is a mistake to use such an optimization statement when writing the exchange program for D-FLAT because then one would filter out tuples whose local cost might exceed that of others but which in the end would yield a better global solution.

An exchange program for MINIMUM VERTEX COVER could look like this:

```
1 1 { chosenChildTuple(I) : childTuple(I) } 1.
2 map(X,Y) :- current(X), chosenChildTuple(I), mapped(I,X,Y).
3 1 { in(X), out(X) } 1 :- introduced(X).
4 map(X,in) :- in(X).
5 map(X,out) :- out(X).
6 :- edge(X,Y), map(X,out), map(Y,out).
7 currentCost(C) :- C = #count{ map(_,in) }.
8 cost(C) :- chosenChildTuple(I), childCost(I,CC), IC = #count{ in(_) },
      C = CC + IC.
```

The `cost/1` predicate is recognized by D-FLAT and specifies the cost of the partial solution (obtained by extending the current tuple with its predecessors, recursively). This number is computed in line 8 by adding to the preceding tuple’s cost (which is provided by D-FLAT as `childCost/2`) the cost which is due to introduced vertices—in this case, the number of vertices guessed as `in`.

A peculiarity when using the default join implementation (as in this example) is that in each tuple we not only need to store the cost of the corresponding partial solution but also the cost of the tuple itself, declared by `currentCost/1`, because upon joining two coinciding tuples and adding the total costs of their associated partial solutions, the portion of the cost that is due to the coinciding child tuples is counted twice and must thus be subtracted from the sum. This is why D-FLAT also recognizes the `currentCost/1` predicate. When implementing the join program manually, this predicate is useless, as calculating the cost of a tuple resulting from a join is then up to the user.

Note that, as soon as the root of the tree decomposition has been processed by D-FLAT, it possesses all the information required for determining the cost and number of optimal solutions in constant time (i.e., without an additional tree traversal), as well as for enumerating all optimal solutions with linear delay by traversing the tree top-down, following each tuple’s pointers to its predecessors to construct solutions. The processing of an optimization problem instance is no different from that of problems without optimization, except that partial solution costs must be specified for each tuple.

3.4 Cyclic Ordering

Up until now, we have always used a graph representation of the problem instance. Some problems, however, can be represented more naturally by a hypergraph. An example is the CYCLIC ORDERING problem, which takes as instance a set V, described by the predicate `vertex/1`, and a set of triples described by `order(A,B,C)`, where A, B, C are declared vertices. $f : V \rightarrow \{1, \ldots, |V|\}$ is called an ordering of the vertices. A triple `order(A,B,C)` is said to be satisfied by an ordering f iff $f(A) < f(B) < f(C)$ or $f(B) < f(C) < f(A)$ or $f(C) < f(A) < f(B)$. The objective is to bring the vertices into an ordering such that all triples are satisfied.

The following monolithic encoding achieves this:

```
1 { map(V,1..N) } 1 :- vertex(V), N = #count { vertex(_) }.
:- map(V1,K), map(V2,K), V1 < V2.
lt(V1,V2) :- map(V1,K1), map(V2,K2), K1 < K2.
sat(A,B,C) :- order(A,B,C), lt(A,B), lt(B,C).
sat(A,B,C) :- order(A,B,C), lt(B,C), lt(C,A).
sat(A,B,C) :- order(A,B,C), lt(C,A), lt(A,B).
:- order(A,B,C), not sat(A,B,C).
```

One can naturally represent the problem by a hypergraph where one considers the elements of V as vertices and the triples as hyperedges. This ensures that the vertices in each triple must appear together in at least one bag and each triple can therefore be checked in an exchange node.

We use this opportunity to introduce a distinction between two ways of how to write exchange encodings: We can either do this, as we did in all the cases before, in a "bottom-up" way, i.e., by selecting preceding child tuples and then constructing the new tuples originating from them. When dealing with just a decision problem, another possibility is to proceed "top-down", i.e., by guessing first an assignment of values to current vertices and then checking if some child tuple exists that is a valid predecessor of the guess. This has the advantage that we avoid a guess over a potentially huge number of child tuples when probably many of the resulting tuples would coincide.

The following exchange encoding for CYCLIC ORDERING proceeds "top-down":

```
1 { map(V,1..N) } 1 :- current(V), N = #count { current(_) }.
:- map(V1,K), map(V2,K), V1 < V2.
lt(V1,V2) :- map(V1,K1), map(V2,K2), K1 < K2.
sat(A,B,C) :- order(A,B,C), lt(A,B), lt(B,C).
sat(A,B,C) :- order(A,B,C), lt(B,C), lt(C,A).
sat(A,B,C) :- order(A,B,C), lt(C,A), lt(A,B).
:- order(A,B,C), current(A;B;C), not sat(A,B,C).
gtChild(I,V1,V2) :- mapped(I,V1,K1), mapped(I,V2,K2), current(V1;V2),
    K1 > K2.
noMatch(I) :- lt(V1,V2), gtChild(I,V1,V2).
match :- childTuple(I), not noMatch(I).
:- not match.
```

The first seven lines are very similar to the monolithic encoding. The remainder of the program makes sure that there is some valid predecessor among the child tuples. Note that such an approach can only be employed for decision problems because counting or enumerating solutions require pointers to child tuples. In fact, this problem differs from the previously discussed insofar as for those the set of values that can be mapped to a vertex was fixed (e.g., three colors or a truth value), whereas here it would be required to assign each vertex a number from 1 up to the total number of vertices (just as the monolithic program does), if one were to really construct a solution to the problem with this approach. This would violate the principles of dynamic programming because subproblems would depend on the whole problem. However, note that in line 1, only a *local* ordering, i.e., an ordering of the current bag elements, is guessed, so the value mapped to a vertex is *not* its position in an ordering of all the vertices.

3.5 Further Examples and Overview

In this section, we have illustrated the functioning of D-FLAT via several examples in the course of which we have also introduced step-by-step the special predicates D-FLAT provides or recognizes. D-FLAT solutions for further problems are provided on the system website at `http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/`. Let it be noted here that it is also possible to handle problems higher on the polynomial hierarchy than **NP**. For instance, one can solve (propositional) ASP itself with D-FLAT, for instance by implementing the algorithm presented in (Jakl et al. 2009). For purposes like this, D-FLAT allows not only for assignments to all vertices but more generally for a hierarchy of assignments. For example, each top-level assignment (as we have been dealing with before) might have subsidiary assignments that are used to determine if the respective top-level assignment is valid. A detailed account of this is beyond the scope of this paper. We refer the reader to the system website for examples of this.

Let us briefly summarize the main features of D-FLAT. In many cases, the user has to implement only the program for exchange nodes, while for join nodes D-FLAT offers a default implementation. For certain problems, this default has to be overridden. In their programs, users can exploit the full language accepted by Gringo. The only restriction is that some predicates are reserved by D-FLAT. Table 1 gives a summary of these predicates and also points out under which circumstances (depending on the program or problem type) which predicates are used. Predicates used in the facts that constitute the problem instance are supplied verbatim to the exchange and join programs.

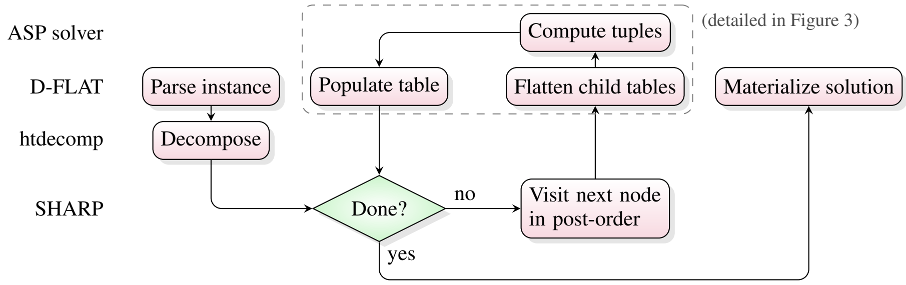
4 The D-FLAT System

4.1 System Description

D-FLAT is written in C++ and can be compiled for many platforms. The main libraries used are: (1) the SHARP framework[7] which takes care of constructing a tree decomposition of the input graph and then semi-normalizing it; as well, SHARP provides the skeleton for D-FLAT's data management; (2) SHARP itself uses the htdecomp library[8] which implements several heuristics for (hyper)tree decompositions, see also (Dermaku et al. 2008); (3) the Gringo/Clasp[9] family of ASP tools (see also (Gebser et al. 2011) for a recent survey) is finally used for grounding and solving the user-specified ASP programs. Figure 4 depicts the control flow between these components in a simplified way: D-FLAT initially parses the instance and constructs a hypergraph representation, which is then used by htdecomp

[7] See `http://www.dbai.tuwien.ac.at/research/project/sharp/`.
[8] See `http://www.dbai.tuwien.ac.at/proj/hypertree/downloads.html`.
[9] See `http://potassco.sourceforge.net/`.

Figure 4: A flowchart illustrating the (simplified) interplay of D-FLAT's components



to build a hypertree decomposition. SHARP is now responsible for traversing the tree in post-order. For each node, it calls D-FLAT which flattens the child tables, i.e., converts them to a set of facts which is given to the ASP solver to compute the new tuples as answer sets. These are collected by D-FLAT which populates the current node's table. When all nodes have been processed like this, D-FLAT reconstructs the solutions from the tables.[10]

Since we leave the decomposition part as well as the solving of the ASP programs to external libraries, D-FLAT immediately takes advantage of improvements in these libraries. It is thus also possible to switch to another ASP solver altogether without changing D-FLAT internals except, of course, the parts calling the solver. Likewise, our approach allows to replace the tree decomposition part with weaker (but more efficient) concepts like graph cuts, etc.

Since the user provides ASP programs for the exchange and, optionally, join nodes at runtime, D-FLAT is independent of the particular problem being solved. This means that there is no need to recompile in order to change the dynamic programming algorithm. It can be used out of the box as a rapid prototyping tool for a wide range of problems.

When executing the D-FLAT binary, the user can adjust its behavior using command-line options. The most important ones are briefly described in Table 2.

The problem instance, which must be a set of facts, is read from the standard input. The parser of D-FLAT recognizes the predicates that declare (hyper)edges, whose names are given as command-line options, and for each such fact introduces a (hyper)edge into the instance graph. The arguments of the (hyper)edge predicates implicitly declare the instance graph's set of vertices. Note that the tree decomposition algorithm used currently does not allow for isolated vertices, but this is no real limitation because usually solutions of a modified instance without the isolated vertices can be trivially extended.

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 exchange program with the file name `exchange.lp` (cf. Section 3.1), instructing D-FLAT to print the number of solutions and enumerate them:

```
dflat -e edge -x exchange.lp < instance.lp
```

[10] The data flow during the procedure within a node (indicated by the dashed box) is illustrated in Figure 3. Note that D-FLAT also features a default join implementation which does not use ASP and is not depicted.

Table 2: D-FLAT's most important command-line options

Argument	Meaning
`-e edge_predicate` (mandatory at least once)	`edge_predicate` is used in the problem instance as the predicate declaring (hyper)edges. Arguments of this predicate implicitly declare the vertices of the input graph.
`-x exchange_program` (mandatory)	`exchange_program` is the file name of the ASP program processing exchange nodes.
`-j join_program` (optional)	`join_program` is the file name of the ASP program processing join nodes. If omitted, the default implementation is used.
`-p problem_type` (optional)	`problem_type` specifies what kind of solution the user is interested in. It can be either "enumeration" (default), "counting", "decision", "opt-enum", "opt-counting" or "opt-value".

4.2 Experiments

In this section, we briefly report on first experimental results for the discussed problems. We compared D-FLAT encodings to monolithic ones using Gringo and Clasp. Each instance was limited to 15 minutes of CPU time and 6 GB of memory. Instances were constructed such that they have small treewidth by starting with a tree-like structure and then introducing random elements until the desired fixed decomposition width is reached.

Traditional ASP solvers employ clever heuristics to quickly either find some model or detect unsatisfiability, thereby being able to solve the decision variant of problems particularly well. In contrast, the dynamic programming approach of D-FLAT currently always calculates the tuple tables in the same way, whatever the problem variant may be—it is only in the final materialization stage that solutions are assembled differently, depending on the problem type.

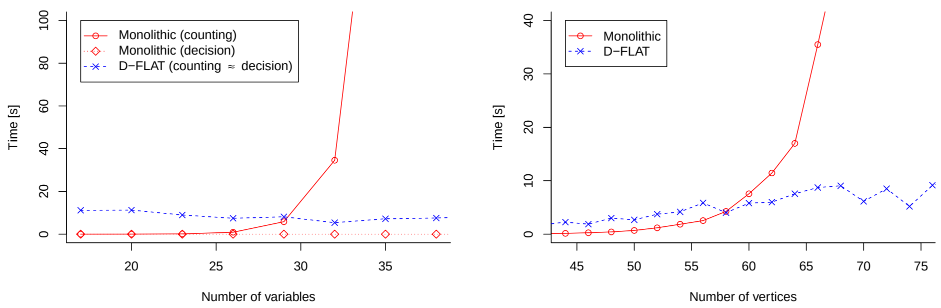
3-COL and SAT are prime examples of problems where traditional ASP solvers are very successful in solving the decision variant efficiently. However, when it no longer suffices to merely find *some* model (e.g., when dealing with counting or enumeration problems), the decomposition exploited by D-FLAT pays off for small treewidths, especially when there is a great number of solutions. In our experiments, the monolithic encodings indeed soon hit the time limit. D-FLAT, on the other hand, even sooner ran out of memory for enumeration due to its materializing all solutions in memory at the same time, and because the number of solutions increased rapidly with larger instances. This is a motivation to improve the materialization procedure in the future with incremental solving techniques.

Where D-FLAT excelled was the counting variant of these problems. It could solve each instance in a matter of seconds, while the monolithic program again ran out of time soon. Figure 5a illustrates the strengths and weaknesses of D-FLAT for the counting and decision variants of SAT: Although the monolithic program almost instantaneously solved the decision problem of all of the test instances, its running time on the counting problem soon exploded (standard ASP-solvers do not provide a dedicated functionality for counting

Figure 5: Comparison of D-FLAT to monolithic encodings

(a) The decision and counting variant of SAT on instances of treewidth 8

(b) Determining the optimum cost for MINIMUM VERTEX COVER on instances of treewidth 12



and thus have to implicitly enumerate all answer sets) whereas D-FLAT remained almost unaffected on average.

Although most of the time traditional ASP-solvers perform very efficiently on decision problems, for some problems they have more difficulties, in particular when the grounding becomes huge. Our investigations show that for the CYCLIC ORDERING problem, D-FLAT often outperforms the monolithic program, but it could also be observed that D-FLAT's running time is heavily dependent on the constructed tree decomposition. For this reason, we averaged over the performance on multiple tree decompositions for each instance size.

The MINIMUM VERTEX COVER problem proved to be a strong suit of D-FLAT (cf. Figure 5b). In optimization problems in general, stopping after the first solution has been found is not an option for traditional solvers, since yet undiscovered solutions might have lower cost. Another advantage of D-FLAT is that traditional solvers (at least in the case of Clasp) require two runs for counting or enumerating all optimal solutions: The first run only serves to determine the optimum cost, while the second starts from scratch and outputs all models having that cost. D-FLAT, in contrast, only requires one run at the end of which it immediately has all the information needed to determine the optimum cost.

As a concluding remark, recall that D-FLAT's main purpose is to provide a means to specify dynamic programming algorithms declaratively and not to compete with traditional ASP solvers, which is why we refrain from extensive benchmarks in this paper. Nonetheless, it can be concluded that D-FLAT is particularly successful for optimization and counting problems (provided the treewidth is small), especially when the number of solutions or the size of the monolithic grounding explodes.

5 Conclusion

Summary. We have introduced D-FLAT, a novel system which allows to specify dynamic programming algorithms over (hyper)tree decompositions by means of Answer-Set Programming. To this end, D-FLAT employs an ASP system as an underlying inference engine to compute the local solutions of the specified algorithm. We have provided case studies illustrating how the rich syntax of ASP allows for succinct and easy-to-read specifications of such algorithms. The system—together with the example programs given in Section 3 and the benchmark instances used in Section 4.2—is free software and available at

`http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/`.

Related Work. Several forms of support of dynamic programming exist in the world of PROLOG, where tabling of intermediate results is a natural concept. We mention here the systems TLP (Guo and Gupta 2008) and B-Prolog (Zhou 2011). In the world of datalog, we refer to the Dyna system (Eisner and Filardo 2011) which provides a wide range of features for memoization. On the other side of the spectrum, the work of (Gottlob et al. 2010) shows that the simple formalism of monadic datalog can be placed instead of monadic second-order (MSO) logic in meta-theorems (about fixed-parameter tractability in terms of treewidth) à la Courcelle. The latter approach is thus more of theoretical interest. The main difference between D-FLAT and the mentioned systems is that the user can make use of the full language of ASP and, in particular, employ the Guess & Check methodology to subproblems. However, it has to be mentioned that our system so far supports only dynamic programming in connection with (hyper)tree decompositions.

Future Work. As a next step, we have to compare D-FLAT in more detail to the aforementioned approaches in terms of both modeling capacities and performance issues. As well, we plan to provide different decomposition options within D-FLAT. In particular, we anticipate that even a rather simple (but efficient) decomposition of the graph, say a simple split into two parts, might in practice lead to performance gains over monolithic encodings. Another line of optimization concerns lazy evaluation strategies, for which incremental ASP techniques (Gebser et al. 2011) have been developed. Finally, the relation of our approach to reactive ASP (Gebser et al. 2011) might provide interesting new research directions.

Acknowledgments Supported by Vienna University of Technology special fund "Innovative Projekte 9006.09/008".

References

BODLAENDER, H. L. 1993. A tourist guide through treewidth. *Acta Cybern. 11,* 1-2, 1–22.

BODLAENDER, H. L. 2005. Discovering treewidth. In *Proc. SOFSEM*. LNCS, vol. 3381. Springer, 1–16.

BODLAENDER, H. L. AND KOSTER, A. M. C. A. 2008. Combinatorial optimization on graphs of bounded treewidth. *Comput. J. 51,* 3, 255–269.

BODLAENDER, H. L. AND KOSTER, A. M. C. A. 2010. Treewidth computations I. Upper bounds. *Inf. Comput. 208,* 3, 259–275.

BREWKA, G., EITER, T., AND TRUSZCZYNSKI, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

CAI, L., HUANG, X., LIU, C., ROSAMOND, F. A., AND SONG, Y. 2008. Parameterized complexity and biopolymer sequence comparison. *Comput. J. 51,* 3, 270–291.

COURCELLE, B. 1990. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, J. van Leeuwen, Ed. Elsevier and MIT Press, 193–242.

DERMAKU, A., GANZOW, T., GOTTLOB, G., MCMAHAN, B. J., MUSLIU, N., AND SAMER, M. 2008. Heuristic methods for hypertree decomposition. In *Proc. MICAI*. LNCS, vol. 5317. Springer, 1–11.

EISNER, J. AND FILARDO, N. W. 2011. Dyna: Extending datalog for modern AI (full version). Available under `http://cs.jhu.edu/~jason/papers/eisner+filardo.datalog11-long.pdf`.

GEBSER, M., GROTE, T., KAMINSKI, R., AND SCHAUB, T. 2011. Reactive answer set programming. In *Proc. LPNMR*. LNCS, vol. 6645. Springer, 54–66.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. 2010. A user's guide to gringo, clasp, clingo, and iclingo. Preliminary Draft. Available under `http://potassco.sourceforge.net`.

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.

GEBSER, M., SABUNCU, O., AND SCHAUB, T. 2011. An incremental answer set programming based system for finite model computation. *AI Commun. 24,* 2, 195–212.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput. 9,* 3/4, 365–386.

GOTTLOB, G., GRECO, G., AND SCARCELLO, F. 2009. Tractable optimization problems through hypergraph-based structural restrictions. In *Proc. ICALP (2)*. LNCS, vol. 5556. Springer, 16–30.

GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 2002. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci. 64,* 3, 579–627.

GOTTLOB, G., PICHLER, R., AND WEI, F. 2010. Monadic datalog over finite structures of bounded treewidth. *ACM Trans. Comput. Log. 12,* 1.

GRECO, G. AND SCARCELLO, F. 2010. Structural tractability of enumerating CSP solutions. In *Proc. CP 2010*. LNCS, vol. 6308. Springer, 236–251.

GUO, H.-F. AND GUPTA, G. 2008. Simplifying dynamic programming via mode-directed tabling. *Softw., Pract. Exper. 38,* 1, 75–94.

JAKL, M., PICHLER, R., AND WOLTRAN, S. 2009. Answer-set programming with bounded treewidth. In *Proc. IJCAI'09*. AAAI Press, 816–822.

KOSTER, A. M. C. A., VAN HOESEL, S., AND KOLEN, A. 2002. Solving partial constraint satisfaction problems with tree-decomposition. *Networks 40(3)*, 170–180.

LARSON, R. E. 1967. A survey of dynamic programming computational procedures. *IEEE Trans. Automat. Contr. 12,* 6, 767–774.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log. 7,* 3, 499–562.

MAREK, V. W. AND TRUSZCZYŃSKI, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*. Springer, 375–398.

NIEMELÄ, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell. 25,* 3–4, 241–273.

ROBERTSON, N. AND SEYMOUR, P. D. 1984. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B 36,* 1, 49–64.

SAMER, M. AND SZEIDER, S. 2010. Algorithms for propositional model counting. *J. Discrete Algorithms 8,* 1, 50–64.

WAGNER, D. B. 1995. Dynamic programming. *The Mathematica Journal 5,* 4, 42–51.

XU, J., JIAO, F., AND BERGER, B. 2005. A tree-decomposition approach to protein structure prediction. In *Proc. CSB 2005*. 247–256.

ZHOU, N.-F. 2011. The language features and architecture of B-Prolog. *CoRR abs/1103.0812*.