D-FLAT^2: Subset Minimization in Dynamic Programming on Tree Decompositions Made Easy

Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran

Institute of Information Systems
TU Wien, Vienna, Austria
[bliem,gcharwat,hecher,woltran]@dbai.tuwien.ac.at

Abstract. Many problems from the area of AI have been shown tractable for bounded treewidth. In order to put such results into practice, quite involved dynamic programming (DP) algorithms on tree decompositions have to be designed and implemented. These algorithms typically show recurring patterns that call for tasks like subset-minimization. In this paper we present D-FLAT^2, a system that allows one to obtain DP algorithms (specified in ASP) from simpler principles, where the DP formalization of subset-minimization is performed automatically. We illustrate the method at work by providing several DP algorithms – given in form of ASP programs – that are more space-efficient than existing solutions, while featuring improved readability, reuse and therefore maintainability of ASP code. Experiments show that our approach also yields a significant improvement in runtime performance.

1 Introduction

Many prominent NP-hard problems in the area of AI have been shown tractable for bounded treewidth, see, e.g., [10, 11]. Thanks to Courcelle's meta-theorem [4], it is sufficient to encode a problem as an MSO sentence in order to obtain such a result. To put this into practice, tailored systems for MSO logic are required, however. While there has been remarkable progress in this direction [13] there is still evidence that designing tree-decomposition-based dynamic programming (DP) algorithms for the considered problems from scratch results in more efficient software solutions (cf. [16]). To facilitate the development of such algorithms, the D-FLAT system [1] has been introduced. It allows for rapid prototyping by automatically generating a tree decomposition upon which it subsequently executes DP steps according to a specification given by the user. The crucial feature of D-FLAT is that the user can encode these problem-specific DP steps in Answer Set Programming (ASP, see [3]), and such an encoding is all that is required from the user.

The actual design of such algorithms can be quite tedious, in particular for problems located at the second level of the polynomial hierarchy like circumscription, abduction or abstract argumentation (see [6, 12]). In many cases, the increased complexity of such problems is caused by subset minimization or maximization subproblems (e.g., minimality of models in circumscription). It is exactly the handling of these subproblems which makes the design of the DP algorithms difficult. Especially in the world of ASP, we can witness such a phenomenon: In order to exploit the full expressive power of this

paradigm, a particular saturation programming technique is required (see, e.g., [14]) in order to express co-NP tests. Several approaches for relieving the user from this task have been proposed [8, 9]. For instance, in order to find minimal models of propositional formulas via ASP, it suffices to express the SAT problem together with a special minimize statement (supported by systems like *metasp* [9]). In this way, one easily obtains a program computing minimal models. Unfortunately, easy-to-use facilities like such minimize statements had no analog in the area of DP so far.

In this paper, we propose a solution to this issue: We outline a method for automatically obtaining DP algorithms for problems requiring optimization, given only an algorithm for a problem variant without optimization. For example, given a DP algorithm for SAT (like in [19]), our approach makes it possible to use this algorithm, together with simple statements on what to minimize, for finding only subset-minimal models. Making optimization implicit in this way makes the programmer's life considerably easier. Furthermore, naive DP algorithms on such problems often suffer from a naive check for subset optimality that requires unnecessarily much space and time. Our method avoids this issue by implicitly proceeding in two stages (instead of one stage in the naive case): First we compute solution candidates without regard to optimization and then we rule out invalid candidates by trying to find counterexamples. Because of its two-phased nature, our approach can do so in an efficient way. We are not aware of any work so far that introduces similar two-phased DP algorithms along with the appropriate data structures.

To underline the practical relevance of our work, we present an implementation of this two-stage algorithm called D-FLAT^2, which is an extension of D-FLAT.[1] We illustrate the simplicity of our approach by providing D-FLAT^2 encodings for several AI problems. This paper's application-oriented exposition complements the description of D-FLAT^2 given in [2], which focuses more on the technical realization of the system.

2 Background

In this section we introduce the concept of DP on tree decompositions (TDs) as realized in the D-FLAT system. We highlight the concepts on basis of enumeration variants for the SAT and $\subseteq$-MINIMAL SAT problems ("*Given a propositional formula ϕ in CNF, what are the (subset-minimal) models of ϕ?*"). In general, given some problem that is tractable for bounded treewidth (such as SAT) and an input instance (e.g., some formula ϕ), DP on TDs consists of the following steps: 1) The input instance is decomposed, thereby obtaining a TD. Each node in the TD represents parts of the original instance (e.g., some atoms and clauses of ϕ). 2) The TD is traversed in post-order. At each TD node, partial solutions are computed (e.g., partial interpretations of ϕ). 3) In order to enumerate the solutions for the whole problem instance (e.g., the models of ϕ), the TD is traversed a second time where the partial solutions are combined.

Usually, an algorithm designer has to implement these three steps for every problem from scratch. D-FLAT is a system for rapid prototyping of DP algorithms. Here, the user only has to develop a problem-specific ASP encoding that defines how the partial

[1] The systems D-FLAT and D-FLAT^2 are available at http://dbai.tuwien.ac.at/proj/dflat/system/ and https://github.com/hmarkus/dflat-2, respectively.

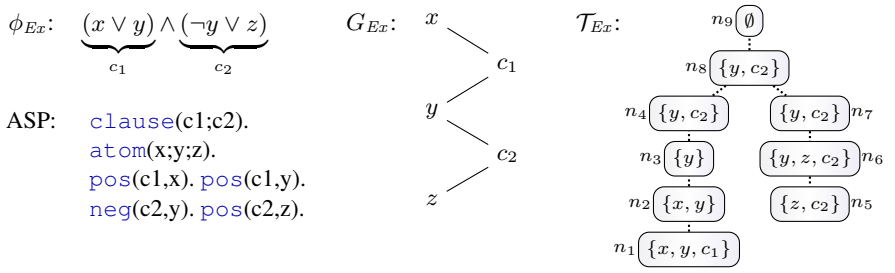

Fig. 1. Instance ϕ_{Ex}, ASP representation, incidence graph G_{Ex} and a semi-normalized TD $\mathcal{T}_{Ex}$.

solutions are constructed. This encoding will be invoked once for every node during a post-order traversal of the TD, and its models specify the partial solutions at the respective node. Communication between D-FLAT and the encoding is implemented via an interface consisting of pre-defined predicates. Overall, when D-FLAT is called together with the encoding and an input instance, it internally executes the steps described above and returns the solution.

Here, we first describe how the input for our running example (SAT) can be represented in D-FLAT. In Section 2.1 we introduce TDs formally, Section 2.2 describes how partial solutions are represented and how the problem-specific encoding can be written. Finally, in Section 2.3 we outline how the partial solutions are combined.

Input representation. In order to construct a TD, the input has to be specified in form of a graph. For SAT, we consider the *incidence graph* $G = (V, E)$ of ϕ, where V is the set of clauses and atoms occurring in ϕ, given as ASP facts `clause`$(\cdot)$ and `atom`$(\cdot)$ respectively. E is given as facts `pos`(c, a) (`neg`(c, a)) that denote that some atom a occurs positively (negatively) in clause c. An example is given in Figure 1.

2.1 Tree Decompositions

The intuition behind tree decomposition (TDs) [18] is to obtain a tree from a (potentially cyclic) graph by subsuming multiple vertices under one node and thereby isolating the parts responsible for cyclicity.

Definition 1. *A* tree decomposition *of a graph $G = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$ 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 met: 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 subtree of T induced by $\{n \in N \mid v \in \chi(n)\}$ is connected.*

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.*

In this work we consider so-called *semi-normalized* TDs where each node n is either a *leaf node* (n has no children), an *exchange node* (n has exactly one child) or a *join node* (n has exactly two children n', n'' with $\chi(n) = \chi(n') = \chi(n'')$). Furthermore, we assume that for root node r of T, $\chi(r) = \emptyset$ holds. D-FLAT constructs a TD

Input predicate	Meaning
`final`	The current tree decomposition node is the root.
`childNode`(N)	N is a child of the current decomposition node.
`bag`(N, V)	Vertex V is contained in the bag of the decomposition node N.
`current`(V)	Vertex V is an element of the current bag.
`introduced`(V)	Vertex V is a current vertex but was in no child node's bag.
`removed`(V)	Vertex V was in a child node's bag but is not in the current one.

Table 1. Input predicates describing the tree decomposition.

in polynomial time (using heuristics), and transforms the TD into a semi-normalized one in linear time without increasing the width. A possible TD $\mathcal{T}_{Ex}$ of our example formula ϕ_{Ex} is depicted in Figure 1. The width of $\mathcal{T}_{Ex}$ is 2. Table 1 lists the predicates that D-FLAT uses for providing the user's problem-specific encoding with information about the tree decomposition it constructed.

2.2 Data Representation and Algorithm Execution

D-FLAT traverses the decomposition in post-order. At each DP node the partial solutions are computed. Here, an ASP solver is called with the following input: a) the user-specified ASP encoding, b) the input instance, c) information about the current and child TD node(s) (see above), and d) the partial solutions computed in the child TD node(s). Each model returned by the solver represents a partial solution of the current node.[2] For problems in NP, *tables* are used to represent partial solutions, for harder problems so-called *item trees* [1] can be employed.

Data representation (Tables). Each row in a table represents a partial solution to the problem. When traversing the TD, tables for the already-visited child nodes are given to the user-specified encoding via predicates as listed in Table 2. Then, the partial solutions for the current TD node are computed via the user-specified encoding, and returned via the output predicates listed in Table 3. A row consists of a set of *items* that can (in general) store an arbitrary string. While items store *fixed* information (e.g., the truth assignment of atoms), auxiliary items contain information that may *change* during TD traversal (e.g., a clause becomes satisfied). We will explain this distinction in detail throughout the following example for SAT. The output predicate `extend`/1 specifies the child row(s) that give rise to the partial solution encoded by the respective model.

Listing 1 gives an example of a user-specified ASP encoding Π_{SAT} for the SAT problem. The encoding makes use of D-FLAT's input interface (see Tables 1 and 2) in the bodies of the rules, and the output interface (see Table 3) in the heads of the rules. The computed partial solutions for our running example are depicted in Figure 2. We will now go through Listing 1 and our example in detail. Let us first consider introduction node n_1 of $\mathcal{T}_{Ex}$. Since n_1 has no children, only Lines 7, 16 and 17 are of interest to us. For atoms x and y in $\chi(n_1) = \{x, y, c_1\}$ we guess their truth assignment (Line 7). In case an atom gets assigned true, it is added to the `item` set of the computed row. Lines 16 and 17 denote that a clause is added to the `auxItem` set in case it

[2] We use colors to highlight input (red), output (orange) and input instance (blue) predicates.

Input predicate	Meaning
childRow(R, N)	R is a table row belonging to decomposition node N.
childItem(R, I)	The item set of table row R contains item I.
childAuxItem(R, I)	The auxiliary item set of table row R contains item I.

Table 2. Input predicates describing tables of decomposition child nodes.

Output predicate	Meaning
item(I)	The item set of the current table row shall contain the item I.
auxItem(I)	The auxiliary item set of the current table row shall contain the item I.
extend(R)	The current table row shall extend the child table row R.

Table 3. Output predicates for constructing the table of the current decomposition node.

is satisfied by the current truth assignment. In n_1, we thus have four partial solutions ("rows"), namely $\{\emptyset, \{x, c_1\}, \{y, c_1\}, \{x, y, c_1\}\}$. In n_2, c_1 is removed from the bag. Whenever an atom was assigned false in a child row, Line 2 makes this explicit, and Line 3 identifies clauses that have not been satisfied yet. We now extend each partial solution of the child node (Line 5). Partial solution $\emptyset$ is not extended, since it does not satisfy c_1 (Line 11). The other partial solutions are extended and their information, restricted to the current bag, is kept via Lines 13 and 14. In Figure 2, this extension is marked with dashed arrows. In join nodes, we extend exactly one row per child table at a time (Line 5). Extended partial solutions have to agree on the truth assignment of atoms (Line 9). Consider join node n_8. Here, the row containing partial solution $\{y, c_2\}$ extends $\{y\}$ in n_4 and $\{y, c_2\}$ in n_7, since the latter two both contain the same truth assignment. This also highlights the difference between item and auxItem: In the former, items have to be contained in both extended partial solutions (they have to agree on the truth assignment of atoms), in the latter a union over the auxiliary items is built.

```
1 % Define false atoms and unsatisfied clauses
2 f(R,X)      ← childRow(R,N), bag(N,X), not childItem(R,X).
3 unsat(R,C)  ← childRow(R,N), bag(N,C), not childAuxItem(R,C).
4 % Guess partial solutions to be extended
5 1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
6 % Guess truth value of introduced atoms
7  { item(A) : atom(A), introduced(A) }.
8 % Only join rows coinciding on truth values of atoms
9 ← extend(X;Y), atom(A), childItem(X,A), f(Y,A).
10 % Rows with unsatisfied, removed clauses are not extended
11 ← clause(C), removed(C), extend(R), unsat(R,C).
12 % True atoms and satisfied clauses are kept
13 item(X)     ← extend(R), childItem(R,X),     current(X).
14 auxItem(C)  ← extend(R), childAuxItem(R,C), current(C).
15 % Through guess, clauses may become satisfied
16 auxItem(C) ← current(C;A), pos(C,A),      item(A).
17 auxItem(C) ← current(C;A), neg(C,A), not item(A).
```

Listing 1. Π_{SAT}: D-FLAT encoding for solving SAT.

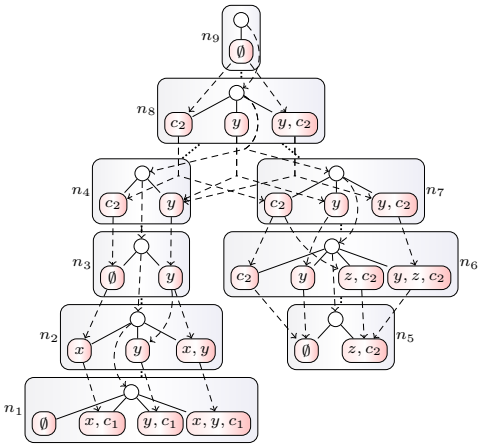

Fig. 2. Tables (`length`(1)-item trees) for SAT of ϕ_{Ex} in $\mathcal{T}_{Ex}$.

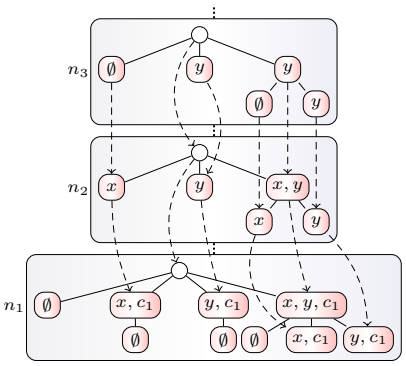

Fig. 3. Item trees for $\subseteq$-MINIMAL SAT of ϕ_{Ex} in $\mathcal{T}_{Ex}$.

Data representation (Item trees). For problems harder than NP, D-FLAT provides *item trees* to store partial solutions. A brief overview is given here, for details we refer to [1]. The predicates specifying item trees computed in the child nodes are given in Table 4, output predicates are given in Table 5. Similar to tables, each node in an item tree contains an `item`/2 and an `auxItem`/2 set. Each item tree node additionally has a set of *extension pointer tuples* that represents its origin, denoted by `extend`/2. (Note that these are now binary predicates.) Each root-to-leaf path has a particular `length`/1, and the *level* of an item tree node is its depth on the path. At the TD's root, each item tree node must be labeled with either `accept` or `reject` if it is a leaf, otherwise with `or`/1 or `and`/1. D-FLAT uses this to filter out solution candidates for which "counterexamples" exist: Only so-called *accepting* nodes are kept at the TD root, where a node is accepting if a) its label is `accept`, or b) its label is `or` and at least one child is accepting, or c) its label is `and` and all children are accepting. One can view the table-based data structure as a special case of item trees, where the `length` of each root-to-leaf path is one, and the root node is of type `or`. Furthermore, contrary to tables, where each model returned by the ASP solver represents a row, for item trees a model represents a single root-to-leaf path in the item tree.

Now we will explain item trees on basis of the $\subseteq$-MINIMAL SAT problem. Conceptually, we store solution candidates at depth 1 of the item trees (similar to partial solutions stored in the rows for SAT). Furthermore, we store so-called *counter candidates* at depth 2. A counter candidate is a potential witness for the solution candidate (its parent) not being subset-minimal (cf., e.g., [12]). In D-FLAT, it is again only required to specify a single ASP encoding, depicted in Listing 2. The encoding defines that item sets for both solution and counter candidates are computed as in the SAT problem. Additionally, this encoding ensures that partial interpretations represented by counter candidates are strict subsets of partial interpretations represented by solution candidates. Line 1 states that we have an item tree of depth 2. Furthermore, the encoding is designed to only return a solution in case there exists some (`or`(0)) solution candidate at level 1, such that

Input predicate	Meaning
atNode(S, N)	S is an item tree node belonging to decomposition node N.
rootOf(S, N)	S is the root of the item tree at decomposition node N.
sub(R, S)	R is an item tree node with child S.
childItem(S, I)	The item set of item tree node S contains item I.
childAuxItem(S, I)	The auxiliary item set of item tree node S contains item I.

Table 4. Input predicates describing item trees of child nodes in the decomposition.

Output predicate	Meaning
item(L, I)	I is in the item set of the node at level L in the current root-to-leaf path.
auxItem(L, I)	I is in the auxiliary item set at level L in the current root-to-leaf path.
extend(L, S)	Node at level L in current root-to-leaf path extends child item tree node S.
length(L)	The current root-to-leaf path has length L.
or(L)/and(L)	The node at level L in the current root-to-leaf path has type "or"/"and".
accept/reject	The leaf in the current root-to-leaf path has type "accept"/"reject".

Table 5. Output predicates for constructing the item tree of the current decomposition node.

no (and(1) in combination with Line 28) smaller counter candidate at level 2 exists. Similar to Π_{SAT}, Lines 2-7 make explicit if an atom is false or a clause is unsatisfied in an item tree node. Exactly one root-to-leaf path of each child item tree is extended (Lines 9-10). Then, for each level it is guessed whether an introduced atom is contained in the interpretation represented by the item set (Line 12), root-to-leaf paths are only joined in case the respective item sets coincide on their truth assignments for current atoms (Line 14), item sets with unsatisfied removed clauses are not extended (Line 16), truth assignments are propagated (Lines 18-19) and the set of satisfied clauses is updated (Lines 21-22). Line 24 guarantees that the interpretation in a counter candidate is a subset of (or equal to) that of a solution candidate. Then, flag `smaller` denotes that the counter candidate represents a proper subset of the solution candidate (Lines 26-27). In the final (i.e., root) node of the TD, solution candidates are rejected that still have a smaller counter candidate, and accepted otherwise (Lines 28-29). Figure 3 contains the computed item trees for TD nodes n_1, n_2 and n_3 of our running example. In n_1, we construct the item sets at depth 1 as described for SAT. The item sets at depth 2 represent counter candidates that are strict subsets of the interpretations at depth 1 (note that we omit counter candidates without `smaller` here). In n_2, clause c_1 is removed. Hence, we remove all item sets representing interpretations that do not satisfy c_1. In n_3, atom x is removed. This results in two item sets at depth 1 that both solely contain y. However, they differ in the counter candidates stored at depth 2.

2.3 Obtaining Solutions

At the TD's root, from the properties of tree decompositions we know that the whole instance has been taken into account. Typically, for decision problems (e.g., satisfiability, credulous, or skeptical reasoning) the result is directly available at the root node. For enumeration tasks the tree is traversed a second time (now in pre-order) and the partial solutions associated with each tree decomposition node are combined in order to obtain the complete solutions. Here, we follow the extension pointer tuples while combining

the contents of the respective item sets. For our running example, considering SAT, we have $\{\{x, c_1, c_2\}, \{x, z, c_1, c_2\}, \{y, z, c_1, c_2\}, \{x, y, z, c_1, c_2\}\}$. The models of ϕ_{Ex} are $\{\{x\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$, and the subset-minimal models are $\{\{x\}, \{y, z\}\}$.

```
length(2). or(0). and(1).
childBag(R,X) ← atNode(R,N), childNode(N), bag(N,X).
% Define false atoms and unsatisfied clauses
f(R,X)      ← childBag(R,X),           not childItem(R,X).
f(S,X)      ← childBag(R,X), sub(R,S), not childItem(S,X).
unsat(R,C) ← childBag(R,C),           not childAuxItem(R,C).
unsat(S,C) ← childBag(R,C), sub(R,S), not childAuxItem(S,C).
% Guess root-to-leaf paths in item trees to be extended
1 { extend(0,R) : rootOf(R,N) } 1 ← childNode(N).
1 { extend(L+1,S) :  sub(R,S) } 1 ← extend(L,R), L<2.
% Guess truth value of introduced atoms
{ item(2,A;1,A) : atom(A), introduced(A) }.
% Only join root-to-leaf paths coinciding on atom truth values
← extend(L,X;L,Y), atom(A), childItem(X,A), f(Y,A), L=1..2.
% Paths with unsatisfied, removed clauses are not extended
← extend(L,R), clause(C), removed(C), unsat(R,C), L=1..2.
% True atoms and satisfied clauses are kept
item(L,X)    ← extend(L,R), childItem(R,X), current(X), L=1..2.
auxItem(L,C) ← extend(L,R), childAuxItem(R,C), current(C),
    L=1..2.
% Through guess, clauses may become satisfied
auxItem(L,C) ← current(C;A), pos(C,A),     item(L,A), L=1..2.
auxItem(L,C) ← current(C;A), neg(C,A), not item(L,A), L=1..2.
% Interpretation at level 2 must be subset of that at level 1
← atom(A), item(2,A), not item(1,A).
% Update subset information; reject larger models at root
auxItem(2,smaller) ← extend(2,S), childAuxItem(S,smaller).
auxItem(2,smaller) ← atom(A), item(1,A), not item(2,A).
reject ← final, auxItem(2,smaller).
accept ← final, not reject.
```

Listing 2. D-FLAT encoding for solving $\subseteq$-MINIMAL SAT.

3 Dynamic Programming with Implicit Subset Minimization

3.1 Technical Outline

There are some issues with DP algorithm specifications involving subset optimization. In particular, the development of D-FLAT^2 is motivated by the following observations.

1) Counter candidates are constructed similarly to solution candidates.
2) Typically, counter candidates are also stored as solution candidates.

3) Counter candidates that are also solution candidates can be omitted if the respective solution candidate turns out to be no solution.

Our approach avoids redundant computations of solution and counter candidates. It supports minimization and maximization on user-specified items (e.g., for $\subseteq$-MINIMAL SAT, on atoms but not on clauses). Here, so-called *reduced item trees* serve as the main data structure, which are item trees of depth 1 that can store additional information at each node n: Besides extension pointer tuples, n has an *optimization item set*, which contains the items on which subset optimization is performed. D-FLAT^2 thus defines the new output predicate `optItem`/1, where `optItem`(S) means that item S is subject to optimization. Instead of using item trees whose nodes at depth 2 represent counter candidates, n contains a set of *counter candidate pointers*, which are hidden from the user. A counter candidate pointer is a pair (c, s), where c is a reference to a sibling of n and s is a Boolean flag that is set to true iff there is an extension of c whose optimization items form a proper subset (or superset for maximization problems) of those of each extension of n (similar to the `smaller` predicate in Listing 2).

Opposed to classical implementations such as the algorithm for $\subseteq$-MINIMAL SAT described in Section 2, we perform two bottom-up traversals of the TD: In the first traversal, we compute all reduced item trees as in classical implementations, but only up to depth 1. In the second traversal, we add counter candidates to nodes at depth 1 appropriately, but instead of creating children that are copies of other item sets from depth 1, we store only counter candidate pointers to already existing item sets. Technical details are given in [2], whereas we here give an application-oriented introduction.

Program $\Pi_{\texttt{optAllItems}} = \{\texttt{optItem(X)} \leftarrow \texttt{item(X)}\ .\}$ uses the `optItem`/1 predicate in a trivial way. With this, we can obtain the D-FLAT^2 encoding $\Pi_{\subseteq\text{-MINIMAL SAT}} = \Pi_{\texttt{optAllItems}} \cup \Pi_{\text{SAT}}$, which allows us to solve $\subseteq$-MINIMAL SAT by simply using the existing encoding for SAT and adding information on what to optimize. In this case, the basic specification $\Pi_{\texttt{optAllItems}}$ suffices, as in $\subseteq$-MINIMAL SAT we consider all items for minimization. D-FLAT^2 encoding $\Pi_{\subseteq\text{-MINIMAL SAT}}$ gives several advantages over the traditional D-FLAT encoding (see Listing 2): It allows us to again use the simplified table-based interface and the overall length of the encoding is greatly reduced.

For some problems, we require counter candidates that are not solution candidates at the same time. An example will be given in Section 3.2, where we consider disjunctive ASP. There, counter candidates are models of a program reduct and not the program itself. In such cases, the item tree nodes representing such counter candidates can be marked with a special flag as "pseudo solution candidates" by means of the atom `auxItem(pseudo)`, which is taken into account by D-FLAT^2. Similar to the traditional D-FLAT methodology, the only thing required from the user of D-FLAT^2 is an ASP encoding where each model corresponds to a single solution or counter candidate.

3.2 Application to Common AI Problems

Circumscription. In the propositional case of Circumscription [15], we are given a theory T and sets of atoms P and Z, and we are interested in models M of T such that there is no model M' with $M' \cap P \subset M \cap P$ and $M \cap Z = M' \cap Z$. The formula whose classical models correspond to exactly those solutions is denoted by $\text{CIRC}(T; P; Z)$.

We can model Circumscription in our approach by a slight modification of our $\subseteq$-MINIMAL SAT algorithm: We only put an atom x in an optimization item set if $x \in P$; and for any $x \in Z$ we add optimization items $t(x)$ or $f(x)$ if the item set contains x or not, respectively (thus making solution candidates with different interpretations of Z incomparable). By applying this technique we have that for any sibling nodes n and n' in an item tree, the optimization item set of n is a subset of the one of n' iff $M_n \cap P \subseteq M_{n'} \cap P$ and $M_n \cap Z = M_{n'} \cap Z$, where M_n is the partial interpretation for node n following extension pointers. The program $\Pi_{\texttt{optForCirc}}$ (Listing 3) takes these considerations into account; moreover, $\Pi_{\text{CIRC}} = \Pi_{\texttt{optForCirc}} \cup \Pi_{\text{SAT}}$ is a D-FLAT^2 implementation of the DP algorithm for Circumscription. As input it expects the theory T to be given as a CNF formula like for Π_{SAT} and the sets P and Z to be given using the unary predicates `p` and `z`, respectively.

```
optItem(X)    ← item(X), p(X).
optItem(t(X)) ← item(X), z(X).
optItem(f(X)) ← not item(X), z(X).
```

Listing 3. $\Pi_{\texttt{optForCirc}}$: used for solving Circumscription via $\Pi_{\text{CIRC}} = \Pi_{\texttt{optForCirc}} \cup \Pi_{\text{SAT}}$.

Disjunctive ASP. While a traditional TD-based DP algorithm for solving disjunctive ASP can be found in [12], here we solve the problem with D-FLAT^2. We first do so via reduction to Circumscription. In the following, for any interpretation, rule or set of atoms X, we write X' to denote the result of replacing each atom a in X with a new atom a'. Given a disjunctive logic program Π consisting of rules of the form $a_1 \vee a_2 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$, we use the notation $h(r) := \bigvee_{1 \leq i \leq k} a_i$, $b^+(r) := \bigwedge_{1 \leq i \leq m} b_i$ and $b^-(r) := \bigwedge_{m+1 \leq i \leq n} \neg b_i$.

As shown in [17], an interpretation I is an answer-set of Π iff $I \cup I'$ is a model of $\bigwedge_{p \in HB(\Pi)}(p \equiv p') \wedge \text{CIRC}(\bigcup_{r \in \Pi}\{((b^+(r) \wedge b^-(r')) \rightarrow h(r))\}; HB(\Pi); HB(\Pi)')$, where $HB(\Pi)$ denotes the Herbrand base of Π. In order to compute models of this formula, we can first calculate models of the Circumscription part and then remove those models (using the "`pseudo`" item for marking pseudo solution candidates), where the truth value of some atom a is different from the one of a'. This amounts to $\Pi_{\texttt{pseudoForASP}}$ (Listing 4), which can be used for solving disjunctive ASP by means of the combined program $\Pi_{\texttt{ASP}} = \Pi_{\texttt{pseudoForASP}} \cup \Pi_{\text{CIRC}}$. The predicate `cor` in this encoding is assumed to be symmetric and occurring in input facts in order to associate each atom a with its corresponding primed variant a'. (We require that a and a' always occur together in some bag, which can be achieved by adding an edge (a, a') to the input graph.)

```
auxItem(pseudo) ← cor(A,B), current(A;B), item(A), not item(B).
auxItem(pseudo) ← extend(R), childAuxItem(R,pseudo).
```

Listing 4. $\Pi_{\texttt{pseudoForASP}}$: used for solving disjunctive ASP via $\Pi_{\texttt{ASP}} = \Pi_{\texttt{pseudoForASP}} \cup \Pi_{\text{CIRC}}$.

In Listing 5, we present an alternative approach to solving disjunctive ASP, which does not resort to Circumscription. In this encoding, $\Pi'_{\texttt{ASP}}$, we generate solution candidates for all interpretations that are candidates for being a classical model of the input

program, which is specified by means of the predicates `head`, `pos` and `neg`. A classical model M of a program P might be no answer set because some $M' \subset M$ is a model of the reduct P^M. To check this, we generate additional item tree nodes that only serve as counter candidates (like M') to the nodes representing classical model candidates (like M). For any atom a from the current bag, if an item set contains a, then the corresponding interpretation sets a to true (otherwise to false). The item tree nodes representing counter candidates can additionally contain items of the form $\texttt{r}(a)$. This signifies that the atom a is false in the respective counter candidate but true in the classical model candidates that reference this counter candidate (by means of their counter candidate pointers). In Lines 22 and 24, we make sure that any item tree node containing an item $\texttt{r}(a)$ is marked with the "`pseudo`" item and will therefore not be considered as a solution but rather serves as a counter candidate only. Lines 27 and 28 are required to ensure that any counter candidate C of any solution candidate M only contains $r(a)$ for atoms a that are also contained in M.

```
1  1 { extend(R) : childRow(R,N) } 1 ← childNode(N).
2  % Guess truth value/rule flag of introduced atoms
3  0 { item(A;r(A)) : atom(A), introduced(A) } 1.
4  % Make explicit when an atom is false or a rule is unsat
5  false(R,X)  ← childRow(R,N), bag(N,X), not childItem(R,X).
6  falser(R,X) ← childRow(R,N), bag(N,X), not childItem(R,r(X)).
7  unsat(R,X)  ← childRow(R,N), bag(N,X), not childAuxItem(R,X).
8  % Only join child item sets that coincide on common atoms
9  ← extend(X;Y), atom(A), childItem(X,A),    false(Y,A).
10 ← extend(X;Y), atom(A), childItem(X,r(A)), falser(Y,A).
11 % Only extend child item sets satisfying all removed rules
12 ← extend(S), rule(X), removed(X), unsat(S,X).
13 % True atoms and satisfied rules remain so unless removed
14 item(X)    ← extend(S), childItem(S,X),    current(X).
15 item(r(X)) ← extend(S), childItem(S,r(X)), current(X).
16 % Through the guess, rules may become satisfied
17 auxItem(R) ← current(R;A), head(R,A),     item(A).
18 auxItem(R) ← current(R;A), pos(R,A),  not item(A).
19 auxItem(R) ← current(R;A), neg(R,A),      item(A).
20 % Rule is not in reduct if a negative body atom is set to true
21 auxItem(R) ← current(R;A), neg(R,A),      item(r(A)).
22 auxItem(pseudo) ← item(r(X)), current(X).
23 % Inherit pseudo flag from child nodes
24 auxItem(pseudo) ← extend(R), childAuxItem(R,pseudo).
25 optItem(S)    ← atom(S), item(S).
26 % Prevents r(S) at level 2 (reduct) if S is not true at level 1
27 optItem(r(S)) ← atom(S), item(S), not auxItem(pseudo).
28 optItem(r(S)) ← atom(S), item(r(S)).
```

Listing 5. $\Pi'_{\texttt{ASP}}$: D-FLAT^2 encoding for solving disjunctive ASP directly.

Abstract Argumentation. Problems from abstract argumentation [5] are further examples where our approach is reasonable. Given an object (A, R), where A is a set of arguments and $R \subseteq A \times A$, we call a set $S \subseteq A$ *admissible* if (1) $(a, b) \notin R$ for all $a, b \in S$ and (2) for each $s \in S$ and $r \in A$, $(r, s) \in R$ implies that there is some $q \in S$ with $(q, r) \in R$. S is *preferred* if it is a subset-maximal admissible set. For any $C \subseteq A$, we call $C^+ = C \cup \{a \mid \exists b \in C \text{ s.t. } (b, a) \in R\}$ the *range* of C. A set S is *semi-stable* if it is admissible and for every admissible $S' \subset S$, $S^+ \not\subset S'^+$ holds.

Listing 6 shows an encoding $\Pi_{\texttt{admissible}}$ for computing admissible sets. At first glance, it seems to be overcomplicated compared to the algorithm in [6]: For computing admissible sets it actually suffices to guess which introduced arguments are in S, whereas we guess in $\Pi_{\texttt{admissible}}$ which arguments are in the set, attackers or neither. In order for a guessed set to be a solution, every attacker has to be defeated by the time it is removed from the bag. Once it can be determined that an attacker is defeated, its status changes from "`attc`" to "`def`". This additional complexity allows us to reuse $\Pi_{\texttt{admissible}}$ for computing semi-stable sets later on.

```
1 1 { extend(R)   : childRow(R,N) } 1 ← childNode(N).
2 % Guess whether an element is in, out or attacking (attc)
3 0 { item(in(A)); attc(A) } 1        ← introduced(A).
4 % Join only arguments with compatible status
5 nIn(S,A)  ← childRow(S,N), bag(N,A), not childItem(S,in(A)).
6 nDef(S,A) ← childRow(S,N), bag(N,A), not
      childAuxItem(S,def(A)), not childAuxItem(S,attc(A)).
7 ← extend(S1), extend(S2), childItem(S1,in(A)),       nIn(S2,A).
8 ← extend(S1), extend(S2), childAuxItem(S1,def(A)),   nDef(S2,A).
9 ← extend(S1), extend(S2), childAuxItem(S1,attc(A)), nDef(S2,A).
10 % Inherit arguments that are in, defeated or attackers
11 item(in(A)) ← extend(S), childItem(S,in(A)),       current(A).
12 chdef(A)    ← extend(S), childAuxItem(S,def(A)),  current(A).
13 attc(A)     ← extend(S), childAuxItem(S,attc(A)), current(A).
14 % Set defeated arguments
15 auxItem(def(A))  ← current(A;B), att(B,A), item(in(B)).
16 auxItem(def(A))  ← chdef(A).
17 % Still remaining (undefeated) attackers
18 auxItem(attc(A)) ← attc(A), not auxItem(def(A)).
19 % Out-arguments are not allowed to be defeated/attackers
20 out(A) ← not attc(A), not chdef(A), current(A).
21 ← auxItem(def(A)), out(A).
22 ← out(A), current(A), item(in(B)), att(A,B).
23 % Assure that the set is conflict-free
24 ← item(in(A)), item(in(B)), att(A,B).
25 % Remove candidates that leave attackers undefeated
26 ← extend(S), childAuxItem(S,attc(A)), removed(A).
```

Listing 6. $\Pi_{\texttt{admissible}}$: D-FLAT encoding for admissible sets.

Given $\Pi_{\texttt{admissible}}$, we can compute preferred sets by simple subset maximization via $\Pi_{\texttt{preferred}} = \Pi_{\texttt{optAllItems}} \cup \Pi_{\texttt{admissible}}$. Furthermore, it is now also easy to compute semi-stable sets by means of $\Pi_{\texttt{optForSemiStable}}$ (Listing 7), which gives us $\Pi_{\texttt{semiStable}} = \Pi_{\texttt{optForSemiStable}} \cup \Pi_{\texttt{admissible}}$. Although the guess from $\Pi_{\texttt{admissible}}$ could have been simplified for the previous examples, here it is indeed required because we need to find those admissible sets that have maximal range.

```
1 optItem(A) ← item(in(A)).
2 optItem(A) ← auxItem(attc(A)).
3 optItem(A) ← auxItem(def(A)).
```

Listing 7. $\Pi_{\texttt{optForSemiStable}}$: used for computing semi-stable sets via $\Pi_{\texttt{semiStable}} = \Pi_{\texttt{optForSemiStable}} \cup \Pi_{\texttt{admissible}}$.

4 System and Evaluation

In this section we focus on experiments for problems from the area of abstract argumentation. We compared systems that implement DP on TDs, namely D-FLAT 1.0.2 and D-FLAT^2 1.0.2. Furthermore, we benchmarked the ASPARTIX system [7] that solves argumentation-related problems directly via ASP. D-FLAT and D-FLAT^2 internally use ASP grounder Gringo 4.4.0 and solver Clasp 3.1.1. The results for ASPARTIX were produced with Gringo 3.0.5, as it is not fully compatible with newer versions of Gringo. For evaluation we used so-called "grid-based" instances, where vertices are arranged on a $n \times m$ matrix, and edges connect horizontally, vertically and diagonally neighboring vertices. Each instance was run five times with different TDs, and every run was limited to one hour and three GB of memory.

System comparison. We considered the problem of enumerating all preferred extensions and compared the systems on grid-based instances with 40 to 65 nodes and treewidth 4. Figure 4 illustrates average runtimes and allocated memory together with the 95 % confidence interval. D-FLAT^2 showed the best performance, while D-FLAT is slightly slower and requires more memory. For ASPARTIX we observed timeouts for instances having more than 55 nodes.

Problem comparison. As D-FLAT^2 is based on D-FLAT, we compared these systems on several problems using grid-based instances with treewidth 4. Moreover, we analyzed the cost of computing preferred and semi-stable sets compared to only obtaining admissible sets. As instances have much more admissible sets than preferred sets, which would bias a performance comparison when doing explicit enumeration, we considered the counting variants of these problems. Results are summarized in Figure 5.

When counting admissible sets, D-FLAT^2 requires slightly more time and memory than D-FLAT due to the overhead imposed by using reduced item trees instead of item trees. For preferred sets, the inefficiency of computing redundant counter candidates in D-FLAT becomes evident. On the contrary, in D-FLAT^2 the difference in runtime for counting preferred instead of admissible sets is barely measurable (i.e. within the 95% confidence interval). Here, we observed that subset maximization comes for free for

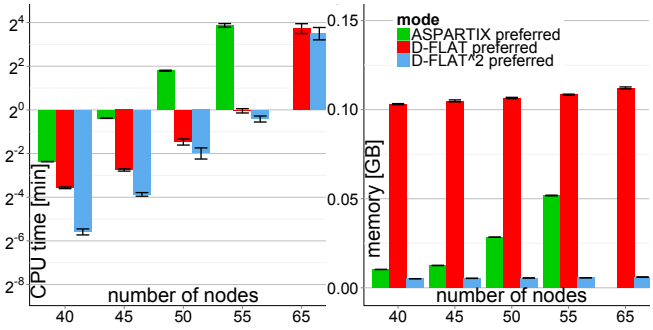

Fig. 4. System comparison: Average CPU time (left) and maximum resident set (right).

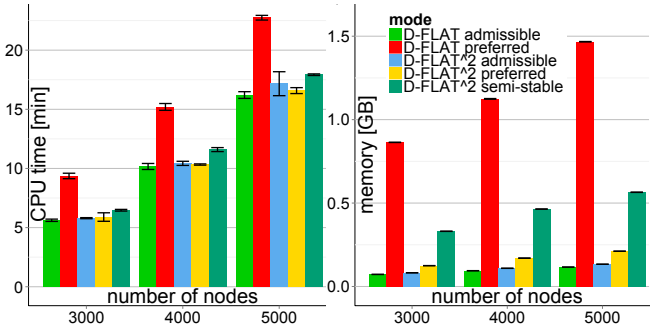

Fig. 5. Problem comparison: Average CPU time (left) and maximum resident set (right).

instances of small treewidth. (We also observed this effect in a comparison of SAT with $\subseteq$-MINIMAL SAT, where the overhead of D-FLAT was even larger.) Finally, for semi-stable sets, D-FLAT was not able solve instances with 500 vertices within the given memory limits. One reason is that for this problem many potential counter candidates have to be computed that turn out to be not even admissible. Thus, our two-phased approach of first computing (not necessarily maximal) solutions and then performing maximization obviously pays off in this case.

5 Conclusion

In this work we presented the D-FLAT^2 system for DP on TDs in order to solve problems involving subset optimization. Users of D-FLAT^2 are only required to provide an ASP program that specifies an algorithm for a version of the problem without optimization. Our method then performs the optimization tasks in an automatic and uniform way, thus making the development of such algorithms significantly easier. We have outlined the underlying ideas of the system and applied it to several problems by giving appropriate ASP encodings. Preliminary experiments indicate that our new approach brings significant advantages in terms of time and memory compared to previous solutions. In the future, we will apply our approach to further problems in order to improve our

implementation. Moreover, we plan to extend our approach to problems that are even higher in the polynomial hierarchy than the second level.

Acknowledgments: This work has been supported by the Austrian Science Fund (FWF): Y698, P25607, P25518.

References

1. Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT system for dynamic programming on tree decompositions. In *Proc. JELIA*, volume 8761 of *LNCS*, pages 558–572, 2014.
2. Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran. Optimization of tree-decomposition-based dynamic programming for AI problems. Unpublished draft. Available at http://dbai.tuwien.ac.at/proj/dflat/dflat-squared-draft.pdf, 2015.
3. Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
4. Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.
5. Phan M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–357, 1995.
6. Wolfgang Dvořák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37, 2012.
7. Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. Answer-set programming encodings for argumentation frameworks. *Argument and Computation*, 1(2):147–177, 2010.
8. Thomas Eiter and Axel Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *TPLP*, 6(1-2):23–60, 2006.
9. Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex optimization in answer set programming. *TPLP*, 11(4-5):821–839, 2011.
10. Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artif. Intell.*, 174(1):105–132, 2010.
11. Georg Gottlob and Stefan Szeider. Fixed-parameter algorithms for artificial intelligence, constraint satisfaction and database problems. *Comput. J.*, 51(3):303–325, 2008.
12. Michael Jakl, Reinhard Pichler, and Stefan Woltran. Answer-set programming with bounded treewidth. In *Proc. IJCAI*, pages 816–822, 2009.
13. Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's theorem – a game-theoretic approach. *Discrete Optimization*, 8(4):568–594, 2011.
14. Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.
15. John McCarthy. Circumscription – a form of non-monotonic reasoning. *Artif. Intell.*, 13(12):27–39, 1980.
16. Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and its Applications. OUP, 2006.
17. David Pearce, Hans Tompits, and Stefan Woltran. Characterising equilibrium logic and nested logic programs: Reductions and complexity. *TPLP*, 9(5):565–616, 2009.
18. Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.
19. Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.