ABu: Efficient Argument Builder for Assumption-Based Argumentation

Tuomo Lehtonen

Aalto University, Finland

Abstract

This paper proposes an algorithm for constructing support-minimal and support-unique bipolar AFs from an ABA framework. This means that the constructed arguments are minimal in their support for a particular claim, and arguments with the same support are combined. The BAF representation uses supports to reduce the number of explicit edges in the framework. An empirical evaluation shows that the algorithm outperforms a recent implementation of the same ideas based on answer set enumeration, as well as previous approaches that directly generate an argumentation framework from an ABA framework. Large reductions in the size of the output framework are also shown empirically.

Keywords

assumption-based argumentation, argument instantiation, rule-based argumentation

1. Introduction

The best performing reasoners for structured argumentation formalisms, notably assumption-based argumentation (ABA) and ASPIC$^+$, tend to be based not on constructing an abstract argumentation framework (AF) [1] and employing AF reasoners, but on directly applying SAT solving [2] or ASP [3] at the structured level [4, 5, 6]. However, instantiating an AF can be useful for various reasons, such as interpretability and explainability [7], and the ability to directly use theoretical and computational tools from abstract argumentation. Abstract argumentation is such a fundamental formalism that constructing an AF in the most efficient manner is an important research goal.

This paper focuses on the major structured argumentation formalism ABA, and investigates redundancy-elimination techniques that were recently revisited (and partly introduced) [8]. The techniques in focus here are: 1) disregarding arguments that are not support-minimal, i.e. arguments whose support set is subset-minimal for deriving a particular claim, 2) disregarding arguments that are not support-unique, i.e. only one argument for each distinct support set is created, and 3) only having attacks to singleton assumption arguments, and supports from singleton assumption arguments. Notably, the output remains quite close to the structure of the original framework, since the remaining arguments are roughly speaking a subset of the standard arguments, with some merged, and the edges are easy to read as the original attack relation, retaining the intuitive interpretation of the framework.

These techniques were previously implemented with answer set enumeration [8]. While this is an appropriate tool in the sense that checking whether there is a subset-minimal argument from a set of assumptions to a claim is NP-hard [9], a specialized algorithm might be expected to have better performance, as it would be able to better leverage the recursive structure of arguments. The challenge is in identifying redundant arguments at runtime in order to limit the combinatorial explosion in the number arguments createad in the first place as much as possible (the number of arguments that a given ABA framework gives rise to is NP-complete [10]). While 1) and 2) have been theoretically known for years (by 2017 at the latest [11]), they had not been comprehensively implemented before. A bipolar AF representation had seemingly not been previously considered at all from the point of view of efficient computation.

SAFA'26: Sixth International Workshop on Systems and Algorithms for Formal Argumentation, September, 2026, Barcelona, Spain
tuomo.lehtonen@aalto.fi (T. Lehtonen)
https://tuomolehtonen.com (T. Lehtonen)
0000-0001-6117-4854 (T. Lehtonen)

An algorithm is proposed in this paper for constructing such non-redundant bipolar AFs efficiently. Crucially, redundant arguments are identified early in the construction process, and either never constructed at all (if a less redundant argument for the same claim already exists) or removed as soon as possible (when a less redundant argument that can replace it is constructed). An empirical evaluation confirms that the new imperative algorithm outperforms the ASP enumeration implementation [8], coming closer in performance to state-of-the-art reasoners on acceptance problems.

2. Preliminaries

In assumption-based argumentation (ABA) [12], a deductive system $(\mathcal{L}, \mathcal{R})$ consists of $\mathcal{L}$, a set of sentences and $\mathcal{R}$, a set of inference rules over $\mathcal{L}$ with the form $a_0 \leftarrow a_1, \ldots, a_n$ with $a_i \in \mathcal{L}$. We denote the head of rule r by $head(r) = \{a_0\}$ and the (possibly empty) body of r with $body(r) = \{a_1, \ldots, a_n\}$. An ABA framework (ABAF) is composed of a deductive system along with a set of *assumptions* and *contraries* to assumptions: an ABAF is a tuple $(\mathcal{L}, \mathcal{R}, \mathcal{A}, {}^-)$, where $(\mathcal{L}, \mathcal{R})$ is a deductive system, $\mathcal{A} \subseteq \mathcal{L}$ a non-empty set of assumptions, and ${}^-$ a function mapping assumptions $\mathcal{A}$ to sentences $\mathcal{L}$.

The focus of this paper is on the commonly-studied logic programming instantiation of ABA [12, 13], i.e., all sets defining ABAFs are finite, the elements in $\mathcal{L}$ are atoms, all rules are finite and explicitly given as a list of body literals, and no assumptions occur as a head of a rule (implying flatness [12]).

A sentence $a \in \mathcal{L}$ is (tree-)*derivable* from $X \subseteq \mathcal{A}$, called the support, and rules $R \subseteq \mathcal{R}$, denoted by $X \models_R s$, if either $X = \{s\}$ and $\mathcal{R} = \emptyset$, or there is a finite tree T with the root labeled with s, and the set of labels for the leaves is X (with the possible addition of $\top$) and for each node that is not a leaf, labelled with $s' \in \mathcal{L}$, there is a rule $r \in R$ with s' as the head and the children of the node labeled with exactly the body elements of r. The symbol "$\top$" represents an empty rule body.

Let $D = (\mathcal{L}, \mathcal{R}, \mathcal{A}, {}^-)$ be an ABA framework, and $A, B \subseteq \mathcal{A}$ be two sets of assumptions. A attacks B in D if $A' \vdash \overline{b}$ for some $A' \subseteq A$, and $b \in B$. In other words, assumptions A attack B if one can derive the contrary of an assumption in B. Semantics define acceptable sets of assumptions based on the concepts of conflict-freeness and defense. An assumption set $A \subseteq \mathcal{A}$ is conflict-free in D if A does not attack itself. Set A defends assumption set $B \subseteq \mathcal{A}$ in D if for all $C \subseteq \mathcal{A}$ that attack B it holds that A attacks C. Let us focus on the complete semantics for brevity; the introduced techniques also work for preferred, stable and grounded semantics [8].

Definition 1. *Further, let $A \subseteq \mathcal{A}$ be a conflict-free set of assumptions in D. The set A is admissible if it defends itself. Then A is* complete *if it is admissible and contains every assumption set defended by A.*

A central reasoning problem in ABA is to find out whether a given atom is acceptable under a semantics. This paper focuses on the NP-complete *credulous acceptance*: is there a complete assumption set that derives the queried sentence.

One can straightforwardly construct an AF with identical semantics from an ABAF [13]: have each tree-derivation be an argument, and let an argument A attack B if A derives is the contrary of some assumption used in B. Given any abstract framework with arguments Arg corresponding to an ABAF, let us use the notation that for each $A \in Arg$, $asms(A)$ gives the support set of A and $claims(A)$ gives the set of sentences that A concludes (in the standard definition, $claims(A)$ is a singleton, but we will combine arguments with the same support set and therefore their claims).

Redundancy elimination in ABA Let us recall redundancy elimination for arguments and attacks for ABA, to the extent studied in [8]. Firstly, only subset-minimal derivations for the same claim are needed, producing *support-minimal* AFs.

Definition 2 (Support minimality). *Given an ABAF $D = (\mathcal{L}, \mathcal{R}, \mathcal{A}, {}^-)$, the support-minimal AF $\mathcal{F}_D^{smin} = (Arg_D^{smin}, Att_D^{smin})$ corresponding to D has $Arg_D^{smin} = \{(S, \{p\}) \in 2^{\mathcal{A}} \times \mathcal{L} \mid S \vdash p$ and $\forall S' \subset S : S' \nvdash p\}$ and $Att_D^{smin} = \{(A, B) \in Arg_D^{smin} \times Arg_D^{smin} \mid \exists b \in asms(B) : \overline{b} \in claims(A)\}$.*

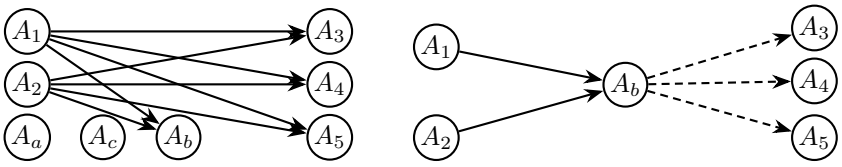


Figure 1: An example (partial) AF and the support-unique BAF from an ABAF.

Secondly, one argument per support set suffices [11], producing *support-unique* frameworks. Support minimality and uniqueness could be separated, but we define the latter based on the former for brevity.

Definition 3 (Support uniqueness). *Given an ABAF $D = (\mathcal{L}, \mathcal{R}, \mathcal{A}, ^-)$, let the support-unique AF $\mathcal{F}_D^{uniq} = (Arg_D^{uniq}, Att_D^{uniq})$ corresponding to D be s.t. $Arg_D^{uniq} = \{(S, C) \in 2^{\mathcal{A}} \times 2^{\mathcal{L}} \mid \exists x \in \mathcal{L} : (S, x) \in Arg_D^{smin}$ and $C = \{p \in \mathcal{L} \mid (S, p) \in Arg_D^{smin}\}\}$, and $Att_D^{uniq} = \{(A, B) \in Arg_D^{uniq} \times Arg_D^{uniq} \mid \exists b \in asms(B) : \overline{b} \in claims(A)\}$.*

Support-minimal and support-unique AFs for an ABAF have the same conclusions (in terms of extensions and claim acceptance) as the standard AF [11]. Note that neither notion is captured by the standard ABA derivations (tree or forward derivations [12, 14]). For example, in an ABAF with rules $(a, b \to x)$, $(a \to x)$, and $(a \to y)$, there is a tree argument for x from $\{a, b\}$, and both x and y are forward-derivable[1] from $\{a, b\}$. In contrast, the argument for x from $\{a, b\}$ is not support-minimal; the only support-unique argument derives $\{x, y\}$ from $\{a\}$.

Beyond minimality of arguments, explicit attack pairs can be replaced with fewer attack and support pairs [8]. We recall the *necessary support* interpretation of bipolar AFs for this purpose (technically, a simpler version without transitivity, since in our construction the support relation is always transitive). Let a BAF be $\mathcal{B} = (Arg, Att, Sup)$. Given a set of arguments $X \subseteq Arg$ and an argument $B \in Arg$, we say that X attacks B if either i) $(A, B) \in Att$ for some $A \in X$, or ii) there is a C such that $(C, B) \in Sup$, and $(A, C) \in Att$. Defence and the semantics are defined as usual: X defends B if for each $C \in Arg$ that attack B, there is an $A \in X$ that attacks C, and the new attack relation is used to define the semantics. Standard ABA semantics are captured by the following BAF formulation [8].

Definition 4. *Consider an ABAF D and the AF $\mathcal{F}$ for it . Given an assumption $a \in \mathcal{A}$, let $b_{singleton}$ be the unique argument B with support $\{b\}$ and $b \in claims(B)$ and Arg_S the set of all such arguments. The BAF $\mathcal{B} = (Arg, Att, Sup)$ for D is such that Arg equals the arguments of $\mathcal{F}$, and $Att = \{(A, b_{singleton}) \in Arg \times Arg_S \mid \overline{b} \in claims(A)\}$ and $Sup = \{(b_{singleton}, A) \in Arg_S \times Arg \mid b \in asms(A)\}$.*

A SAT encoding similar to the standard SAT encodings for AF reasoning can be used [8], in particular to answer acceptance queries or finding acceptable assumption sets under a given semantics.

Consider Figure 1 as an example. Let A_1 and A_2 claim $\overline{b}$, using the assumptions a and c respectively, and arguments A_3, A_4, A_5 to use b in their support (along with some other assumptions). In the standard AF (left side), each attack is duplicated from the arguments deriving $\overline{b}$ to each argument using b. On the right side, we have the support-unique BAF. Here A_a/A_1 and A_c/A_2 are merged to be support-unique, and they only attack A_b, while A_b supports each argument using b.

3. Argument Builder Algorithm

Algorithms 1 and 2 detail pseudocode for procedurally generating a support-unique BAF from a given ABAF. The main components of the algorithm are as follows. The main part of the procedure, i.e. Algorithm 1 is a relatively straightforward bottom-up argument construction scheme. It starts from arguments constructed with rules that have only assumptions in their body. When a new argument is

[1]Forward-derivability corresponds intuitively to a derivation tree existing for the claim from a subset of the support.

constructed, the argument and its claim, h, are stored into *newArgs*. The algorithm attempts to construct further arguments by looping through each rule in whose body h occurs. For each such rule, a new argument is (potentially) created for each different way to derive the other body elements, using existing arguments. If CreateOrUpdate always simply created a new argument for $head(r)$ using the support S, the algorithm would produce the standard AF corresponding to the input ABAF.

Support minimality and support uniqueness CreateOrUpdate (Algorithm 2) employs three redundancy-elimination steps in order to produce support-unique AFs. It has as parameters the claim h and support set S of the potential argument (and the set of all generated arguments *Args*). Lines 6–8 are the basic case where a new argument is simply created with h as the claim and S as the support. The first and last blocks enforce support minimality. Line 1 simply terminates the subprocedure if an argument for h already exists such that it has a subset of S as its support set. Lines 9–12 handle cases where the *new* argument is minimal compared to an existing one: firstly h is deleted from the claims of that non-minimal argument, and secondly the argument is deleted altogether if it has no further claims. Finally, Lines 2–4 enforce support uniqueness by not creating a new argument if one with the same support already exists, and instead adding h to the claims of the existing argument.

Despite checking whether there is a subset-minimal argument from a set of assumptions to a claim being NP-hard [9], enforcing minimality does not add a large overhead here, since it can be done with local checks when (potentially) creating a new argument. Removing redundant argument is thus likely not much less efficient than not doing these checks even if no redundancies exist, while it can be much more efficient when many redundancies exist. A key to notice is that redundancies compound: if one redundant argument is allowed to exists for a claim p, this can double the number of new arguments when p is used to create further arguments. Eliminating redundant arguments as soon as possible is therefore very important; post-processing would be less efficient.

Algorithm 1: Generating a support-unique AF from an ABAF

1 *Args* $\leftarrow \varnothing$; Initialize *newArgs* with rules with only assumptions in body
2 **while** *newArgs* $\neq \varnothing$ **do**
3 Pop (A, h) from *newArgs* `// h is an atom newly derived by A`
4 **foreach** *rule* r *with* $h \in$ *body*(r) **do**
5 **if** $head(r) \in claims(A)$ **then**
6 **continue** `// circularity guard`
7 *rest* $\leftarrow$ *body*$(r) \setminus \{h\}$
8 **foreach** $b \in$ *rest* **do**
9 *args_for_body*$[b] \leftarrow \{A' \mid b \in claims(A')\}$
10 **foreach** *combination* $(C_1, \ldots, C_n) \in \prod_{b \in rest}$ *args_for_body*$[b]$ **do**
11 $S \leftarrow asms(A) \cup \bigcup_i asms(C_i)$
12 CreateOrUpdate(*Args*, $head(r)$, S)
13 **return** *Args*

Sketch for correctness Consider the correctness (all non-redundant arguments are constructed), and optimality (only support-minimal and unique arguments are constructed) of the algorithm. For correctness, consider two aspects. First, all tree arguments are considered by the main program, i.e. CreateOrUpdate is called for each such argument. This is because whenever a new argument is created for a claim h, this is added to *newArgs* and then all rules using h in the body and all known ways to derive the other body elements are considered; any potential derivation not considered here simply would not be supported at this point. Second, it is clear by inspecting the checks in CreateOrUpdate that only arguments that are certainly redundant are deleted or not created in the first place.

As for optimality, suppose for contradiction that a redundant argument was created and was not deleted before the algorithm terminated. This means that there exists two arguments, A and A', such

that either they share a claim while the support set of one is a subset of the other, or their support sets are equal. Assume that A was constructed first and A' second, and that $h \in claims(A) \cap claims(A')$ and $asms(A) \supseteq asms(A')$ (the other cases are analogous or simpler). It then holds that $A \in Args$ at the point when CreateOrUpdate(*Args*, h, $asms(A')$) is called, and therefore h is removed from $claims(A)$ on lines 9–10 and A is removed from *Args* unless it also concludes something else. Note that A might have already been used to generate any number of further arguments, say B, which are redundant compared to the corresponding argument that one gets by replacing A with A', say B'. These redundancies are noticed when that actual argument B' is created, resulting in a non-redundant framework in the end[2].

Algorithm 2: CreateOrUpdate(*Args*, h, S)

```
1  if ∃A' : h ∈ claims(A') and asms(A') ⊆ S then return          // skip non-minimal
2  if ∃A ∈ Args : asms(A) = S then
3  |  claims(A) ← claims(A) ∪ {h}                          // Update existing argument
4  |  newArgs ← newArgs ∪ {(A, h)}
5  else
6  |  Let asms(A) = S, claims(A) = {h}                        // Create new argument
7  |  Args ← Args ∪ {A}
8  |  newArgs ← newArgs ∪ {(A, h)}
9  foreach A' ∈ Args with h ∈ claims(A') and asms(A') ⊇ S do
10 |  claims(A') ← claims(A') \ {h}          // Remove non-minimal derivation for h
11 |  if claims(A') = ∅ then
12 |  |  Args ← Args \ {A'}
```

Constructing attacks and supports Generating the attacks and supports for the BAF formulation is actually faster than generating the attacks for the AF formulation. For BAF, the pairs are local in the sense that for argument A, the number of edges, i.e. outgoing attacks and incoming supports, only depend on the size of $claims(A)$ and $asms(A)$, not on how many other arguments these occur in.

4. ABA Reasoners

An implementation of the algorithm, called ABU$_I$, was written in C++, using CaDiCal [15] as the SAT solver to reason on the resulting (B)AFs. It is available at https://bitbucket.org/lehtonen/argument-builder. We test our implementation against the reasoners detailed next, with the exception of 100BA and SCALLOP, which are similar to ASPFORABA, but were outperformed by it in the most recent ICCMA competition (see https://argumentationcompetition.org/2025/rankings.html).

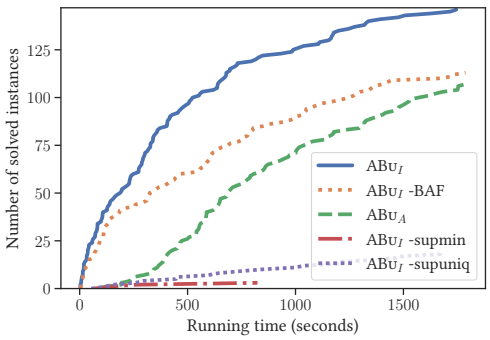
ABU$_A$ produces support-unique (bipolar) AFs, using ASP enumeration [8]. While ASP solvers are quite optimized, parts of arguments are essentially considered multiple times, since the ASP enumeration approach enumerates minimal derivations for each sentence separately. Concretely, if there are arguments $a \to x$ and $a \to x \to y$, ABU$_A$ finds support-minimal arguments for x and y separately, and the existence of the former does not help when constructing the latter argument.

ABA2AF, written in Java, produces, generating support-unique but not fully support-minimal AFs (and solve the AF via ASP) [11]. Differently to ABU$_I$, ABA2AF works top down, starting from claims and generates derivations, logging all subarguments to avoid reconstructing them.

ASPFORABA is the state-of-the-art approach for central reasoning problems [5, 16, 4]. Rather than generate the worst-case exponential AF, ASPFORABA directly encodes problems, such as credulous or skeptical acceptance, into ASP. Not producing a graphical framework potentially limits interpretability.

100BA [6] and SCALLOP implement a similar idea as ASPFORABA, but use a SAT solver rather than an ASP solver, leaving more solver choices as well as for different methods of enforcing acyclicity, which

[2]Redundant arguments being initially created raises the question of which search order heuristics minimizes their number.



(a)

Solver	#solved	avg time (s)
ASPFORABA	276	217.7
ACBAR	185	487.6
ABU$_I$	147	447.5
ABU$_I$ -BAF	114	541.3
ABU$_A$	109	847.8
ABU$_I$ -supuniq	20	834.7
ABU$_I$ -supmin	4	323.2
ABA2AF; MS-DIS; TWEETY	0	—

(b)

Figure 2: (a) Cumulative number of solved instances by versions of ABU. (b) Number of solved instances and mean runtime (over solved instances) for each solver.

is essential to avoid invalid circular reasoning in ABA. Their efficiency is close to ASPFORABA [6, 4].

ACBAR employs a polynomial-time translation from an ABAF to a *different ABAF*, the AF of which is polynomial-sized (and then solves with a SAT solver). This avoids the exponentiality of argument construction, but loses the structure of the initial framework, potentially hindering legibility. New assumptions are created for every rule, and moreover cyclicity in rules is handled via duplicating rules.

MS-DIS implements a version of dispute derivations with multi-shot ASP solving (and outperforms previous dispute derivation implementations) [17]. Disputes are a sequence of moves concerning the acceptability of a claim, ending in a state where either the proponent or opponent has no more legal moves. In the first case, the claim is rejected and in the latter the claim is accepted.

TWEETY [18] does a simple ABAF to AF translation and uses a SAT solver on the resulting AF.

5. Empirical Evaluation

A per-instance time limit of 1800 seconds and memory limit of 16 GB was used. A recent benchmark set was used, from the generator ABCGEN, providing challenging instances that have multiple stable extensions and some circularity in the rules [6].

Figure 2 shows the performance of different variants of ABU$_I$ and existing approaches on answering credulous acceptance under complete semantics in flat ABA. The specifiers -BAF, -supmin and -supuniq mean, respectively, that an AF is used instead of a BAF, and that support minimality and support uniqueness checks, respectively, are skipped. We can see that ABU$_I$ outperforms ABU$_A$, the ASP enumeration approach of constructing support-unique BAFs, with over 30 instances more solved and a lower mean runtime. In fact, from the cumulative plot it is clear that ABU$_I$ can be quite a bit faster on easier instances. For example, under a time limit of 500 seconds, ABU$_A$ solves approximately 25 instances, while ABU$_I$ is able to solve almost 100 instances. Thus ABU$_I$ might be considered more practical even to a larger degree than the number of solved instances suggests, since in the real world there might be diminishing returns to solving instances in tens of minutes or more.

As for the ABU$_I$ variants, performance deteriorates significantly when either non-support-minimal arguments (-supmin) or non-support-unique (but still minimal) arguments (-supuniq) are allowed. The BAF representation has a smaller but noticeable impact. We can further see in the table (Figure 2 (b)) that ABU$_I$ does not quite match ACBAR (not to mention ASPFORABA), but narrows the gap. ABU$_I$ outperforms the other baselines very comfortably: MS-DIS, TWEETY and ABA2AF were unable to solve any instances; ABA2AF was able to generate the AF for 11 instances with a mean runtime of 428 seconds.

The size of the generated frameworks is relevant via its impact on running time efficiency and also as a determinant of the ease of storing and interpreting a framework. Figure 3 shows on the x-axis the number of support-unique arguments (resp attacks+supports) generated by ABU$_I$ and on the y-axis the ratio of arguments (resp attacks+supports) generated by the other solver compared to ABU$_I$. Instances

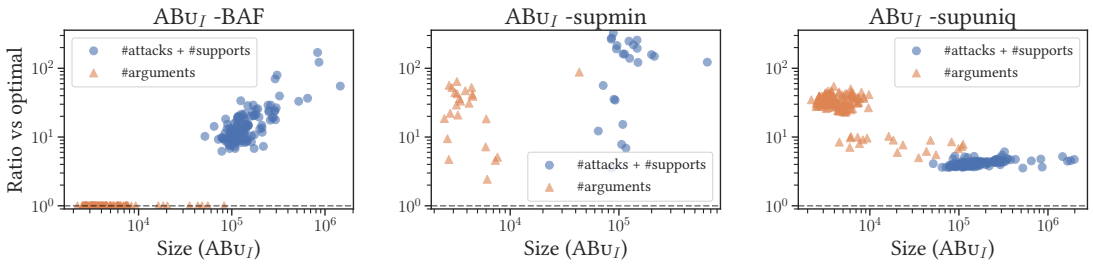


Figure 3: Number of arguments and attacks+supports constructed by ABU variants compared to optimal (ABU$_I$).

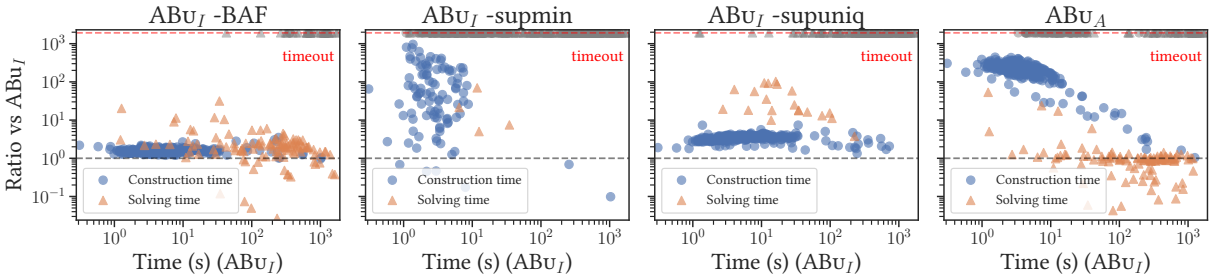


Figure 4: Comparison between ABU$_I$ and variants of ABU on constructing and solving the (B)AFs.

where the B(AF) was generated but the final task was not solved within the time limit are included. From the left side of the figure we can see that for most instances, there is around an order of magnitude more attacks with the AF formulation than there are attacks and supports for the BAF formulation, with a maximum of over two orders of magnitude. In the center and right-side plots we see that without support minimality and support uniqueness, the difference is even larger, with usually quite a bit more than 10 times more arguments and attacks for ABU$_I$-supmin and ABU$_I$-supuniq than ABU$_I$.

Finally, in Figure 4 the x-axis distinguishes the construction (blue dot) and (B)AF solving (orange triangle) time for ABU$_I$, and the y-axis shows how many times slower a given variant of ABU is. Instances where the comparison, but not ABU$_I$, timed out are on the red timeout line. First, BAF solving dominates the runtime for ABU$_I$ itself. Using AF instead of BAF yields relatively modest, but consistent increases in both construction and solving time, and foregoing support minimality or uniqueness yields more significant increases. Support minimality seems more important for construction than uniqueness: there are lots of construction timeouts for ABU$_I$-supmin where ABU$_I$ succeeds, while there are none for ABU$_I$-supuniq. Solving these larger frameworks is still hard: ABU$_I$-supuniq ultimately solves a fraction of the instances of ABU$_I$. Finally, we can observe that ABU$_I$ outperforms ABU$_A$ in the construction time in particular. There is a wider discrepancy between ABU$_I$ and ABU$_A$ in "easy" instances, as also seen in Figure 2.

6. Discussion

This imperative implementation of support-unique BAF construction presents a clear improvement over previous work, in particular the ASP enumeration implementation, which already massively outperformed all other reasoners based on a "direct" ABAF to AF translation. One caveat for the practical relevance of the evaluation is that the efficiency gains might be less pronounced on instances with less of a combinatorial explosion in either the arguments (i.e. there are fewer different ways to derive the same claims), or attacks (i.e. the attack relation is less dense). The benchmark set has both significant amounts of both redundancy types, since it was designed to craft difficult instances. Nonetheless, there is no reason to think ABU$_I$ would ever fare much worse than other (B)AF construction approaches, since the employed techniques incur little runtime overhead and no size overhead. A promising direction for further research is to implement similar ideas for other AF-based structured formalisms, such as ASPIC$^+$, based on the AF-level investigation of support minimality and uniqueness [8].

Acknowledgments

This work has been financially supported by Helsinki Institute for Information Technology HIIT and supported with computational resources by the Aalto Science-IT project.

Declaration on Generative AI

During the preparation of this work, the author used Big Pickle via OpenCode Zen to draft content, namely the LaTeX sources for Figure 1 and Algorithms 1 and 2. Further, the author used Big Pickle to assist in writing the scripts for generating Figures 2–4 as well as to assist in implementing the algorithm, mainly by helping in translating an initial Python implementation to the final C++ version. After using this tool, the author reviewed and edited the content and tested the code as needed and takes full responsibility for the publication's content.

References

[1] P. M. Dung, On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games, Artif. Intell. 77 (1995) 321–358.
[2] A. Biere, Bounded model checking, in: Handbook of Satisfiability, volume 185 of *FAIA*, IOS Press, 2009, pp. 457–481.
[3] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Communications of the ACM 54 (2011) 92–103.
[4] M. Järvisalo, T. Lehtonen, A. Niskanen, ICCMA 2023: 5th international competition on computational models of argumentation, Artif. Intell. 342 (2025) 104311.
[5] T. Lehtonen, J. P. Wallner, M. Järvisalo, Declarative algorithms and complexity results for assumption-based argumentation, J. Artif. Intell. Res. 71 (2021) 265–318.
[6] A. Niskanen, M. F. Rankooh, T. Lehtonen, M. Järvisalo, Reasoning in assumption-based argumentation via SAT, in: M. Ortiz, R. Wassermann, T. Schaub (Eds.), Proc. KR, 2025, p. 707–717.
[7] K. Cyras, A. Rago, E. Albini, P. Baroni, F. Toni, Argumentative XAI: A survey, in: Z. Zhou (Ed.), Proc. IJCAI, ijcai.org, 2021, pp. 4392–4399.
[8] T. Lehtonen, Constructing compact structured argumentation frameworks, in: Proc. COMMA, FAIA, IOS Press, 2026. Accepted for publication.
[9] T. Lehtonen, A. Rapberger, M. Ulbricht, J. P. Wallner, Argumentation frameworks induced by assumption-based argumentation: Relating size and complexity, in: Proc. KR, 2023, pp. 440–450.
[10] H. Strass, A. Wyner, M. Diller, *EMIL*: Extracting meaning from inconsistent language: Towards argumentation using a controlled natural language interface, Int. J. Approx. Reason. 112 (2019) 55–84.
[11] T. Lehtonen, J. P. Wallner, M. Järvisalo, From structured to abstract argumentation: Assumption-based acceptance via AF reasoning, in: Proc. ECSQARU, volume 10369 of *LNCS*, Springer, 2017, pp. 57–68.
[12] A. Bondarenko, P. M. Dung, R. A. Kowalski, F. Toni, An abstract, argumentation-theoretic approach to default reasoning, Artif. Intell. 93 (1997) 63–101.
[13] K. Čyras, X. Fan, C. Schulz, F. Toni, Assumption-based argumentation: Disputes, explanations, preferences, in: P. Baroni, D. Gabbay, M. Giacomin, L. van der Torre (Eds.), Handbook of Formal Argumentation, College Publications, 2018, pp. 365–408.
[14] P. M. Dung, P. Mancarella, F. Toni, Computing ideal sceptical argumentation, Artif. Intell. 171 (2007) 642–674.
[15] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, F. Pollitt, CaDiCaL 2.0, in: CAV, volume 14681 of *LNCS*, Springer, 2024, pp. 133–152. doi:10.1007/978-3-031-65627-9_7.
[16] T. Lehtonen, J. P. Wallner, M. Järvisalo, Harnessing incremental answer set solving for reasoning in assumption-based argumentation, Theory Pract. Log. Program. 21 (2021) 717–734.

[17] M. Diller, P. Gorczyca, ABA disputes in ASP: advancing argument games through multi-shot solving, in: Proc. PRIMA, Lecture Notes in Computer Science, Springer, 2025, pp. 566–584.
[18] M. Thimm, Tweety: A comprehensive collection of Java libraries for logical aspects of artificial intelligence and knowledge representation, in: C. Baral, G. D. Giacomo, T. Eiter (Eds.), Proc. KR, AAAI Press, 2014, pp. 528–537.