A Practical Account into Counting Dung's Extensions by Dynamic Programming

Ridhwan Dewoprabowo[1], Johannes Klaus Fichte[2(✉)], Piotr Jerzy Gorczyca[1], and Markus Hecher[2]

[1] TU Dresden, Dresden, Germany
piotr.gorczyca@tu-dresden.de
[2] TU Wien, Vienna, Austria
{johannes.fichte,markus.hecher}@tuwien.ac.at

Abstract. Abstract argumentation and Dung's framework are popular for modeling and evaluating arguments in artificial intelligence. We consider various counting problems in abstract argumentation under practical aspects. We revisit algorithms and establish a framework that employs dynamic programming on tree decompositions for counting extensions of abstract argumentation frameworks under admissible, stable, and complete semantics. We provide an empirical evaluation and investigate conditions under which our approach is useful.

1 Introduction

Abstract argumentation (Dung's framework) is a concept for modeling and evaluating arguments in AI and reasoning [3,8,24]. For finding so-called extensions to *abstract argumentation frameworks (AFs)*, a variety of solvers are available and frequently evaluated in competitions, e.g., ASPARTIX, ConArg, μ-toksia, and PYGLAF. Lately, interest in counting increased due to a variety of applications in probabilistic reasoning, reasoning about uncertainty, and verification. For example, abstract argumentation allows to establish cognitive computational models for human reasoning for which counting enables quantitative reasoning [7]. The recent 2021 ICCMA competition also asked for counting [22] despite being #P-hard. In propositional model counting, a system called DPDB [18] allows to effectively implement counting algorithms that exploit low primal treewidth of the input and proved competitive regardless of theoretical worst-case limitations. In fact, various problems in abstract argumentation can also be solved efficiently using dynamic programming on tree decompositions if the input has low treewidth[10]. Here, we consider various counting problems in abstract argumentation under practical aspects. Our main contributions are as follows.

1. We revisit theoretical algorithms and formulate abstract argumentation problems in relational algebra, which form the basis for our solver `A-DPDB`[1].

[1] System and supplement are available on github:gorczyca/dp_on_dbs and Zenodo.

G. Gottlob et al. (Eds.): LPNMR 2022, LNAI 13416, pp. 387–400, 2022.
https://doi.org/10.1007/978-3-031-15707-3_30

Fig. 1. An AF with the given attack relation (left) and a TD of the framework (right).

2. We establish a dedicated counting solver for counting extensions of AFs under admissible, stable, and complete semantics.
3. We provide an empirical evaluation and illustrate that `A-DPDB` works fine if combined with existing solvers.

2 Preliminaries

For a function f that maps from a set S to a set D, we let $\mathsf{dom}(f) := S$ be the *domain of* f. An *argumentation framework* [8] is a pair $F = \langle A, R\rangle$ where A is a set of arguments and $R \subseteq A \times A$ is an attack relation, representing attacks among arguments. We write $a \rightarrowtail b$ to denote an attack $(a, b) \in R$. In addition, for $S \subseteq A$, we denote $S \rightarrowtail a$ if there exists $b \in S$ such that $b \rightarrowtail a$; and $a \rightarrowtail S$ if $a \rightarrowtail b$, respectively. Further, for $S' \subseteq A$, we write $S \rightarrowtail S'$ if $S \rightarrowtail b'$ for some $b' \in S'$. Let $F = \langle A, R\rangle$ be an AF. A set $S \subseteq A$ is *conflict-free* (in F) if there are no $a, b \in S$, such that $a \rightarrowtail b$. An argument a is *defended* by S in F if for each $b \in A$ with $b \rightarrowtail a$ there exists a $c \in S$ such that $c \rightarrowtail b$. The semantics of our main interest are: (i) S is *admissible* if it is conflict-free in F and each $a \in S$ is defended by S in F. (ii) S is *stable* if it is conflict-free in F and for each $a \in A \setminus S$, there exists a $b \in S$, such that $b \rightarrowtail a$. (iii) S is *complete* if it is admissible in F and each $a \in A$ that is defended by S in F is contained in S.

Example 1. *Consider the AF from Fig. 1. We observe that* $\{a, c\}$ *and* $\{b, d\}$ *are admissible, stable, and complete sets. Further,* $\emptyset$ *is complete (admissible).* ◁

Tree Decompositions and Treewidth. We assume that the reader is familiar with basic graph terminology. We define the *tree decomposition, TD for short,* of a graph G as a pair $\mathcal{T} = (T, \chi)$, where T is a rooted tree and χ a function that assigns to each node $t \in V(T)$ a set $\chi(t) \subseteq V(G)$, called *bag*, such that (i) $V(G) = \bigcup_{t \in V(T)} \chi(t)$, (ii) $E(G) \subseteq \{\{u, v\} \mid t \in V(T), \{u, v\} \subseteq \chi(t)\}$, and (iii) for each $r, s, t \in V(T)$, such that s is a node in the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. We let $width(\mathcal{T}) := max_{t \in V(T)} |\chi(t)| - 1$ and define the *treewidth* $tw(G)$ of G as the minimum $width(\mathcal{T})$ over every TD $\mathcal{T}$ of G.

Example 2. *Consider the AF from Example 1. We can construct a TD illustrated in Fig. 1. Since the largest bag is of size* 3*, the TD has width* 2. ◁

To simplify case distinctions in the algorithms for sake of presentation, we assume *nice TDs* as given below. Our implementation does neither make an assumption on TDs being nice nor converts TDs into nice TDs. For a node $t \in V(T)$, $type(t)$

is defined as follows: *leaf* t has no children and $\chi(t) = \emptyset$; *join* if t has children t' and t'' with $t' \neq t''$ and $\chi(t) = \chi(t') = \chi(t'')$; *intr* ("introduce") if t has a single child t', $\chi(t') \subseteq \chi(t)$ and $|\chi(t)| = |\chi(t')| + 1$; and *forget* ("forget") if t has a single child t', $\chi(t') \supseteq \chi(t)$ and $|\chi(t')| = |\chi(t)| + 1$. A tree decomposition is *nice* if for every node $t \in V(T)$, $type(t) \in \{leaf, join, intr, forget\}$. It is folklore, that a nice TD can be computed from a given TD $\mathcal{T}$ in linear time without increasing the width, assuming the width of $\mathcal{T}$ is fixed. Let $\mathcal{T} = (T, \chi)$ be a tree decomposition of an AF F and let $t \in T$. For a subtree of T that is rooted in t we define $X_{\geq t}$ as the union of all bags within this subtree. Moreover, $X_{>t}$ denotes $X_{\geq t} \setminus \chi(t)$. We also have the sub-framework in t, denoted by $F|_{\chi(t)}$ or F_t, consists of all arguments $x \in \chi(t)$ and the attack relations (x_1, x_2) where $x_1 \in \chi(t)$, $x_2 \in \chi(t)$ and $(x_1, x_2) \in R$ [10].

Relational Algebra. Our algorithms operate on sets of records, which can simply be seen as tables. It is well-known that operations on tables can consisely be described by relational algebra [6] forming the basis of *SQL (Structured Query Language)* [25]. We briefly recall basic definitions. A *column* a is of a certain finite *domain* $\mathbf{dom}(a)$. Then, a *row* r over set $\mathrm{col}(r)$ of columns is a set of pairs of the form (a, v) with $a \in \mathrm{col}(r), v \in \mathsf{dom}(a)$ such that for each $a \in \mathrm{col}(r)$, there is exactly one $v \in \mathsf{dom}(a)$ with $(a, v) \in r$. To *access* the value v of an attribute a in a row r, we sometimes write $r.a$, which returns the unique value v with $(a, v) \in r$. A *table* τ is a finite set of rows r over set $\mathrm{col}(\tau) := \mathrm{col}(r)$ of columns, using domain $\mathsf{dom}(\tau) := \bigcup_{a \in \mathrm{col}(\tau)} \mathsf{dom}(a)$. We define *renaming* of τ, given a set A of columns and a bijective mapping $m : \mathrm{col}(\tau) \to A$ with $\mathsf{dom}(a) = \mathsf{dom}(m(a))$ for $a \in \mathrm{col}(\tau)$, by $\rho_m(\tau) := \{(m(a), v) \mid (a, v) \in \tau\}$. In SQL, renaming can be achieved via the `AS` keyword. *Selection* of rows in τ according to a given equality formula φ over term variables $\mathrm{col}(\tau)$ is defined by $\sigma_\varphi(\tau) := \{r \mid r \in \tau, \varphi$ is satisfied under the induced assignment $r\}$. We abbreviate for binary $v \in \mathrm{col}(\tau)$ with $\mathsf{dom}(v) = \{0, 1\}$, v=1 by v and v=0 by $\neg v$. Selection in SQL is specified using keyword `WHERE`. Given a relation τ' with $\mathrm{col}(\tau') \cap \mathrm{col}(\tau) = \emptyset$. Then, we refer to the *cross-join* by $\tau \times \tau' := \{r \cup r' \mid r \in \tau, r' \in \tau'\}$. Further, a *$\theta$-join* according to φ corresponds to $\tau \bowtie_\varphi \tau' := \sigma_\varphi(\tau \times \tau')$. In SQL a θ-join can be achieved by specifying the two tables (cross-join) and the selection φ by means of `WHERE`. Assume a set $A \subseteq \mathrm{col}(\tau)$ of columns. Then, we let table τ *projected to* A be given by $\Pi_A(\tau) := \{r_A \mid r \in \tau\}$, where $r_A := \{(a, v) \mid (a, v) \in r, a \in A\}$. This can be lifted to *extended projection* $\dot{\Pi}_{A,S}$, additionally given a set S of expressions of the form $a \leftarrow f$, such that $a \in \mathrm{col}(\tau) \setminus A$, f is an arithmetic function that takes a row $r \in \tau$, and there is at most one such expression for each $a \in \mathrm{col}(\tau) \setminus A$ in S. Formally, we define $\dot{\Pi}_{A,S}(\tau) := \{r_A \cup r^S \mid r \in \tau\}$ with $r^S := \{(a, f(r)) \mid a \in \mathrm{col}(r), (a \leftarrow f) \in S\}$. SQL allows to specify projection directly after the keyword `SELECT`. Later, we use *aggregation by grouping* ${}_A G_{(a \leftarrow g)}$, where $a \in \mathrm{col}(\tau) \setminus A$ and a so-called *aggregate function* $g : 2^\tau \to \mathsf{dom}(a)$, which intuitively takes a table of (grouped) rows. Therefore, we let ${}_A G_{(a \leftarrow g)}(\tau) := \{r \cup \{(a, g(\tau[r]))\} \mid r \in \Pi_A(\tau)\}$, where $\tau[r] := \{r' \mid r' \in \tau, r \subseteq r'\}$. Therefore, we use for a set S of integers the function $g =$ `SUM` for summing up values in S. SQL uses projection (`SELECT`) to specify A and the function g, distinguished via the keyword `GROUP BY`.

Dynamic Programming on TDs. A solver based on *dynamic programming (DP)* evaluates a given input instance $\mathcal{I}$ in parts along a given TD of a graph representation G of the input. Therefore, the TD is traversed bottom up, i.e., in post-order. For each node t of the TD, the intermediate results are stored in a set τ_t of records, *table* for short. The tables are obtained by a local algorithm, which depends on the graph representation. The algorithm stores results of problem parts of $\mathcal{I}$ in τ_t, while considering only tables $\tau_{t'}$ for child nodes t' of t. Various solvers that use dynamic programming have been implemented in the past for SAT, ASP, or ELP. Tools that allow for meta techniques using ASP for the description of the DP algorithm including various semantics for abstract argumentation exist. However, these tools are not competitive and do not support counting problems. DPDB [18] is a tool that utilizes database management systems (DBMS) to efficiently perform table manipulation operations needed during DP, which otherwise need tedious manual implementation. Its successor NestHDB [21] uses abstractions and a different graph representation.

3 Utilizing Treewidth for AFs

First, we revisit existing DP algorithms for counting extensions of AFs under stable and admissible semantics [10]. From there, we formulate different cases of the DP algorithm in relational algebra and extend it to counting. Later, we illustrate that we can instantiate these relational algebras as SQL queries, which are however created dynamically. In a way, our algorithms present a concise generator for SQL queries. Above, we already described the main idea on traversing a TD and constructing tables. Below, we only provide the *table algorithms* that are executed in each step during the traversal depending on the semantics.

Stable Semantics. We start with the algorithm for stable semantics, which is less elaborate than the other semantics and hence easier to understand. We follow standard definitions [8].We start from describing "local solutions". An extension of an argumentation framework is a set $S \subseteq A$, which satisfies the conditions for stable semantics. When traversing the TD, the algorithm constructs partial extensions to the input framework according to the vertices that occur in the bag currently considered. Formally, we are interested in B-restricted stable sets. Therefore, assume that an argumentation framework $F = \langle A, R\rangle$ and the set $B \subseteq A$ of arguments are given. A set $S \subseteq A$ is a *B-restricted stable set* for F, if S is conflict-free in F and S attacks all $a \in B \setminus S$. Then, a partial extension can simply be that a vertex is known not to be in the set (in), not in the set (def) due to being defeated by the set, or potentially not in the set (out). More formally, a *(stable) coloring* at t for an $X_{>t}$-restricted stable set S is a mapping $C : \chi(t) \to \{\text{in}, \text{def}, \text{out}\}$ such that (i) $C(a) = \text{in}$ if $a \in S$; (ii) $C(a) = \text{def}$ if $S \rightarrowtail a$; and (iii) $C(a) = \text{out}$ if $S \not\rightarrowtail a$ and $a \notin S$. Next, we briefly describe the table algorithm. In order to concisely present and to restrict the number of case distinctions, we assume that the algorithm runs along a nice TD. In practice, we need to interleave the cases to obtain competitive runtime behavior. Otherwise, unnecessary copying operations would make the implementation practically

Listing 1: Table algorithm $\mathbb{S}(t, \chi(t), F_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for stable semantics on TDs.

In: Node t, bag $\chi(t)$, AF F_t, sequence $\langle \tau_1, \ldots, \tau_\ell \rangle$ of child tables. **Out:** Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t := \{\langle \emptyset, 1 \rangle\}$

2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**

3 $\quad \tau_t := \{\langle J \sqcup \{b \mapsto \text{def} \mid b \in J^{\text{out}}, J^{\text{in}} \rightarrowtail b\}, c \rangle \mid \langle I, c \rangle \in \tau_1, J \in \{I^+_{a \mapsto \text{in}}, I^+_{a \mapsto \text{out}}\}, J^{\text{in}} \not\rightarrowtail J^{\text{in}}\}$

4 **else if** type$(t) =$ *forget, and* $a \notin \chi(t)$ *is removed* **then**

5 $\quad \tau_t := \{\langle I^-_a, \Sigma_{\langle J, c \rangle \in \tau_1 : I^-_a = J^-_a, a \notin J^{\text{out}}} c \rangle \mid \langle I, \cdot \rangle \in \tau_1, a \notin I^{\text{out}}\}$

6 **else if** type$(t) =$ *join* **then**

7 $\quad \tau_t := \{\langle I_1 \sqcup \{b \mapsto \text{def} \mid b \in I_2^{\text{def}}\}, c_1 \cdot c_2 \rangle \mid \langle I_1, c_1 \rangle \in \tau_1, \langle I_2, c_2 \rangle \in \tau_2, I_1^{\text{in}} = I_2^{\text{in}}\}$

$S^-_s := S \setminus \{s \mapsto \text{in}, s \mapsto \text{def}, s \mapsto \text{out}\}$, $S^l := \{s \mid S(s) = l\}$, $S^+_s := S \cup \{s\}$, $S \sqcup D := \bigcup_{s \in \text{dom}(S) \setminus \text{dom}(D)} \{s \mapsto S(s)\} \cup D$.

infeasible. Table algorithm $\mathbb{S}$, as presented in Listing 1, details all cases needed for the stable semantics. Parts of tuples that talk about extensions are illustrated red and counters in green. Each table τ_t consist of rows of the form $\langle I, c \rangle$, where I is a *coloring at* t and c is an integer forming a *counter* storing the number of extensions. Leaf node t consist of an empty mapping (coloring) and counter 1. For an introduce node t with introduced variable $a \in \chi(t)$, we extend each coloring I of the child table to a coloring J that additionally includes a in its domain. Therefore, we guess colors for a and keep only well-defined colorings that are obtained after ensuring conflict-freeness and setting arguments to def accordingly. When forgetting an atom a at node t, the colorings of child tables are projected to $\chi(t)$ and counters summed up of colorings that are the same after projection. However, it is important to not consider colorings, where a is set to out in order to compute $X_{>t}$-restricted stable sets. For join nodes, we update def colorings (behaves like a logical "or") and multiply the counters of extensions that are colored "in" and coincide in terms of arguments.

Listing 2 naturally introduces the algorithm for stable semantics using relational algebra instead of set theory. For each node t, tables τ_t are pictured as relations, where τ_t distinguishes for each argument $x \in \chi(t)$ unique attributes x and d_x, also just called columns, with additional attributes depending on the problem at hand. So these two columns a and d_a are of type `BOOLEAN` for every argument $a \in \chi(t)$, where for columns (a, d_a) we have that $(0, 0)$ represents out, $(0, 1)$ represents def, and $(1, -)$ represents in where "–" refers to not setting the value at all. For leaf nodes t, we create a fresh empty table τ_t, cf., Line 1. When an argument a is introduced, we perform a Cartesian product with the previously computed table and guess for argument a whether it is in the extension or not. We ensure only well-defined colorings, i.e., conflict-freeness and we potentially update color def for all bag arguments. Further, for nodes t with type$(t) =$ *forget*, we ensure that the removed argument is not colored out, we project out the removed argument, and perform grouping in order to maintain the counter, since several rows of τ_1 might have the exact same coloring after projection in τ_t. For a join node t, we use extended projection and θ-joins, where

Listing 2: Table algorithm $\mathbb{S}(t, \chi(t), F_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for stable semantics.

In: Node t, bag $\chi(t)$, framework $F_t = (A_t, R_t)$, sequence $\langle \tau_1, \ldots, \tau_\ell \rangle$ of child tables. **Out:** Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t := \{\{(\text{cnt}, 1)\}\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t := \dot{\Pi}_{\chi(t), \bigcup_{b \in \chi(t)} \{d_b \leftarrow d_b \vee (\neg b \wedge [\bigvee_{(c,b) \in R_t} c])\}} (\tau_1 \bowtie_{\bigwedge_{(b,c) \in R_t} \neg b \vee \neg c} \{\{(a,1),(d_a,0)\},\{(a,0),(d_a,0)\}\})$
4 **else if** type$(t) =$ *forget, and* $a \notin \chi(t)$ *is removed* **then**
5 $\quad \tau_t := \{b, d_b \mid b \in \chi(t)\} G_{\text{cnt} \leftarrow \texttt{SUM}(\text{cnt})} (\Pi_{\text{col}(\tau_1) \setminus \{a, d_a\}} (\sigma_{a \vee \neg d_a} (\tau_1)))$
6 **else if** type$(t) =$ *join* **then**
7 $\quad \tau_t := \dot{\Pi}_{\chi(t), \bigcup_{b \in \chi(t)} \{\text{cnt} \leftarrow \text{cnt} \cdot \text{cnt}', d_b \leftarrow d_b \vee d'_b\}} (\tau_1 \bowtie_{\bigwedge_{b \in \chi(t)} b = b'} \rho_{\bigcup_{x \in \text{col}(\tau_2)} \{x \mapsto x'\}} \tau_2)$

we join on the coloring agreeing on those arguments in the extension, update defeated colors, and multiply the corresponding counters, accordingly.

Example 3 illustrates a resulting SQL query at an introduce node of the TD, where we interleave cases and drop the requirement on nice TDs.

Example 3. *Consider the TD from Example 2 at node h_1, which is both an introduce and forget node. Following Listing 2 for stable semantics, we obtain the SQL query below.*

```
SELECT a, b, d, d_a, d_b, d_d,
       sum(cnt) AS cnt
FROM (WITH introduce AS
    (SELECT true val UNION SELECT false)
 SELECT i_a.val AS a, i_b.val AS b,
        i_d.val AS d, d AS d_a,
        a AS d_b, false AS d_d, 1 AS cnt
  FROM introduce i_a, /*introduce a,b,d*/
   introduce i_b, introduce i_d) AS cand
WHERE (a OR d_a) AND /*forget a*/
  (NOT a OR NOT b) AND /*conflict-free*/
  (NOT d OR NOT a)
GROUP BY a, b, d, d_a, d_b, d_d
```

◁

Admissible Semantics. In the following subsection, we extend the algorithm presented above. We present colorings for the admissible semantics following earlier work [10]. Given an argumentation framework $\langle A, R \rangle$ and a set $B \subseteq A$ of arguments. A set $S \subseteq A$ is a *B-restricted admissible set* for F, if S is conflict-free in F and S defends itself in F against all $a \in B$. Based on this definition, we construct colorings that locally satisfy certain conditions allowing to extend them to a coloring of the entire framework, which in turn can then be used to construct an admissible set of arguments. To this end, assume for an argumentation framework F a TD $\mathcal{T} = (T, \chi)$ and a node t of T. Formally, an *(admissible) coloring* at t for an $X_{>t}$-restricted admissible set S is a mapping $C : \chi(t) \to \{\text{in}, \text{def}, \text{out}, \text{att}\}$ such that for each $a \in \chi(t)$: (i) $C(a) = \text{in}$ if

Listing 3: Table algorithm $\mathbb{A}(t, \chi(t), F_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for admissible semantics.

In: Node t, bag $\chi(t)$, framework $F_t = (A_t, R_t)$, sequence $\langle \tau_1, \ldots, \tau_\ell \rangle$ of child tables. **Out:** Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t := \{\{(\mathrm{cnt}, 1)\}\}$
2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**
3 $\quad \tau_t := \dot{\Pi}_{\chi(t), \bigcup_{b \in \chi(t)} \{d_b \leftarrow \mathsf{df}_t(d_b, b)\}} (\tau_1 \bowtie_{\bigwedge_{(b,c) \in R_t} \neg b \vee \neg c} \{\{(a,1),(d_a,0)\},\{(a,0),(d_a,0)\}\})$
4 **else if** type$(t) =$ *forget, and* $a \notin \chi(t)$ *is removed* **then**
5 $\quad \tau_t := {}_{\{b, d_b \mid b \in \chi(t)\}} G_{\mathrm{cnt} \leftarrow \mathtt{SUM(cnt)}} (\Pi_{\mathrm{col}(\tau_1) \setminus \{a, d_a\}} (\sigma_{a \vee d_a = 1} (\tau_1)))$
6 **else if** type$(t) =$ *join* **then**
7 $\quad \tau_t := \dot{\Pi}_{\chi(t), \bigcup_{b \in \chi(t)} \{\mathrm{cnt} \leftarrow \mathrm{cnt} \cdot \mathrm{cnt}', d_b \leftarrow \mathsf{jn}(d_b, d'_b)\}} (\tau_1 \bowtie_{\bigwedge_{b \in \chi(t)} b = b'} \rho_{\bigcup_{x \in \mathrm{col}(\tau_2)} \{x \mapsto x'\}} \tau_2)$

Let $\mathsf{jn}(d, e) := 2$ if $d{=}2$ or $e{=}2$; else 1 if $d{=}1$ or $e{=}1$; else 0, and $\mathsf{df}_t(d, b) := \mathsf{jn}(d, 2$ if $(\bigvee_{(c,b) \in R_t} c)$; else 1 if $(\bigvee_{(b,c) \in R_t} c)$; else 0).

$a \in S$; (ii) $C(a) = \mathrm{def}$ if $S \rightarrowtail a$; (iii) $C(a) = \mathrm{att}$ if $S \not\rightarrowtail a$ and $a \rightarrowtail S$; and (iv) $C(a) = \mathrm{out}$ if $S \not\rightarrowtail a$ and $a \not\rightarrowtail S$.

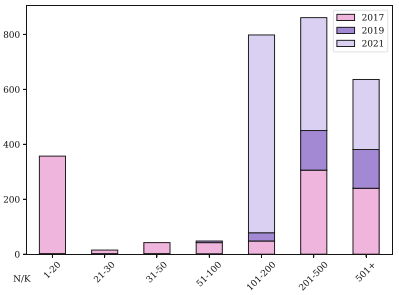
The algorithm to compute the admissible semantics extends the algorithm for stable semantics, as presented above. Intuitively, those arguments colored att need to become def eventually in order to obtain A-restricted admissible sets. In our implementation, we represent the range for the colorings in a database table with a `BOOLEAN` column a and a `SMALLINT` column d_a for every argument $a \in \chi(t)$. Then, $(0,0)$ represents out, $(0,1)$ represents att, $(0,2)$ represents def, and $(1,-)$ represents in. Alternatively, one can also exploit the `NULL` value in SQL, which reduces preallocated memory for the d_a columns as we can use the more compact data type `BOOLEAN` instead of `SMALLINT`. There, we have $(0, \mathtt{NULL})$ represents out, $(0,0)$ represents att, $(0,1)$ represents def, and $(1,-)$ represents in (as before).

The following example illustrates a query that we obtain at node h_1 of our running example similar to the used definition in relational algebra of Listing 2.

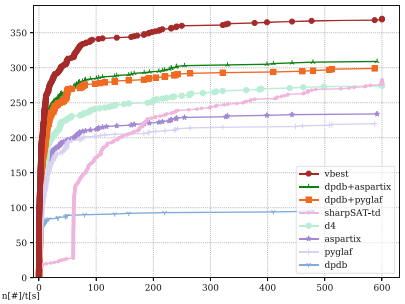
Example 4. *Consider the TD and introduce/forget node h_1 of our running Example 2. We construct a query for admissible extensions as follows.*

```
SELECT a, b, d, d_a, d_b, d_d, sum(cnt) AS cnt
FROM (WITH introduce AS
    (SELECT true val UNION SELECT false)
 SELECT i_a.val AS a, i_b.val AS b, i_d.val AS d,
    CASE WHEN i_d.val THEN 2/*coloring*/
     WHEN i_b.val THEN 1 ELSE 0 END AS d_a,
    CASE WHEN i_a.val THEN 2 ELSE 0 END AS d_b,
    CASE WHEN i_a.val THEN 1 ELSE 0 END AS d_d, 1 AS cnt
 FROM introduce i_a, /*introduce a,b,d*/
 introduce i_b, introduce i_d) AS cand
WHERE (a OR d_a = 1) AND /*forget a*/
  (NOT a OR NOT b) AND /*conflict-free*/
  (NOT d OR NOT a)
GROUP BY a, b, d, d_a, d_b, d_d
```

◁



(a) Distribution of heuristically computed widths. The x-axis lists intervals into which the heuristically computed width of a TD falls (K). The y-axis states the number (N) of instances.



(b) Runtime of various solvers for admissible semantics. The x-axis depicts the runtime sorted in ascending order for each solver individually and the y-axis refers to the number of instances.

Fig. 2. Illustration of results on ICCMA competitions '17, '19, and '21. Distribution of upper bounds on treewidth (left) and runtime results for admissible semantics (right).

Complete Semantics. Subsequently, we turn our attention to complete semantics. We provide definitions for colorings that can be used to construct solutions by dynamic programming and when its satisfying all conditions for the complete semantics [5]. Given an AF $F = \langle A, R\rangle$ and a set $B \subseteq A$ of arguments. A labeling $\mathcal{L} = \langle \mathcal{L}_{\text{in}}, \mathcal{L}_{\text{def}}, \mathcal{L}_{\text{out}}\rangle$ where $\mathcal{L}_{\text{in}}, \mathcal{L}_{\text{def}}, \mathcal{L}_{\text{out}} \subseteq A$ for F is a *B-restricted complete labeling* for F if $\mathcal{L}_{\text{in}}$ is conflict-free, $\mathcal{L}_{\text{in}} \not\rightarrowtail \mathcal{L}_{\text{out}}$, $\mathcal{L}_{\text{out}} \not\rightarrowtail \mathcal{L}_{\text{in}}$, and for each $a \in B$ we have (i) $a \in \mathcal{L}_{\text{in}}$ if and only if $\{b \mid (b, a) \in R\} \subseteq \mathcal{L}_{\text{def}}$; (ii) $a \in \mathcal{L}_{\text{def}}$ if and only if $\mathcal{L}_{\text{in}} \rightarrowtail a$; (iii) $a \in \mathcal{L}_{\text{out}}$ if and only if $\mathcal{L}_{\text{in}} \not\rightarrowtail a$ and $\mathcal{L}_{\text{out}} \rightarrowtail a$. Let $\mathcal{T} = (T, \chi)$ be a TD of F and t be a node of T. A *(complete) coloring at t* is a function $C_t : \chi(t) \rightarrow \{\text{in}, \text{def}, \text{defp}, \text{out}, \text{outp}\}$ such that for each $a \in \chi(t)$: (i) $C(a) = \text{in}$ if $a \in \mathcal{L}_{\text{in}}$; (ii) $C(a) = \text{def}$ if $a \in \mathcal{L}_{\text{def}}$ and $\mathcal{L}_{\text{in}} \rightarrowtail a$; (iii) $C(a) = \text{defp}$ if $a \in \mathcal{L}_{\text{def}}$ and $\mathcal{L}_{\text{in}} \not\rightarrowtail a$; (iv) $C(a) = \text{out}$ if $a \in \mathcal{L}_{\text{out}}$, $\mathcal{L}_{\text{in}} \not\rightarrowtail a$, $a \not\rightarrowtail \mathcal{L}_{\text{in}}$, and $\mathcal{L}_{\text{out}} \rightarrowtail a$; and (iv) $C(a) = \text{outp}$ if $a \in \mathcal{L}_{\text{out}}$, $\mathcal{L}_{\text{in}} \not\rightarrowtail a$, $a \not\rightarrowtail \mathcal{L}_{\text{in}}$, and $\mathcal{L}_{\text{out}} \not\rightarrowtail a$.

Intuitively, colors defp and outp are used to mark candidates for def and out. For such candidates, required properties need to be "proven" eventually. We further extended the algorithm of Listing 3 and implemented the handling of complete colorings. In our implementation, we represent the values for colorings in an SQL database table with a `SMALLINT` column a and a `BOOLEAN` column p_a for the "provability of the color of a", as follows: $(0, 1)$ represents out, $(0, 0)$ represents outp, $(1, -)$ stands for in, $(2, 1)$ represents def, and $(2, 0)$ states defp.

4 Preliminary Empirical Evaluation

In order to draw conclusions concerning the efficiency of our approach, we conducted a series of experiments. **Design of Experiment:** We draw a small experiment to study the following questions: (Q1.1) What are upper bounds on the treewidth for common instances in abstract argumentation? (Q1.2) Are

there instances on which we can expect that solvers exploiting treewidth perform well? (Q2.1) Does the parameterized algorithm perform well on instances of low treewidth? (Q2.2) Is there a certain characteristic on the instances where our solver performs better than others? (Q2.3) Is the system competitive on its own with other solvers or can it be useful in a solving portfolio? **Instances:** We considered sets of instances from the International Competitions on Computational Models of Argumentation ICCMA'17, '19, and '21. Since the hard instances of the 2019 competition are partially contained in the ICCMA'21 set, we omit the 2019 instances. In the following, we refer by '19 to the hard instances of the 2019 competition contained in the '21 competition and by '21 to the new instances of the '21 competition. The instances originate from various domains. Details can be found online [22]. **Constructing TDs:** To construct TDs, we use the decomposer that heuristically outputs tree decompositions. The outputted TDs are correct, but are not necessarily of smallest width, i.e., the width of the resulting TD can be larger than the treewidth. Note that computing the treewidth is itself an NP-complete problem. We do not require a tree decomposition of smallest width. Larger width w increases the runtime of our implementation, since the runtime is in 2^w. There is no effect on correctness with respect to the problem statement from taking decompositions of larger width. In practice, we favor a fast heuristic, namely, `htd`, over decomposers such as Flow-Cutter or TCS-Meiji that provide slightly smaller width, but require longer running times.

Treewidth Classification of the Instances. Towards answering (Q1.1) and (Q1.2), we investigate whether the considered instances are relevant and solvable for an approach where the runtime already theoretically depends on the width of the heuristically computed TDs. In Fig. 2a, we present the distribution of upper bounds on the treewidth in intervals of the considered instances by competition. Decompositions of smaller width can be primarily found in the '17 instances. Recall that our parameterized algorithms have single or double exponential runtime bounds in the treewidth [10]. Hence, we immediately see that the '19 and '21 instances are theoretically out of reach for `A-DPDB`. For the '19 and '21 instances, we are currently unable to state a detailed picture as high width might also originate in unreliable heuristics. It is well-known that certain heuristics cannot provide a small width on very large instances even if a much smaller width is possible. Still there is a notable number of instances in the 2017 competition, which seem within reach answering Questions (Q1.1) and (Q1.2). Quite a number of instances have width beyond 100. There, we have no hope to solve them by a treewidth-based approach without preprocessing or using abstractions instead of the primal graph. Still, quite a number of instances have relatively small treewidth and the instances of high treewidth mostly originate in random generators.

Performance Comparison and Solvers. In order to address a performance analysis of `A-DPDB` itself and in comparison to other argumentation solvers, we run a more detailed experiment. Counts are represented with arbitrary precision for all solvers. For comparison, we evaluate leading solvers of the ICCMA'21 competition. Namely, μ-`toksia` [23], `aspartix` [9], and `pyglaf` [1]. The solvers

μ-toksia, aspartix, pyglaf performed well during ICCMA'17, '19, and '21. In addition, we can employ state-of-the-art propositional model counters such as the model counting competition 2021 winner SharpSAT-td or d4 on encodings of the argumentation semantics of interest. Therefore, we can use the ASP encoding from aspartix[2] directly by lp2normal and lp2sat, which translates the ground ASP instance into a SAT formula. There is only a minimal overhead between a direct CNF encoding and an ASP encoding translated into CNF in case of the relevant encodings. In more detail, most ASP encodings here are tight and therefore *do not need additional constraints* to handle cyclic dependencies of the resulting programs as one might fear from translations into CNF. SharpSAT-td employs TDs of small width, but only as in a process to speed up its internal selection heuristic, which is in stark contrast to our approach that provides strict theoretical guarantees. SharpSAT-td implements dedicated preprocessing techniques for model counting from which a translation profits. To our knowledge dedicated preprocessing for argumentation is missing. In addition, SharpSAT-td uses FlowCutter as heuristic. Both techniques make the solver incomparable to ours. We did not consider NestHDB as the translation to SAT is not treewidth-aware. All solvers including A-DPDB support complete and stable semantics. Admissible semantics is not always available to the user even though implemented, e.g., μ-toksia. We refrained from modifying the solver.

Enhancing Existing Solvers. From the results above on our instance classification with respect to treewidth and our theoretical knowledge about the implemented parameterized algorithm, we must expect clear practical limitations of A-DPDB. Still, it might solve instances that existing techniques cannot solve. Therefore, we also consider A-DPDB together with other solvers, which is usually referred to as *portfolio solver*. However, classical solving portfolios are oftentimes detected based on machine-learning techniques that train for specific instances. Our setting is different, we can simply enhance an existing solver by using DP if a heuristically computed decomposition is below 19. We obtained this threshold experimentally from simple considerations on memory consumption. Our new solvers named A-DPDB+X consist of $X \in \{\texttt{aspartix}, \mu\texttt{-toksia}, \texttt{pyglaf}\}$.

Hardware, Measure, and Restrictions. All solvers ran on a cluster consisting of 12 nodes equipped with two Intel Xeon E-2650 v4 CPUs running at 2.2 GHz. We follow standard guidelines for empirical evaluations [20] and measure runtime using perf. Details on the hardware will be made available in the supplemental material. We mainly compare wall clock time and follow the setup of the International Competition on Computational Models of Argumentation (ICCMA). Run times larger than 600 s count as timeout and main memory (RAM) was restricted to 64 GB. In contrast to dedicated counting competitions the runtime in the setup of the ICCMA competition is much smaller, which is also far more resource friendly. Solvers were executed sequentially without any parallel execution of other runs, i.e., we jobs run exclusively on one machine.

[2] μ-toksia does not have encodings readily accessible as it is tightly coupled to a SAT solver. This would require extraction from source code or implementing it ourselves.

Table 1. Overview on solved instances (left) as well as observed counts (right).

solver	adm.	complete	stable
aspartix	236	362	469
... /d4[2]	347	406	483
... /sharpSAT-td[2]	*368*	*410*	*487*
dpdb	96	100	113
...+aspartix	**311**	**379**	475
...+μ-toksia21	95	367	468
...+pyglaf	300	372	**478**
μ-toksia21	–	299	446
pyglaf	221	336	463
sharpSAT-td	284	350	387
vbest	371	411	505

(a) Number of solved instances of various solvers. "–" indicates that the solver does not support the semantics. Bold entries indicate the best result, italic entries refer to the best result among non-portfolio solvers. [2] selects solvers also based on treewidth.

	adm.	complete	stable
median	2.9	0.5	0.0
mean	11.6	8.3	3.8
max	512.6	487.7	498.2
aspartix	7.9	8.3	8.7
dpdb	154.6	119.9	75.0
mu_toksia21	–	5.1	5.2
pyglaf	6.1	6.5	5.8
sharpSAT-td	512.6	487.7	498.2

(b) Observed counts. The lower part states the maximum count observed for the respective solver. Counts are stated in $\log_{10}$ format, meaning that 2.9 represents a count of about $0.794 \cdot 10^3$ whereas 516.6 represents about $3.98107 \cdot 10^{516}$.

Experimental Results. Table 1a lists the number of solved instances for various solvers, considered semantics, and over '17, '19, '21 competition instances. In addition, Fig. 2b visualizes the runtime behavior of various solvers for the admissible semantics. Table 1b illustrates the observed counts on the instances in terms of average and median of the computed count per semantics as well as the maximum count of an instance solved by solver. Notably, `A-DPDB` solved instances for which the decomposer constructed a TD of up to width 19 for complete, 35 for admissible, and 50 for stable semantics. For stable, few instances were solved where the heuristic computed TDs of width 99 containing few bags.

Discussion. When taking a more detailed look into the results, we observe that `aspartix`, μ`-toksia`, and `pyglaf` mostly solve instances that have a small number of solutions and perform overall quite well when the count is fairly low. This is not surprising, since each of the three solvers works by enumerating extensions, which can be quite expensive in practice. For all semantics, `A-DPDB` alone solves the least instances, but is perfectly suitable for enhancing existing solvers `A-DPDB+aspartix` and `A-DPDB+pyglaf`, respectively, solve the most instances. The solvers `d4` and `sharpSAT-td` can easily be used to solve abstract argumentation instances for various semantics. In fact, we see a reasonable performance on instances even if counts are larger. For admissible semantics, `sharpSAT-td` solves more instances than `aspartix` and `A-DPDB`, but much less instances than our system `A-DPDB+aspartix`. More precisely, `A-DPDB+aspartix` solves ≈24% instances more than `aspartix` and ≈10% more than `sharpSAT-td`. When considering a virtual configuration that takes the best result of `sharpSAT-td` and `aspartix` (`sharpSAT-td/aspartix`), we obtain the best result. It solves 22% and 35% more instances than `sharpSAT-td` and `aspartix` alone. Note this combination is a virtual best configuration, not a solving portfolio. For complete, we see an improvement of about 4%, 8%, and 21% more solved instances over `aspartix`, `sharpSAT-td`, and μ`-toksia`, respectively. `d4` and μ`-toksia` solve a

similar number of instances, however the former solves also instances that have high counts. For stable, we observe only 2% improvement of the portfolio, but it solves 30% more instances than `d4` and 21% more than `sharpSAT-td`.

Summary. In summary, `A-DPDB` alone has a very limited performance. The behavior was quite well expected from the results in the first part of our experimental evaluation. We expect this behavior, since DP profits significantly from preprocessing, which has to our knowledge not been investigated for argumentation. Our results show that estimating treewidth can provide useful insights into constructing a solving portfolio – regardless of the used solvers. In contrast to machine learning-based heuristics, which are commonly used in the automated reasoning community, we can statically decide which "subsolver" we take without a training phase on a subset of the existing instances. We expect that tightly coupling a #SAT solver into an argumentation solver would be successful.

5 Conclusion and Future Work

We present a practical approach to counting in abstract argumentation. Counting allows to take quantitative aspects of extensions into account. This enables us to quantify on extensions and comprehend also semantics that are sometimes considered problematic, e.g., admissible sets. Beyond, it facilitates reasoning stronger than brave and skeptical decisions [4,13,14,19]. We can ask for the relationship between total possible extensions and observed extensions (plausibility), which also forms the basis for probabilistic tasks. Our implementation `A-DPDB` is based on dynamic programming on TDs showing competitive behavior in a system that combines existing solvers with `A-DPDB`. While existing solvers can be used to count solutions by enumeration, we provide an approach that works by a compact representation and systematically splitting the search space. We also illustrate translating argumentation problems into propositional model counting showing notable performance. Since these solvers also implement dedicated simplification techniques for propositional counting, it opens the question whether argumentation semantics can benefit from argumentation specific preprocessing.

We expect that our work opens a variety of further directions. First, `A-DPDB` forms the basis for using more general graph representations (NestHDB), which showed notable performance gains in the propositional case also over established model counters [21]. In principle, DP works for problems on any level of the PH. While theoretical lower-bounds (under the exponential-time-hypothesis) suggest high runtime (depending on the level of the hierarchy) [12,15], parameters that combine treewidth with other approaches might be fruitful, e.g., [11]. Besides, counting might help to improve the reliability of existing systems [2,17]. From the performance of propositional model counters, which also include preprocessing, we expect notable speed up for argumentation specific preprocessing. Even though we executed `A-DPDB` sequentially, parallel execution is possible in principle, which could improve on larger instances of low treewidth [16].

Acknowledgements. Research was funded by the DFG through the Collaborative Research Center, Grant TRR 248 project ID 389792660, the BMBF, Grant 01IS20056_NAVAS, the Vienna Science and Technology Fund (WWTF) grant ICT19-065, and the Austrian Science Fund (FWF) grants P32830 and Y698.

References

1. Alviano, M.: The PYGLAF argumentation reasoner. In: ICLP 2017 (Technical Communications). OASICS, vol. 58, pp. 2:1–2:3, Dagstuhl (2017)
2. Alviano, M., Dodaro, C., Fichte, J.K., Hecher, M., Philipp, T., Rath, J.: Inconsistency proofs for ASP: the ASP - DRUPE format. TPLP **19**(5–6), 891–907 (2019)
3. Amgoud, L., Prade, H.: Using arguments for making and explaining decisions. AIJ **173**(3–4), 413–436 (2009)
4. Besin, V., Hecher, M., Woltran, S.: Utilizing treewidth for quantitative reasoning on epistemic logic programs. TPLP **21**(5), 575–592 (2021)
5. Charwat, G.: Tree-decomposition based algorithms for abstract argumentation framework. Master's thesis, TU Wien, Vienna, Austria (2012)
6. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
7. Dietz, E., Fichte, J.K., Hamiti, F.: A quantitative symbolic approach to individual human reasoning. In: Proceedings of CogSci 2022 (2022, to appear)
8. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. AIJ **77**(2), 321–357 (1995)
9. Dvořák, W., Rapberger, A., Wallner, J.P., Woltran, S.: ASPARTIX-V19 - an answer-set programming based system for abstract argumentation. In: Herzig, A., Kontinen, J. (eds.) FoIKS 2020. LNCS, vol. 12012, pp. 79–89. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39951-1_5
10. Dvořák, W., Pichler, R., Woltran, S.: Towards fixed-parameter tractable algorithms for abstract argumentation. AIJ **186**, 1–37 (2012)
11. Fandinno, J., Hecher, M.: Treewidth-aware complexity in ASP: not all positive cycles are equally hard. In: AAAI 2021, pp. 6312–6320. AAAI Press (2021)
12. Fichte, J.K., Hecher, M., Kieler, M.F.I.: Treewidth-aware quantifier elimination and expansion for QCSP. In: Simonis, H. (ed.) CP 2020. LNCS, vol. 12333, pp. 248–266. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_15
13. Fichte, J.K., Hecher, M., Meier, A.: Knowledge-base degrees of inconsistency: complexity and counting. In: AAA 2021, pp. 6349–6357. No. 7, The AAAI Press (2021)
14. Fichte, J.K., Hecher, M., Nadeem, M.A.: Plausibility reasoning via projected answer set counting–a hybrid approach. In: IJCAI 2022 (2022, to appear)
15. Fichte, J.K., Hecher, M., Pfandler, A.: Lower bounds for QBFs of bounded treewidth. In: LICS 2020, pp. 410–424. Associating for Computing Machinery, New York (2020)
16. Fichte, J.K., Hecher, M., Roland, V.: Parallel model counting with CUDA: algorithm engineering for efficient hardware utilization. In: CP 2021, pp. 24:1–24:20 (2021)
17. Fichte, J.K., Hecher, M., Roland, V.: Proofs for propositional model counting. In: SAT 2022 (2022, to appear)
18. Fichte, J.K., Hecher, M., Thier, P., Woltran, S.: Exploiting database management systems and treewidth for counting. TPLP **22**(1), 128–157 (2022)

19. Fichte, J.K., Gaggl, S.A., Rusovac, D.: Rushing and strolling among answer sets - navigation made easy. In: Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI 2022). The AAAI Press (2022, to appear)
20. Fichte, J.K., Hecher, M., McCreesh, C., Shahab, A.: Complications for computational experiments from modern processors. In: CP 2021, pp. 25:1–25:21 (2021)
21. Hecher, M., Thier, P., Woltran, S.: Taming high treewidth with abstraction, nested dynamic programming, and database technology. In: Pulina, L., Seidl, M. (eds.) SAT 2020. LNCS, vol. 12178, pp. 343–360. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51825-7_25
22. Lagniez, J., Lonca, E., Mailly, J., Rossit, J.: Design and results of ICCMA 2021. CoRR abs/2109.08884 (2021). https://arxiv.org/abs/2109.08884
23. Niskanen, A., Järvisalo, M.: μ-toksia: an efficient abstract argumentation reasoner. In: KR 2020, pp. 800–804 (2020)
24. Rago, A., Cocarascu, O., Toni, F.: Argumentation-based recommendations: fantastic explanations and how to find them. In: IJCAI 2018, pp. 1949–1955 (2018)
25. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. II. Computer Science Press, New York (1989)