dynPARTIX - A Dynamic Programming Reasoner for Abstract Argumentation

Wolfgang Dvořák[1(✉)], Michael Morak[2], Clemens Nopp[2], and Stefan Woltran[2]

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria
wolfgang.dvorak@univie.ac.at
[2] Institute of Information Systems, Vienna University of Technology, Vienna, Austria

Abstract. The aim of this paper is to announce the release of a novel system for abstract argumentation which is based on decomposition and dynamic programming. We provide first experimental evaluations to show the feasibility of this approach.

1 Introduction

Argumentation has evolved as an important field in AI, with abstract argumentation frameworks (AFs, for short) as introduced by Dung [5] being its most popular formalization. Several semantics for AFs have been proposed (see e.g. [2] for an overview), but here we shall focus on the so-called preferred semantics. Reasoning under this semantics is known to be intractable [6]. An interesting approach to dealing with intractable problems comes from parameterized complexity theory which suggests to focus on parameters that allow for fast evaluations as long as these parameters are kept small. One important parameter for graphs (and thus for argumentation frameworks) is tree-width, which measures the "tree-likeness" of a graph. To be more specific, tree-width is defined via a certain decomposition of graphs, the so-called tree decomposition. Recent work [7] describes novel algorithms for reasoning in the preferred semantics, such that the performance mainly depends on the tree-width of the given AF, rather than on the size of the AF. To put this approach to practice, we shall use the *SHARP* framework,[1] a C++ environment which includes heuristic methods to obtain tree decompositions [4], provides an interface to run algorithms on these decompositions, and offers further useful features, for instance for parsing the input. For a description of the *SHARP* framework, see [9].

The main purpose of our work here is to support the theoretical results from [7] with experimental ones. Therefore we use different classes of AFs and analyze the performance of our approach compared to an implementation based on answer-set programming (see [8]). Our prototype system together with the

This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT08-028, by the Austrian Science Fund (FWF): P20704-N18, and by the Vienna University of Technology program "Innovative Ideas".

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

H. Tompits et al. (Eds.): INAP/WLP 2011, LNAI 7773, pp. 259–268, 2013.
DOI: 10.1007/978-3-642-41524-1_14,

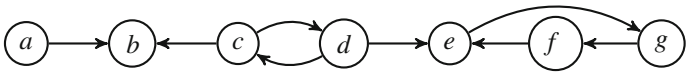
used benchmark instances is available as a ready-to-use tool from http://www.dbai.tuwien.ac.at/research/project/argumentation/dynpartix/.

2 Background

Argumentation Frameworks

An *argumentation framework* (AF) is a pair $F = (A, R)$ where A is a set of arguments and $R \subseteq A \times A$ is the attack relation. If $(a, b) \in R$ we say a attacks b. For $S \subseteq A$ and $a \in A$, we write $S \rightarrowtail a$ (resp. $a \rightarrowtail S$) iff there exists $b \in S$, such that $b \rightarrowtail a$ (resp. $a \rightarrowtail b$). An $a \in A$ is *defended* by a set $S \subseteq A$ iff for each $(b, a) \in R$, there exists a $c \in S$ such that $(c, b) \in R$. An AF can naturally be represented as a digraph.

Example 1. Consider the AF $F = (A, R)$, with $A = \{a, b, c, d, e, f, g\}$ and $R = \{(a, b), (c, b), (c, d), (d, c), (d, e), (e, g), (f, e), (g, f)\}$. The graph representation of F is given as follows:



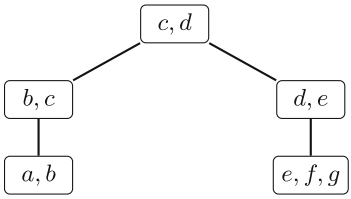
We require the following semantical concepts: Let $F = (A, R)$ be an AF. A set $S \subseteq A$ is (i) *conflict-free* in F, if there are no $a, b \in S$, such that $(a, b) \in R$; (ii) *admissible* in F, if S is conflict-free in F and each $a \in S$ is defended by S; (iii) a *preferred extension* of F, if S is a $\subseteq$-maximal admissible set in F. in Example 1, we get the admissible sets $\{\}, \{a\}, \{c\}, \{d\}, \{d, g\}, \{a, c\}, \{a, d\}$, and $\{a, d, g\}$. Consequently, the preferred extensions of this framework are $\{a, c\}, \{a, d, g\}$.

The typical reasoning problems associated with AFs are the following: (1) Credulous acceptance asks whether a given argument is contained in at least one preferred extension of a given AF; (2) skeptical acceptance asks whether a given argument is contained in all preferred extensions of a given AF. Credulous acceptance is NP-complete, while skeptical acceptance is even harder, namely Π_2^P-complete [6].

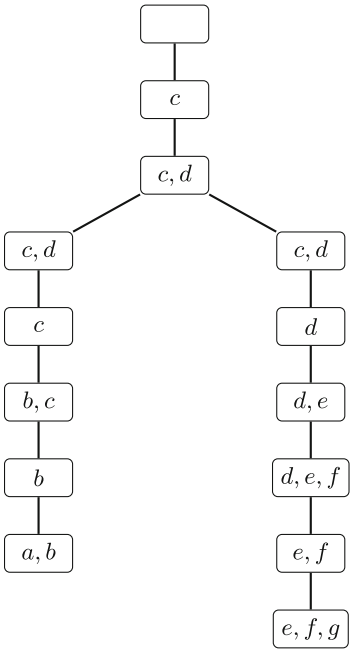
Tree Decompositions and Tree-Width

As already outlined, tree decompositions will underlie our implemented algorithms. We briefly recall this concept (which is easily adapted to AFs). A *tree decomposition* of an undirected graph $G = (V, E)$ is a pair $(\mathcal{T}, \mathcal{X})$ where $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is a tree and $\mathcal{X} = (X_t)_{t \in V_\mathcal{T}}$ is a set of so-called bags, which has to satisfy the following conditions:

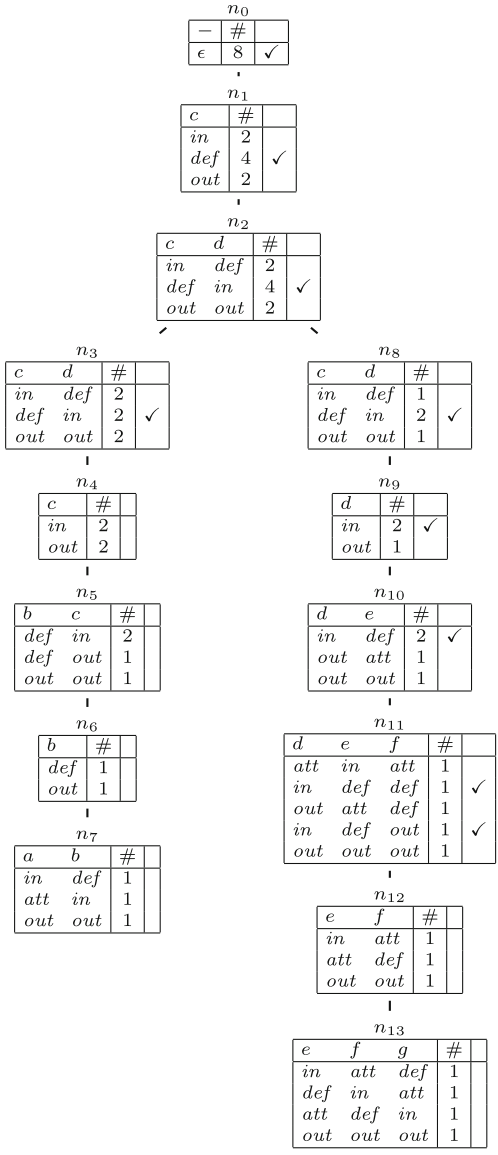
(a) $\bigcup_{t \in V_\mathcal{T}} X_t = V$, i.e. $\mathcal{X}$ is a cover of V;
(b) for each $v \in V$, the subgraph of $\mathcal{T}$ induced by $\{t \mid v \in X_t\}$ is connected;
(c) for each $\{v_i, v_j\} \in E$, $\{v_i, v_j\} \subseteq X_t$ for some $t \in V_\mathcal{T}$.



(a) Tree-decomposition of our running example.



(b) Normalized tree-decomposition of our running example.



(c) Computation of vcolorings for our running example on the given nice tree-decomposition

Fig. 1. Illustration of the algorithm for admissible sets.

The width of a tree decomposition is given by $\max\{|X_t| \mid t \in V_{\mathcal{T}}\} - 1$. The *tree-width* of G is the minimum width over all tree decompositions of G.

It can be shown that our example AF has tree-width 2 and we illustrate a tree decomposition of width 2 in Fig. 1(a).

However for our purposes we shall use so-called normalized decompositions, that is the tree-decomposition has a root node r with $X_r = \emptyset$ and consists only of nodes of one of the following types. A node $t \in V_{\mathcal{T}}$ is a:

- Leaf-node if t has no children nodes in $\mathcal{T}$;
- Branch-node if t has two successors t', t'' in $\mathcal{T}$ and $X_t = X_{t'} = X_{t''}$
- Insert-node if t has only one successor t' and $X_t = X_{t'} \cup v$ for some $v \in V$;
- Removal-node if t has only one successor t' and $X_t = X_{t'} \setminus v$ for some $v \in V$.

A normalized version of the first tree-decomposition is presented in Fig. 1(b).

Dynamic programming algorithms traverse such tree decompositions and compute local solutions for each node in the decomposition. Thus the combinatorial explosion is now limited to the size of the bags, that is, to the width of the given tree decomposition.

3 Dynamic Programming Algorithm

In this section we sketch the dynamic programming algorithm for credulous acceptance. For more detailed explanations as well as for the dynamic programming algorithm for skeptical acceptance, the interested reader is referred to [7]. In the following we tacitly assume an AF $F = (A, R)$ and a corresponding normalized tree-decomposition $(\mathcal{T}, \mathcal{X})$.

Towards an algorithm for admissible sets we need the following concept: A set of arguments E is a *B-restricted admissible set for* F, if E is conflict-free in F and E defends itself against all $a \in A \cap B$. Clearly we have that the A-restricted admissible sets coincide with the admissible sets. Now the main idea behind the dynamic programming algorithm is to consider for each node $t \in \mathcal{T}$, the AF $F_{\geq t}$ induced by the union of the bags $X_{\geq t}$ of the sub-tree of $\mathcal{T}$ rooted at t and computing the $X_{\geq t} \setminus X_t$-restricted admissible sets. As for the root note r, $X_r = \emptyset$ and $X_{\geq t} = A$ we then have a handle on the admissible sets.

Next we consider how we represent $(X_{\geq t} \setminus X_t)$-restricted admissible sets in a node t. First let us mention that by the definition of tree-decompositions we have that the AF $F_{\geq t}$ already contains all attacks incident with arguments in $X_{\geq t} \setminus X_t$ and thus we do not need the status of these arguments for the computation in the ancestor nodes of t. Hence for each node t it suffices to store the status of the arguments X_t for each $X_{\geq t} \setminus X_t$-restricted admissible set E. This is implemented by so called *vcolorings* $C_t : X_t \mapsto \{in, def, att, out\}$ for t with the following intuition: $C_t(a) = in$ iff $a \in E$; $C_t(a) = def$ iff $E \rightarrowtail a$; $C_t(a) = att$ iff $E \not\rightarrowtail a$ and $a \rightarrowtail E$; $C_t(a) = out$ iff $E \not\rightarrowtail a$ and $a \not\rightarrowtail E$. We have that each $(X_{\geq t} \setminus X_t)$-restricted admissible set corresponds to exactly one vcoloring, but one vcoloring in general corresponds to several $X_{\geq t} \setminus X_t$-restricted admissible sets.

In the following we discuss how to compute the vcolorings for each node-type. Starting with leaf-nodes t we are interested in $\emptyset$-restricted admissible sets of $F_{\geq t}$,

which coincide with the conflict-free sets. So we simply compute the conflict-free sets of $F_{\geq t}$ and map them to the corresponding vcolorings.

In a removal-node t with successor t' and $X_t = X_{t'} \setminus \{a\}$ we consider the successor's vcolorings $C_{t'}$ with $C_{t'}(a) \neq att$ and project them to X_t. We have that $C_{t'}(a) = att$ corresponds to a violation of admissibility, i.e. a attacks an argument in E and is not attacked by E, and as $a \in X_{\geq t} \setminus X_t$ the sets E corresponding to $C_{t'}$ are not $X_{\geq t} \setminus X_t$-restricted admissible.

Now let us focus on Insert node t with successor t' and $X_t = X_{t'} \cup \{a\}$. Again we consider vcolorings $C_{t'}$ of t'. Given a $X_{\geq t} \setminus X_t$ restricted admissible set E and adding a new argument a to the AF there are two ways to update E, either adding the new argument to E or not. This observation is mirrored by the following two operations. First we construct the vcoloring C_t^1 extending $C_{t'}$ to a such that a is labeled by one of the labels def, att, out, depending on the attacks between a and the arguments $\{a \in X_{t'} \mid C_{t'}(a) = in\} =: [C_{t'}]$. Moreover if $[C_{t'}] \cup \{a\}$ is conflict-free in F we also generate a vcoloring C_t^2 extending $C_{t'}$ such that $C_t^2(a) = in$ and faithfully update labels att, out according to attacks incident with a.

In a branch node t with successors t', t'' we union two sub-frameworks $F_{\geq t'}, F_{\geq t''}$ that intersect on X_t. Consequently to obtain an $(X_{\geq t} \setminus X_t)$-restricted admissible set of $F_{\geq t}$ we can combine each $(X_{\geq t'} \setminus X_{t'})$-restricted admissible set of $F_{\geq t'}$ with each $(X_{\geq t''} \setminus X_{t''})$-restricted admissible set of $F_{\geq t'}$ as long they coincide on X_t. Thus the vcolorings C of t are computed by combining vcolorings C' of t' and vcolorings C'' of t'' such that $[C'] = [C'']$. The coloring C computed from C',C'' is defined as follows. For $b \in X_t$ we have: $C(b) = in$ iff $C'(b) = C''(b) = in$; $C(b) = def$ iff $C'(b) = def$ or $C''(b) = def$; $C(b) = out$ iff $C'(b) = out$ and $C''(b) = out$; and $C(b) = att$ in the remaining cases.

Proposition 1. *For node t and $a \in X_t$. There is a vcoloring C_t for t with $C_t(a) = in$ iff a is contained in an $X_{\geq t} \setminus X_t$-restricted admissible sets of $F_{\geq t}$.*

Finally we discuss how credulous acceptance can be decided via vcolorings. We just mark each vcoloring which assigns the value *in* to the argument we are interested in and accordingly pass this mark up to the root. That is we mark a coloring if it is constructed by using at least one marked coloring. Finally at the (empty) root node we have that the argument is credulously accepted iff the vcoloring of the root is marked.

Example 2. Recall our running example, the computation of vcolorings is illustrated in Fig. 1(c). For deciding the credulous acceptance of argument d we mark vcolorings corresponding to at least one set containing d with a ✓, according to the above rules. The argument d is introduced two times, in the node n_3 and in the node n_{11}. Thus, we mark their vcolorings C satisfying $C(d) = in$. Now consider n_8 with the colorings $C_1(c) = in, C_1(d) = def$, $C_2(c) = def, C_2(d) = in$ and $C_3(c) = out, C_3(d) = out$. The child node n_9 has colorings $C_1'(d) = in$ and $C_2'(d) = out$, the first marked. As C_2 is constructed via C_1' it is also marked and as C_1 and C_3 are both constructed via C_2' they are not marked. ◊

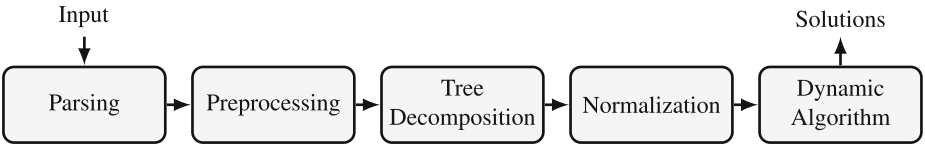


Fig. 2. Architecture of the *SHARP* framework.

4 Implementation and the *SHARP* Framework

dynPARTIX implements these dynamic programming algorithms based on tree decompositions using the *SHARP* framework [9], which is a purpose-built framework for implementing algorithms that are based on tree decompositions. Figure 2 shows the typical architecture, that systems working with the *SHARP* framework follow. In fact, *SHARP* provides interfaces and helper methods for the preprocessing and dynamic algorithm steps as well as ready-to-use implementations of various tree decomposition heuristics, i.e. Minimum-Fill, Maximum-Cardinality-Search and Minimum-Degree heuristics (cf. [4]), as well as different normalization algorithms.

As mentioned before, *dynPARTIX* builds on normalized tree decompositions provided by *SHARP*, which contain the four mentioned types of nodes. To implement our algorithms we just have to provide data structures storing the vcolorings of a node and the methods for each of these node types.

SHARP handles data-flow management and provides data structures where the calculated (partial) solutions to the problem under consideration can be stored. The amount of dedicated code for *dynPARTIX* comes to around 2700 lines in C++. Together with the *SHARP* framework (and the used libraries for the tree-decomposition heuristics), our system roughly comprises of 13000 lines of C++ code.

5 System Specifics

Currently the implementation is able to calculate the admissible and preferred extensions of a given argumentation framework and to check if credulous or skeptical acceptance holds for a specified argument. The basic usage of *dynPARTIX* is as follows:

```
> ./dynpartix [-f <file>] [-s <semantics>]
        [--enum | --count | --cred <arg> | --skept <arg>]
```

The argument `-f <file>` specifies the input file, the argument `-s <semantics>` selects the semantics to reason with, i.e. either admissible or preferred, and the remaining arguments choose one of the reasoning modes.

Input file conventions: We borrow the input format from the *ASPARTIX* system [8]. *dynPARTIX* thus handles text files where an argument a is encoded as

`arg(a)` and an attack (a, b) is encoded as `att(a,b)`. For instance, consider the following encoding of our running example and let us assume that it is stored in a file `inputAF`.

```
arg(a). arg(b). arg(c). arg(d). arg(e). arg(f). arg(g).
att(a,b). att(c,b). att(c,d). att(d,c).
att(d,e). att(e,g). att(f,e). att(g,f).
```

Enumerating extensions: First of all, *dynPARTIX* can be used to compute extensions, i.e. admissible sets and preferred extensions. For instance to compute the admissible sets of our running example one can use the following command:

```
> ./dynpartix -f inputAF -s admissible
```

Credulous Reasoning: *dynPARTIX* decides credulous acceptance using proof procedures for admissible sets (even if one reasons with preferred semantics) to avoid unnecessary computational costs. The following statement decides if the argument d is credulously accepted in our running example.

```
> ./dynpartix -f inputAF -s preferred --cred d
```

Indeed the answer would be *YES* as $\{a, d, g\}$ is a preferred extension.

Skeptical Reasoning: To decide skeptical acceptance, *dynPARTIX* uses proof procedures for preferred extensions which usually results in higher computational costs (but is unavoidable due to complexity results). To decide if the argument d is skeptically accepted, the following command is used:

```
> ./dynpartix -f inputAF -s preferred --skept d
```

Here the answer would be *NO* as $\{a, c\}$ is a preferred extension not containing d.

Counting Extensions: Recently the problem of counting extensions has gained some interest [1]. We note that our algorithms allow counting without an explicit enumeration of all extensions (thanks to the particular nature of dynamic programming; see also [10]). Counting preferred extensions with *dynPARTIX* is done by

```
> ./dynpartix -f inputAF -s preferred --count
```

6 Benchmark Tests

In this section we compare *dynPARTIX* with *ASPARTIX* [8], one of the most efficient reasoning tools for abstract argumentation (for an overview of existing argumentation systems see [8]). For our benchmarks we used randomly generated AFs of low tree-width. To ensure that AFs are of a certain tree-width we considered random grid-structured AFs. In such a grid-structured AF each argument

is arranged in an $n \times m$ grid and attacks are only allowed between neighbours in the grid (we used an 8-neighborhood here to allow odd-length cycles, which are crucial for the full complexity of preferred semantics). When generating the benchmark instances we varied the following parameters: the number of arguments from 25 to 500; the tree-width; and the probability that a possible attack is actually in the AF.

The benchmark tests were executed on an Intel®Core$^{\text{TM}}$2 CPU 6300@1.86GHz machine running SUSE Linux version 2.6.27.48. We generated a total of 4800 argumentation frameworks with varying parameters as mentioned above. The two graphs on the left-hand side compare the running times of *dynPARTIX* and *ASPARTIX* (using dlv) on instances of small treewidth (viz. 3 and 5). For the graphs on the right-hand side, we have used instances of higher width. Results for credulous acceptance are given in the upper graphs and those for skeptical acceptance in the lower graphs. The y-axis gives the runtimes in logarithmic scale; the x-axis shows the number of arguments. Note that the upper-left picture has different ranges on the axes compared to the three other graphs. We remark that the test script stopped a calculation if it was not finished after 300 s. For these cases we stored the value of 300 s in the database.

Interpretation of the Benchmark Results: We observe that, independent of the reasoning mode, the runtime of *ASPARTIX* is only minorly affected by the tree-width while *dynPARTIX* strongly benefits from a low tree-width, as expected by theoretical results [7].

For the *credulous acceptance* problem we have that our current implementation is competitive only up to tree-width 5. Considering Fig. 3(a) and (b), there is to note that for credulous acceptance *ASPARTIX* decided every instance in less than 300 s, while *dynPARTIX* exceeded this value in 4 % of the cases.

Now let us consider the *skeptical acceptance* problem. As mentioned before, skeptical acceptance is computationally much harder than credulous acceptance, which is reflected by the bad runtime behaviour of *ASPARTIX*. Indeed we have that for tree-width ≤ 5, *dynPARTIX* has a significantly better runtime behaviour, and that it is competitive on the whole set of test instances. As an additional comment to Fig. 3(c) and (d), we note that for skeptical acceptance, *dynPARTIX* was able to decide about 71 % of the test cases within the time limit, while *ASPARTIX* only finished 41 %.

Finally let us briefly mention the problem of *Counting preferred extensions*. On the one side we have that *ASPARTIX* has no option for explicitly counting extensions, so the best thing one can do is enumerating extensions and then counting them. It can easily be seen that this can be quite inefficient, which is reflected by the fact that *ASPARTIX* only finished 21 % of the test instances in time. On the other hand we have that the dynamic algorithms for counting preferred extensions and deciding skeptical acceptance are essentially the same and thus have the same runtime behaviour.

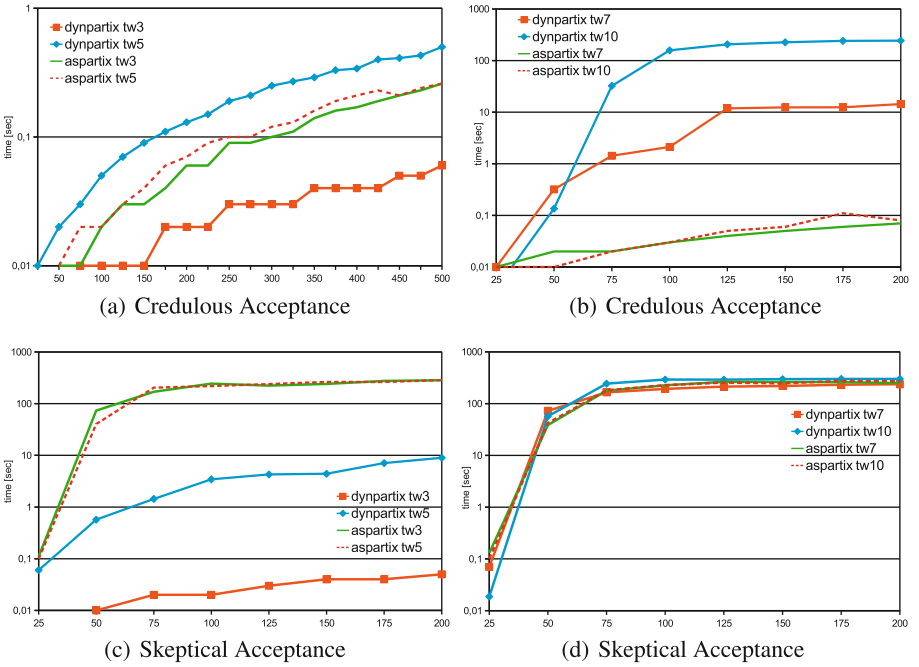


Fig. 3. Runtime of *dynPARTIX* for graphs of different tree-width compared to *ASPARTIX*.

7 Discussion

In this paper, we have presented a novel system for abstract argumentation which is based on decomposition and dynamic programming. Experimental evaluations show that such an approach is able to outperform systems relying on answer-set programming, at least on certain instances. This indicates that despite the high sophistication answer-set programming systems have reached nowadays, structural features of the problem instance are not sufficiently recognized yet by these systems. As ongoing work we thus focus on a combination of the both paradigms, i.e. decomposition and making use of declarative programming languages, see http://www.dbai.tuwien.ac.at/research/project/dynasp/dflat/ for further details.

For future work, we need a more comprehensive empirical evaluation, in particular on real world instances. To this end, we need more knowledge about the tree-width typical argumentation instances comprise, i.e. whether it is the case that such instances have low tree-width. Due to the unavailability of benchmark libraries for argumentation, so far we had to omit such considerations. we plan to extend *dynPARTIX* by additional argumentation semantics mentioned in [2] and by further reasoning modes, which can be efficiently computed on tree decompositions. Finally, we plan to further develop *dynPARTIX* for non-normalized tree decompositions [3].

References

1. Baroni, P., Dunne, P.E., Giacomin, M.: On extension counting problems in argumentation frameworks. In: Proceedings of the COMMA 2010, pp. 63–74 (2010)
2. Baroni, P., Giacomin, M.: Semantics of abstract argument systems. In: Rahwan, I., Simari, G.R. (eds.) Argumentation in Artificial Intelligence, pp. 25–44. Springer, Heidelberg (2009)
3. Charwat, G.: Tree-decomposition based algorithms for abstract argumentation frameworks. Master's thesis, TU Wien (2012)
4. Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B., Musliu, N., Samer, M.: Heuristic methods for hypertree decomposition. In: Gelbukh, A., Morales, E.F. (eds.) MICAI 2008. LNCS (LNAI), vol. 5317, pp. 1–11. Springer, Heidelberg (2008)
5. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. **77**(2), 321–358 (1995)
6. Dunne, P.E., Bench-Capon, T.J.M.: Coherence in finite argument systems. Artif. Intell. **141**(1/2), 187–203 (2002)
7. Dvořák, W., Pichler, R., Woltran, S.: Towards fixed-parameter tractable algorithms for abstract argumentation. Artif. Intell. **186**, 1–37 (2012)
8. Egly, U., Gaggl, S., Woltran, S.: Answer-set programming encodings for argumentation frameworks. Argument Comput. **1**(2), 147–177 (2010)
9. Morak, M.: SHARP - a smart hypertree-decomposition-based algorithm framework for parameterized problems. TU Wien. http://www.dbai.tuwien.ac.at/research/project/sharp/sharp.pdf (2010)
10. Samer, M., Szeider, S.: Algorithms for propositional model counting. J. Discrete Algorithms **8**(1), 50–64 (2010)