A SAT-based Approach for Computing Extensions in Abstract Argumentation

Federico Cerutti[1], Paul E. Dunne[2], Massimiliano Giacomin[3], and Mauro Vallati[4]

[1] School of Natural and Computing Science, King's College, University of Aberdeen, AB24 3UE, Aberdeen, United Kingdom
f.cerutti@abdn.ac.uk

[2] Department of Computer Science, Ashton Building, University of Liverpool, Liverpool L69 7ZF, United Kingdom
ped@csc.liv.ac.uk

[3] Department of Information engineering, University of Brescia, via Branze, 38, 25123, Brescia, Italy
massimiliano.giacomin@ing.unibs.it

[4] School of Computing and Engineering, University of Huddersfield, Huddersfield, HD1 3DH, United Kingdom
m.vallati@hud.ac.uk

Abstract. This paper presents a novel SAT-based approach for the computation of extensions in abstract argumentation, with focus on preferred semantics, and an empirical evaluation of its performances. The approach is based on the idea of reducing the problem of computing complete extensions to a SAT problem and then using a depth-first search method to derive preferred extensions. The proposed approach has been tested using two distinct SAT solvers and compared with three state-of-the-art systems for preferred extension computation. It turns out that the proposed approach delivers significantly better performances in the large majority of the considered cases.

1 Introduction

Dung's theory of abstract argumentation frameworks [17] provides a general model, which is widely recognized as a fundamental reference in computational argumentation in virtue of its simplicity, generality, and ability to capture a variety of more specific approaches as special cases. An abstract argumentation framework (AF) consists of a set of arguments and of an *attack* relation between them. The concept of *extension* plays a key role in this simple setting, where an *extension* is intuitively a set of arguments which can "survive the conflict together". Different notions of extensions and of the requirements they should satisfy correspond to alternative *argumentation semantics*, whose definitions and properties are an active investigation subject since two decades (see [6, 5] for an introduction).

The main computational problems in abstract argumentation are naturally related to extensions and can be partitioned into two classes: *decision* problems and *construction*

problems. Decision problems pose yes/no questions like "Does this argument belong to one (all) extensions?" or "Is this set an extension?", while construction problems require to explicitly produce some of the extensions prescribed by a semantics. In particular, *extension enumeration* is the problem of constructing all the extensions prescribed by a given semantics for a given AF. The complexity of extension-related decision problems has been deeply investigated and, for most of the semantics proposed in the literature they have been proven to be intractable. Intractability extends directly to construction/enumeration problems, given that their solutions provide direct answers to decision problems.

Theoretical analysis of worst-case computational issues in abstract argumentation is in a state of maturity with the available complexity results covering all Dung's traditional semantics and several subsequent prominent approaches in the literature (for a summary see [19]). On the practical side, however, the investigation on efficient algorithms for abstract argumentation and on their empirical assessment is less developed, with few results available in the literature. This paper contributes to fill this gap by proposing a novel approach and implementation for enumeration of Dung's *preferred extensions*, corresponding to one of the most significant argumentation semantics, and comparing its performances with other state-of-the-art implemented systems. We focus on extension enumeration since it can be considered the most general problem, i.e. its solution provides complete information concerning the justification status of arguments (making it possible to determine, for instance, if two arguments cannot be accepted in the same extension) and the proposed approach can be easily adapted to solve also the decision problems mentioned above.

The paper is organized as follows. Section 2 recalls the necessary basic concepts and state-of-the-art background. Section 3 introduces the proposed approach while Section 4 describes the test setting and comments the experimental results. Section 5 provides a comparison with related works and then Section 6 concludes the paper.

2 Background

An argumentation framework [17] consists of a set of arguments[5] and a binary attack relation between them.

Definition 1. *An* argumentation framework *(AF) is a pair $\Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$ where $\mathcal{A}$ is a set of arguments and $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$. We say that $\boldsymbol{b}$* attacks *$\boldsymbol{a}$ iff $\langle \boldsymbol{b}, \boldsymbol{a} \rangle \in \mathcal{R}$, also denoted as $\boldsymbol{b} \rightarrow \boldsymbol{a}$. The set of attackers of an argument $\boldsymbol{a}$ will be denoted as $\boldsymbol{a}^- \triangleq \{\boldsymbol{b} : \boldsymbol{b} \rightarrow \boldsymbol{a}\}$.*

The basic properties of conflict–freeness, acceptability, and admissibility of a set of arguments are fundamental for the definition of argumentation semantics.

Definition 2. *Given an AF $\Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$:*

- *a set $S \subseteq \mathcal{A}$ is* conflict–free *if $\nexists\, \boldsymbol{a}, \boldsymbol{b} \in S$ s.t. $\boldsymbol{a} \rightarrow \boldsymbol{b}$;*
- *an argument $\boldsymbol{a} \in \mathcal{A}$ is* acceptable *with respect to a set $S \subseteq \mathcal{A}$ if $\forall \boldsymbol{b} \in \mathcal{A}$ s.t. $\boldsymbol{b} \rightarrow \boldsymbol{a}$, $\exists\, \boldsymbol{c} \in S$ s.t. $\boldsymbol{c} \rightarrow \boldsymbol{b}$;*

[5] In this paper we consider only *finite* sets of arguments.

– *a set $S \subseteq \mathcal{A}$ is* admissible *if S is conflict–free and every element of S is acceptable with respect to S.*

An argumentation semantics σ prescribes for any $AF\ \Gamma$ a set of *extensions*, denoted as $\mathcal{E}_\sigma(\Gamma)$, namely a set of sets of arguments satisfying some conditions dictated by σ. In [17] four "traditional" semantics were introduced, namely *complete*, *grounded*, *stable*, and *preferred* semantics. Other literature proposals include *semi-stable* [12], *ideal* [18], and *CF2* [7] semantics. Here we need to recall the definitions of complete (denoted as $\mathcal{CO}$) and preferred (denoted as $\mathcal{PR}$) semantics only, along with a well known relationship between them.

Definition 3. *Given an $AF\ \Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$:*

– *a set $S \subseteq \mathcal{A}$ is a* complete extension*, i.e. $S \in \mathcal{E}_{\mathcal{CO}}(\Gamma)$, iff S is admissible and $\forall \boldsymbol{a} \in \mathcal{A}$ s.t. $\boldsymbol{a}$ is acceptable w.r.t. S, $\boldsymbol{a} \in S$;*
– *a set $S \subseteq \mathcal{A}$ is a* preferred extension*, i.e. $S \in \mathcal{E}_{\mathcal{PR}}(\Gamma)$, iff S is a maximal (w.r.t. set inclusion) admissible set.*

Proposition 1. *For any $AF\ \Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$, S is a preferred extension iff it is a maximal (w.r.t. set inclusion) complete extension. As a consequence $\mathcal{E}_{\mathcal{PR}}(\Gamma) \subseteq \mathcal{E}_{\mathcal{CO}}(\Gamma)$.*

It can be noted that each extension S implicitly defines a three-valued *labelling* of arguments, as follows: an argument **a** is labelled `in` iff $\mathbf{a} \in S$, is labelled `out` iff $\exists\ \mathbf{b} \in S$ s.t. $\mathbf{b} \rightarrow \mathbf{a}$, is labelled `undec` if neither of the above conditions holds. In the light of this correspondence, argumentation semantics can be equivalently be defined in terms of labellings rather than of extensions (see [11, 5]). In particular, the notion of *complete labelling* [13, 5] provides an equivalent characterization of complete semantics, in the sense that each complete labelling corresponds to a complete extension and vice versa. Complete labellings can be (redundantly) defined as follows.

Definition 4. *Let $\langle \mathcal{A}, \mathcal{R} \rangle$ be an argumentation framework. A total function $\mathcal{L}ab : \mathcal{A} \mapsto \{\texttt{in}, \texttt{out}, \texttt{undec}\}$ is a* complete labelling *iff it satisfies the following conditions for any $\boldsymbol{a} \in \mathcal{A}$:*

– $\mathcal{L}ab(\boldsymbol{a}) = \texttt{in} \Leftrightarrow \forall \boldsymbol{b} \in \boldsymbol{a}^- \mathcal{L}ab(\boldsymbol{b}) = \texttt{out}$;
– $\mathcal{L}ab(\boldsymbol{a}) = \texttt{out} \Leftrightarrow \exists \boldsymbol{b} \in \boldsymbol{a}^- : \mathcal{L}ab(\boldsymbol{b}) = \texttt{in}$;
– $\mathcal{L}ab(\boldsymbol{a}) = \texttt{undec} \Leftrightarrow \forall \boldsymbol{b} \in \boldsymbol{a}^- \mathcal{L}ab(\boldsymbol{b}) \neq \texttt{in} \wedge \exists \boldsymbol{c} \in \boldsymbol{a}^- : \mathcal{L}ab(\boldsymbol{c}) = \texttt{undec}$;

Moreover, it is proved in [11] that preferred extensions are in one-to-one correspondence with those complete labellings maximizing the set of arguments labelled `in`.

The introduction of preferred semantics is one of the main contribution of Dung's paper. Its name, in fact, reflects a sort of preference w.r.t. other traditional semantics, as it allows multiple extensions (differently from grounded semantics), the existence of extensions is always guaranteed (differently from stable semantics), and no extension is a proper subset of another extension (differently from complete semantics). Also in view of its relevance, computational complexity of preferred semantics has been analyzed early [15, 14] in the literature, with standard decision problems in argumentation semantics resulting to be intractable in the case of $\mathcal{PR}$.

As to algorithms for computing preferred extensions, two basic approaches have been considered in the literature. On one hand, one may develop a dedicated algorithm to obtain the problem solution, on the other hand, one may translate the problem instance at hand into an equivalent instance of a different class of problems for which solvers are already available. The results produced by the solver have then to be translated back to the original problem.

The three main dedicated algorithms for computing preferred extensions in the literature [16, 24, 25] share the same idea based on labellings: starting from an initial default labelling, a sequence of transitions (namely changes of labels) is applied leading to the labellings corresponding to preferred extensions. The three algorithms differ in the initial labelling, the transitions adopted, and the use of additional intermediate labels besides the three standard ones. The algorithm proposed in [25] has been shown to outperform the previous ones and will be therefore taken as the only term of comparison for this family of approaches.

As to the translation approach, the main proposal we are aware of is the ASPARTIX system [22], which provides an encoding of AFs and the relevant computational problems in terms of Answer Set Programs which can be processed by a solver like DLV [23]. Recently an alternative encoding of ASPARTIX using metaASP has been proposed [20] and showed to outperform the previous version when used in conjunction with gringo/claspD solver. ASPARTIX is a very general system, whose capabilities include the computation of preferred extensions, and both versions will be used as reference for this family of approaches.

3 The PrefSat approach

The approach we propose, called PrefSat, can be described as a depth-first search in the space of complete extensions to identify those that are maximal, namely preferred extensions. Each step of the search process requires the solution of a SAT problem through invocation of a SAT solver. More precisely, the algorithm is based on the idea of encoding the constraints corresponding to complete labellings of an AF as a SAT problem and then iteratively producing and solving modified versions of the initial SAT problem according to the needs of the search process. The first step for a detailed presentation of the algorithm concerns therefore the adopted SAT encoding for complete labellings.

3.1 A SAT encoding for complete labellings

A propositional formula over a set of boolean variables is satisfiable iff there exists a truth assignment of the variables such that the formula evaluates to True. Checking whether such an assignment exists is the satisfiability (SAT) problem. Given an AF $\Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$ we are interested in identifying a boolean formula, called *complete labelling formula* and denoted as Π_Γ, such that each satisfying assignment of the formula corresponds to a complete labelling. While this might seem a clear-cut task, several syntactically different encodings can be devised which, while being logically equivalent, can significantly affect the performance of the overall process of searching a satisfying assignment. For instance, adding some "redundant" clauses to a formula may speed up

the search process, thanks to the additional constraints. On the other hand, increasing syntactic complexity might lead to worse performances, thus a careful selection of the encoding is needed.

In order to explore alternative encodings, let us consider as a starting point a boolean formula encoding the constraints of Definition 4 in conjunctive normal form (CNF), as required by the SAT solver. To this purpose, we have to introduce some notation. Letting $k = |\mathcal{A}|$ we can identify each argument with an index in $\{1, \dots k\}$ or, more precisely, we can define a bijection $\phi : \{1, \dots, k\} \mapsto \mathcal{A}$ (the inverse map will be denoted as ϕ^{-1}). ϕ will be called an indexing of $\mathcal{A}$ and the argument $\phi(i)$ will be sometimes referred to as argument i for brevity. For each argument i we define three boolean variables, I_i, O_i, and U_i, with the intended meaning that I_i is true when argument i is labelled `in`, false otherwise, and analogously O_i and U_i correspond to labels `out` and `undec`. Formally, given $\Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$ we define the corresponding set of variables as $\mathcal{V}(\Gamma) \triangleq \cup_{1 \leq i \leq |\mathcal{A}|} \{I_i, O_i, U_i\}$. Now we express the constraints of Definition 4 in terms of the variables $\mathcal{V}(\Gamma)$, with the additional condition that for each argument i exactly one of the three variables has to be assigned the value True. For technical reasons we restrict to"non-empty" extensions (in the sense that at least one of the arguments is labelled `in`), thus we add the further condition that at least one variable I_i is assigned the value True. The detail of the resulting CNF is given in Definition 5.

Definition 5. *Given an AF $\Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$, with $|\mathcal{A}| = k$ and $\phi : \{1, \dots, k\} \mapsto \mathcal{A}$ an indexing of $\mathcal{A}$, the* C_1 encoding *defined on the variables in $\mathcal{V}(\Gamma)$, is given by the conjunction of the formulae listed below:*

$$\bigwedge_{i \in \{1,\dots,k\}} \Big((I_i \vee O_i \vee U_i) \wedge (\neg I_i \vee \neg O_i) \wedge (\neg I_i \vee \neg U_i) \wedge (\neg O_i \vee \neg U_i) \Big) \tag{1}$$

$$\bigwedge_{\{i | \phi(i)^- = \emptyset\}} (I_i \wedge \neg O_i \wedge \neg U_i) \tag{2}$$

$$\bigwedge_{\{i | \phi(i)^- \neq \emptyset\}} \left(I_i \vee \left(\bigvee_{\{j | \phi(j) \to \phi(i)\}} (\neg O_j) \right) \right) \tag{3}$$

$$\bigwedge_{\{i | \phi(i)^- \neq \emptyset\}} \left(\bigwedge_{\{j | \phi(j) \to \phi(i)\}} \neg I_i \vee O_j \right) \tag{4}$$

$$\bigwedge_{\{i | \phi(i)^- \neq \emptyset\}} \left(\bigwedge_{\{j | \phi(j) \to \phi(i)\}} \neg I_j \vee O_i \right) \tag{5}$$

$$\bigwedge_{\{i | \phi(i)^- \neq \emptyset\}} \left(\neg O_i \vee \left(\bigvee_{\{j | \phi(j) \to \phi(i)\}} I_j \right) \right) \tag{6}$$

$$\bigwedge_{\{i|\phi(i)^- \neq \emptyset\}} \left(\bigwedge_{\{k|\phi(k) \rightarrow \phi(i)\}} \left(U_i \vee \neg U_k \vee \left(\bigvee_{\{j|\phi(j) \rightarrow \phi(i)\}} I_j \right) \right) \right) \quad (7)$$

$$\bigwedge_{\{i|\phi(i)^- \neq \emptyset\}} \left(\left(\bigwedge_{\{j|\phi(j) \rightarrow \phi(i)\}} (\neg U_i \vee \neg I_j) \right) \wedge \left(\neg U_i \vee \left(\bigvee_{\{j|\phi(j) \rightarrow \phi(i)\}} U_j \right) \right) \right) \quad (8)$$

$$\bigvee_{i \in \{1,\ldots k\}} I_i \quad (9)$$

C_1 corresponds to the conditions of Definition 4 with the addition of the non-emptyness requirement. Formula (1) states that for each argument i one and only one label has to be assigned. Formula (2) settles the case of unattacked arguments that must be labelled `in`. Formula (3) states that argument i is labelled `in` if all its attackers are labelled `out`, while Formula (4) settles the reverse (i.e. the 'only if') condition. Formula (5) corresponds to the constraint that argument i is labelled `out` if at least one of its attackers is labelled `in`, while Formula (6) corresponds to the 'only if' condition. Formula (7) is a bit more articulated: it states that argument i is labelled `undec` if none of its attackers is labelled `in` and at least one of its attackers is labelled `undec`, and again Formula (8) corresponds to the 'only if' condition. Finally, formula (9) ensures non-emptyness, i.e. that at least one argument is labelled `in`.

The encoding of Definition 5 is redundant, i.e. it is possible to drop out some clauses so as to obtain a simpler logically equivalent encoding. The following proposition, whose proof, based on the requirement that $\mathcal{L}ab$ is a total function (cf. Def. 4), is omitted due to space limitations, shows the alternative non redundant encodings.

Proposition 2. *Referring to the formulae listed in Definition 5, the following encodings are equivalent:*

$C_1 : (1) \wedge (2) \wedge (3) \wedge (4) \wedge (5) \wedge (6) \wedge (7) \wedge (8) \wedge (9)$
$C_1^a : (1) \wedge (2) \wedge (3) \wedge (4) \wedge (5) \wedge (6) \wedge (9)$
$C_1^b : (1) \wedge (2) \wedge (5) \wedge (6) \wedge (7) \wedge (8) \wedge (9)$
$C_1^c : (1) \wedge (2) \wedge (3) \wedge (4) \wedge (7) \wedge (8) \wedge (9)$
$C_2 : (1) \wedge (2) \wedge (4) \wedge (6) \wedge (8) \wedge (9)$
$C_3 : (1) \wedge (2) \wedge (3) \wedge (5) \wedge (7) \wedge (9)$

C_1^a, C_1^b and C_1^c correspond to Definition 4 where the third, first and second iff condition is removed, respectively, C_2 codifies the 'only if' ($\Rightarrow$) part of the conditions and C_3 the 'if' ($\Leftarrow$) part[6].

In Section 4 we evaluate the performance of the overall approach for enumerating the preferred extensions given the above six encodings. In the next section we describe the core of our proposal.

[6] C_1^a and C_2 correspond to the alternative definitions of complete labellings in [13], where a proof of their equivalence is provided.

Algorithm 1 Enumerating the preferred extensions of an AF

1: **Input:** $\Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$
2: **Output:** $E_p \subseteq 2^{\mathcal{A}}$
3: $E_p := \emptyset$
4: $cnf := \Pi_\Gamma$
5: **repeat**
6: $prefcand := \emptyset$
7: $cnfdf := cnf$
8: **repeat**
9: $lastcompfound := SS(cnfdf)$
10: **if** $lastcompfound \,!= \varepsilon$ **then**
11: $prefcand := lastcompfound$
12: **for a** $\in INARGS(lastcompfound)$ **do**
13: $cnfdf := cnfdf \wedge I_{\phi^{-1}(\mathbf{a})}$
14: **end for**
15: $remaining := FALSE$
16: **for a** $\in \mathcal{A} \setminus INARGS(lastcompfound)$ **do**
17: $remaining := remaining \vee I_{\phi^{-1}(\mathbf{a})}$
18: **end for**
19: $cnfdf := cnfdf \wedge remaining$
20: **end if**
21: **until** $(lastcompfound \,!= \varepsilon \wedge INARGS(lastcompfound) \,!= \mathcal{A})$
22: **if** $prefcand \,!= \emptyset$ **then**
23: $E_p := E_p \cup \{INARGS(prefcand)\}$
24: $oppsolution := FALSE$
25: **for a** $\in \mathcal{A} \setminus INARGS(prefcand)$ **do**
26: $oppsolution := oppsolution \vee I_{\phi^{-1}(\mathbf{a})}$
27: **end for**
28: $cnf := cnf \wedge oppsolution$
29: **end if**
30: **until** $(prefcand \,!= \emptyset)$
31: **if** $E_p = \emptyset$ **then**
32: $E_p = \{\emptyset\}$
33: **end if**
34: **return** E_p

3.2 Enumerating preferred extensions

We are now in a position to illustrate the proposed procedure, called PrefSat and listed in Algorithm 1, to enumerate the preferred extensions of an AF $\Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$.

Algorithm 1 resorts to two external functions: SS, and $INARGS$. SS is a SAT solver able to prove unsatisfiability too: it accepts as input a CNF formula and returns a variable assignment satisfying the formula if it exists, ε otherwise. $INARGS$ accepts as input a variable assignment concerning $\mathcal{V}(\Gamma)$ and returns the corresponding set of arguments labelled as `in`. Moreover we take for granted the computation of Π_Γ from Γ (using one of the equivalent encodings shown in Proposition 2), which is carried out in the initialization phase (line 4).

Theorem 1 proves the correctness of Algorithm 1.

Theorem 1. *Given an AF* $\Gamma = \langle \mathcal{A}, \mathcal{R} \rangle$ *Algorithm 1 returns* $E_p = \mathcal{E}_{\mathcal{PR}}(\Gamma)$.

The proof of the above Theorem is omitted due to space limitations, however we provide an explanation of the algorithm. The algorithm mainly consists of two nested **repeat-until** loops. Roughly, the inner loop (lines 8–21) corresponds to a depth-first search which, starting from a non-empty complete extension, produces a sequence of complete extensions strictly ordered by set inclusion. When the sequence can no more be extended, its last element corresponds to a maximal complete extension, namely to a preferred extension. The outer loop (lines 5–30) is in charge of driving the search: it ensures, through proper settings of the variables, that the inner loop is entered with different initial conditions, so that the whole space of complete extensions is explored and all preferred extensions are found.

Let us now illustrate the operation of Algorithm 1 in detail. Given the correspondence between variable assignments, labellings, and extensions, we will resort to some terminological liberty for the sake of conciseness and clarity (e.g. stating that the solver returns an extension rather than that it returns an assignment which corresponds to a labelling which in turn corresponds to an extension). In the first iteration of the outer loop, the assignment of line 7 results in $cnfdf = \Pi_\Gamma$ in virtue of the initialization of line 4. Then the inner loop is entered and, at line 9, SS is invoked on Π_Γ. Due to the non-emptyness condition in Π_Γ, SS returns ε if the only complete extension (and hence the only preferred extension) of Γ is the empty set. In this case, lines 11–19 are not executed and the loop is directly exited. As a consequence, $prefcand$ is still empty at line 22 and also the outer loop is directly exited. The condition of line 31 then holds, the assignment of line 32 is executed and the algorithm terminates returning $\{\emptyset\}$.

Let us now turn to the more interesting case where there is at least one non-empty complete extension. Then, the first solver invocation returns (non deterministically) one of the non-empty complete extensions of the framework which is assigned to $lastcompfound$ at line 9. Then the condition of line 10 is verified and $lastcompfound$ is set as the candidate preferred extension (line 11). In lines 12–19 the formula $cnfdf$ is updated in order to ensure that the next call to SS returns a complete extension which is a strict superset of $lastcompfound$ (if any exists). This is achieved by imposing that all elements of $lastcompfound$ are labelled `in` (lines 12–14) and that at least one further argument is labelled `in` (lines 15–19). In the next iteration (if any), the modified $cnfdf$ is submitted to SS. If a solution is found, the inner loop is iterated in the same way: at each successful iteration a new, strictly larger, complete extension is found. According to the conditions stated in line 21, iteration of the inner loop will then terminate when the call to SS is not successful or when $lastcompfound$ covers all arguments, since in this case no larger complete extension can be found. If a new preferred extension has been found, it is added to the output set E_p (line 23). Then, a formula is produced which ensures that any further solution includes at least an argument not included in the already found one (lines 25–27). This formula is then added to cnf (line 28). The outer loop then restarts resetting variables at lines 6–7 in preparation for a new execution of the inner loop. The inner loop is entered with $cnfdf$ updated at line 7, this ensures that the call to SS either does not find any solution (and then the algorithm terminates returning E_p as already set) or finds a new complete extension which is not a subset of

any of the preferred extensions already found and is then extended to a new preferred extension in the subsequent iterations of the loop.

4 The Empirical Analysis

The algorithm described in the previous section has been implemented in C++ and integrated with two alternative SAT solvers, namely PrecoSAT and Glucose. PrecoSAT [9] is the winner of the SAT Competition[7] 2009 on the Application track. Glucose [3, 4] is the winner of the SAT Competition in 2011 and of the SAT Challenge 2012 on the Application track.

This choice gave rise to the following two systems:

- PrefSat with PrecoSAT (**PS-PRE**);
- PrefSat with Glucose (**PS-GLU**).

To assess empirically the performance of the proposed approach with respect to other state-of-the-art systems and to compare the two SAT solvers on the SAT instances generated by our approach, we ran a set of tests on randomly generated AFs.

The experimental analysis has been conducted on 2816 AFs that were divided in different classes, according to two dimensions: the number of arguments, $|\mathcal{A}|$ and the criterion of random generation of the attack relation. As to $|\mathcal{A}|$ we considered 8 different values, ranging from 25 to 200 with a step of 25. As to the generation of the attack relation we used two alternative methods. The first method consists in fixing the probability p_{att} that there is an attack for each ordered pair of arguments (self-attacks are included): for each pair a pseudo-random number uniformly distributed between 0 and 1 is generated and if it is lesser or equal to p_{att} the pair is added to the attack relation. We considered three values for p_{att}, namely 0.25, 0.5, and 0.75. Combining the 8 values of $|\mathcal{A}|$ with the 3 values of p_{att} gives rise to 24 test classes, each of which has been populated with 50 AFs.

The second method consists in generating randomly, for each AF, the number n_{att} of attacks it contains (extracted with uniform probability between 0 and $|\mathcal{A}|^2$). Then the n_{att} distinct pairs of arguments constituting the attack relation are selected randomly. Applying the second method with the 8 values of $|\mathcal{A}|$ gives rise to 8 further test classes, each of which has been populated with 200 AFs. Since the experimental results show minimal changes between the sets of AFs generated with the two methods, hereafter we silently drop this detail.

Further, we also considered, for each value of $|\mathcal{A}|$, the extreme cases of empty attack relation ($p_{att} = n_{att} = 0$) and of fully connected attack relation ($p_{att} = 1, n_{att} = |\mathcal{A}|^2$), thus adding 16 "singleton" test classes.

The tests have been run on the same hardware (a Quad-core Intel(R) Xeon(TM) CPU 2.80GHz with 4 GByte RAM and Linux operating system). As in the well-known international planning competition[8] (IPC), a limit of 15 minutes was imposed to compute the preferred extensions for each AF. No limit was imposed on the RAM usage,

[7] http://www.satcompetition.org/

[8] http://www.plg.inf.uc3m.es/ipc2011-learning/Rules

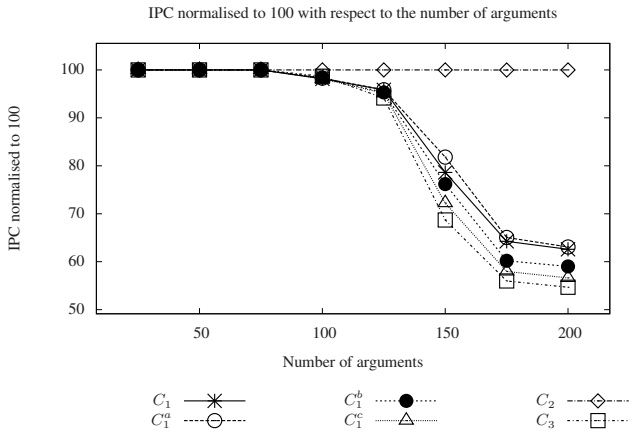

Fig. 1: IPC w.r.t. $|\mathcal{A}|$ (all test cases), comparing **PS-GLU** using respectively encodings C_1, C_1^a, C_1^b, C_1^c, C_2, and C_3 (cf. Proposition 2).

but a run fails at saturation of the available memory, including the swap area. The systems under evaluation have been compared with respect to the ability to produce solutions within the time limit and to the execution time (obtained as the real value of the command `time -p`). As to the latter comparison, we adopted the IPC speed score, also borrowed from the planning community, which is defined as follows:

- For each test case (in our case, each test AF) let T^* be the best execution time among the compared systems (if no system produces the solution within the time limit, the test case is not considered valid and ignored).
- For each valid case, each system gets a score of $1/(1 + \log_{10}(T/T^*))$, where T is its execution time, or a score of 0 if it fails in that case. Runtimes below 1 sec get by default the maximal score of 1.
- The (non normalized) IPC score for a system is the sum of its scores over all the valid test cases. The normalised IPC score ranges from 0 to 100 and is defined as $(IPC/\#\text{ of valid cases}) * 100$.

First of all, we ran an investigation on which of the alternative encodings introduced in Proposition 2 performs best. While there are cases where **PS-PRE** performs better using C_1^a and others where it performs better using C_2 (with minimal differences on average), it is always outperformed by **PS-GLU** using C_2, thus we refer to **PS-GLU** to illustrate the difference of performance induced by the alternative encodings. In Figure 1, we compare the empirical results obtained by executing **PS-GLU**, and Table 1 summarises the average times. It is worth to mention that **PS-GLU** always computed the preferred extensions irrespective of the chosen encoding. As we can see, the overall performance is significantly dependent on the set of conditions used, where the greatest performance (considering the generated AFs) is C_2, and then in sequence, generally C_1^a, C_1, C_1^b, C_1^c and C_3, although we have empirical evidences (omitted due to space

$\|\mathcal{A}\|$	C_1	C_1^a	C_1^b	C_1^c	C_2	C_3
25	5.97E-03	5.91E-03	5.43E-03	5.31E-03	**6.25E-04**	3.92E-03
50	3.50E-02	3.39E-02	3.38E-02	3.38E-02	**9.74E-03**	3.10E-02
75	1.06E-01	1.02E-01	1.05E-01	1.06E-01	**2.74E-02**	1.02E-01
100	2.76E-01	2.65E-01	2.78E-01	2.91E-01	**6.39E-02**	2.89E-01
125	5.24E-01	5.03E-01	5.54E-01	5.95E-01	**1.15E-01**	6.23E-01
150	1.27E+00	1.22E+00	1.39E+00	1.43E+00	**2.46E-01**	1.60E+00
175	2.06E+00	1.98E+00	2.46E+00	2.82E+00	**4.80E-01**	3.51E+00
200	5.00E+00	4.89E+00	6.17E+00	7.90E+00	**1.38E+00**	1.00E+01

Table 1: Average time (in seconds) for computing the preferred extensions according to the different labellings encoding of Proposition 2 grouped by $|\mathcal{A}|$. In bold the best one.

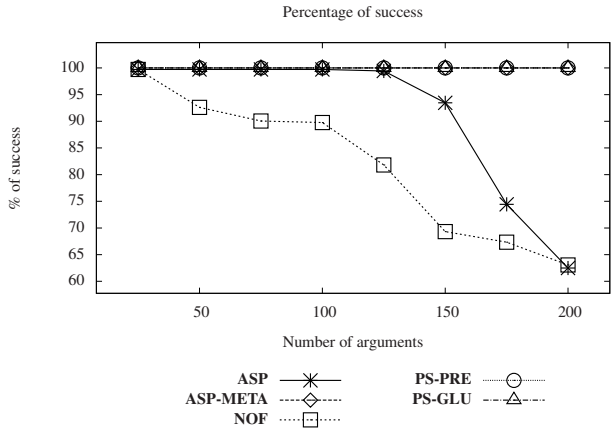

Fig. 2: Percentage of success over all test cases.

constraints) showing that on dense graphs there are situations where C_3 performs better than C_1.

In order to evaluate the overall performance of Algorithm 1 (cf. Section 3), let us compare **PS-PRE** and **PS-GLU** both using encoding C_2 with the other three notable systems at the state of the art:

- ASPARTIX with `dlv` as ASP solver (denoted as **ASP**);
- ASPARTIX-META with `gringo` as grounder and `claspD` as ASP solver (denoted as **ASP-META**) as presented in [20];
- the system presented in [25] (**NOF**)[9].

None of five (considering also our **PS-GLU** and **PS-PRE**) systems uses parallel execution.

[9] We thank Samir Nofal for kindly providing the source code.

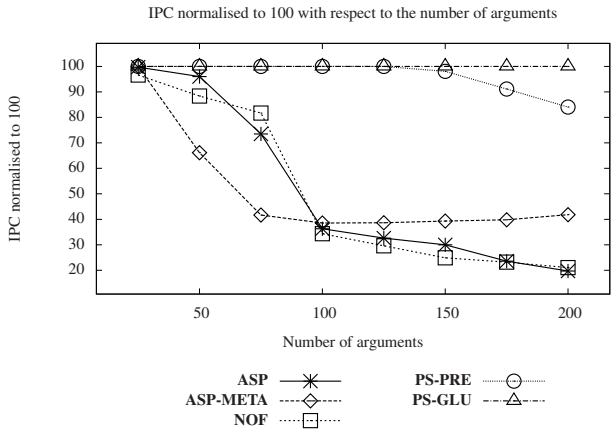

Fig. 3: IPC w.r.t. $|\mathcal{A}|$ (all test cases).

Concerning the ability to produce solutions, Figure 2 summarizes the results concerning all test cases grouped w.r.t. $|\mathcal{A}|$. **PS-GLU**, **PS-PRE** (both exploiting C_2 encoding), and **ASP-META** were able to produce the solution in all cases. On the other hand, the success rate of both **ASP** and **NOF** decreases significantly with the increase of $|\mathcal{A}|$. We observed that the failure reasons are quite different: **ASP** reached in all its failure cases the 15 minutes time limit, while **NOF** ran out of memory before reaching the time limit. In the light of this observation, **NOF**'s evaluation has certainly been negatively affected by the relatively scarce memory availability of the test platform, but, from another perspective, the results obtained on this platform give a clear indication about the different resource needs of the compared systems.

$\|\mathcal{A}\|$	**ASP**	**ASP-META**	**NOF**	**PS-PRE**	**PS-GLU**
25	7.78E-02	2.70E-01	3.24E-01	3.87E-03	**6.27E-04**
50	3.32E-01	1.00E+00	5.43E-01	2.32E-02	**1.04E-02**
75	1.03E+00	2.30E+00	1.18E+00	5.98E-02	**2.96E-02**
100	3.75E+00	4.33E+00	3.81E+00	1.36E-01	**6.84E-02**
125	1.63E+01	6.95E+00	8.50E+00	2.46E-01	**1.24E-01**
150	3.16E+01	1.16E+01	1.47E+01	4.59E-01	**2.24E-01**
175	6.65E+01	1.61E+01	2.64E+01	6.65E-01	**3.21E-01**
200	1.24E+02	2.27E+01	5.02E+01	1.02E+00	**4.79E-01**

Table 2: Average time (in seconds) for computing the preferred extensions needed by the five systems grouped by $|\mathcal{A}|$ on AF s for which all the systems computed correctly the preferred extensions. In bold the best one.

Turning to the comparison of execution times, Figure 3 presents the values of normalized IPC considering all test cases grouped w.r.t. $|\mathcal{A}|$, while Table 2 shows the average time needed by the five systems for computing the preferred extension. Both **PS-PRE** and **PS-GLU** performed significantly better (note that the IPC score is logarithmic) than **ASP** and **NOF** for all values of $|\mathcal{A}| > 25$, and the performance gap increases with increasing $|\mathcal{A}|$. Moreover, **PS-GLU** is significantly faster than **PS-PRE** for $|\mathcal{A}| > 175$ (again the performance gap increases with increasing $|\mathcal{A}|$). **ASP** and **NOF** obtained quite similar IPC values with more evident differences at lower values of $|\mathcal{A}|$. Surprisingly, **ASP-META** performed worse that its older version **ASP** (and also of **NOF**) on frameworks with number of arguments up to 100 (cf. Table 2). Although this may seem in contrast with results provided in [20], it has to be remarked that the IPC measure is logarithmic w.r.t. the best execution time, while [20, Fig. 1] uses a linear scale, and this turned to be a disadvantage when analysing the overall performance. Indeed, the maximum difference of execution times between **ASP** and **ASP-META** executed on frameworks up to 100 arguments is around 1.2 seconds, while the axis of ordinate of [20, Fig. 1] ranges between 0 and 300, thus making impossible to note this difference.

5 Comparison with Related Works

The relationship between argumentation semantics and the satisfiability problem has been already considered in the literature, but less effort has been devoted to the study of a SAT-based algorithm and its empirical evaluation. For instance, in [8] three approaches determining semantics extensions are preliminary described, namely the *equational checking*, the *model checking*, and the *satisfiability cheking* of which three different formulations for, respectively, stable extension, admissible set, and complete extension are presented from a theoretical perspective, without providing any empirical evaluation.

More recently, in [10], and similarly in [1], relationships between argumentation semantics and constraint satisfaction problems are studied, with different formulation for each semantics or decision problem. In particular, [1] proposes an extensive study of CSP formulations for decision problems related to stable, preferred, complete, grounded and admissible semantics, while [10] shows an empirical evaluation of their approach through their software ConArg, but for conflict-free, admissible, complete and stable extensions only.

Probably the most relevant work is [21], where a method for computing credulous and skeptical acceptance for preferred, semi-stable, and stage semantics has been studied, implemented, and empirically evaluated using an algorithm based upon a NP-oracle, namely a SAT solver. Differently from our work, this approach is focused on acceptance problems only and does not address the problem of how to enumerate the extensions. As we do believe that the approach we showed in this paper can be easily adapted for dealing with both credulous acceptance (we have just to force the SAT solver to consider a given argument as labelled `in`) and skeptical acceptance (we have just to check whether a given argument is in all the extensions), we have already started a theoretical and empirical investigation on this subject.

Finally, as the computation of the preferred extension using [13]'s labelling approach requires a maximisation process, at a first sight this seems to be quite close to a MaxSAT problem [2], which is a generalisation of the satisfiability problem. The idea is that sometimes some constraints of a problem can not be satisfied, and a solver should try to satisfy the maximum number of them. Although there are approaches aimed at finding the maximum w.r.t. set inclusion satisfiable constraints (i.e. nOPTSAT[10]), the MaxSAT problem is conceptually different from the problem of finding the preferred extensions. Indeed, for determining the preferred extensions we maximise the acceptability of a subset of variables, while in the MaxSAT problem it is not possible to bound such a maximisation to a subset of variables only. However, a deeper investigation that may lead to the definition of argumentation semantics as MaxSAT problems is already envisaged as a future work.

6 Conclusions

We presented a novel SAT-based approach for preferred extension enumeration in abstract argumentation and assessed its performances by an empirical comparison with other state-of-the-art systems. The proposed approach turns out to be efficient and to generally outperform the best known dedicated algorithm and the ASP-based approach implemented in the ASPARTIX system. The proposed approach appears to be applicable for extension enumeration of other semantics (in particular stable and semi-stable) and this represents an immediate direction of future work. As to performance assessment, we are not aware of other systematic comparisons concerning computation efficiency in Dung's framework apart the results presented in [25], where different test sets were used for each pairwise comparison, with a maximum argument cardinality of 45. The comparison provided in [20] is aimed just at showing the differences between the two different encoding of ASPARTIX. Java-based tools mainly conceived for interactive use, like ConArg [10] or Dungine [26], are not suitable for a systematic efficiency comparison on large test sets and could not be considered in this work. It can be remarked however that they adopt alternative solution strategies (translation to a CSP problem in ConArg, argument games in Dungine) whose performance evaluation is an important subject of future work.

We are currently working to integrate the proposed approach into the SCC-recursive schema introduced in [7] to encompass several semantics (including grounded, preferred and stable semantics). More specifically, the approach proposed in this paper can be applied to the sub-frameworks involved in the base-case of the recursion: since such local application decreases the number of variables involved, we expect a dramatic performance increase.

References

1. Amgoud, L., Devred, C.: Argumentation frameworks as constraint satisfaction problems. Annals of Mathematics and Artificial Intelligence pp. 1–18 (2013)

[10] `www.star.dist.unige.it/~emanuele/nOPTSAT/`

2. Ansótegui, C., Bonet, M.L., Levy, J.: Sat-based maxsat algorithms. Artificial Intelligence 196(0), 77 – 105 (2013)
3. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: Proceedings of IJCAI 2009. pp. 399–404 (2009)
4. Audemard, G., Simon, L.: Glucose 2.1. http://www.lri.fr/~simon/?page=glucose (2012)
5. Baroni, P., Caminada, M., Giacomin, M.: An introduction to argumentation semantics. Knowledge Engineering Review 26(4), 365–410 (2011)
6. Baroni, P., Giacomin, M.: Semantics of abstract argumentation systems. In: Argumentation in Artificial Intelligence, pp. 25–44. Springer (2009)
7. Baroni, P., Giacomin, M., Guida, G.: SCC-recursiveness: a general schema for argumentation semantics. Artificial Intelligence 168(1-2), 165–210 (2005)
8. Besnard, P., Doutre, S.: Checking the acceptability of a set of arguments. In: Proceedings of NMR 2004. pp. 59–64 (2004)
9. Biere, A.: P{re,ic}osat@sc'09. In: SAT Competition 2009 (2009)
10. Bistarelli, S., Santini, F.: Modeling and solving afs with a constraint-based tool: Conarg. In: Theory and Applications of Formal Argumentation, vol. 7132, pp. 99–116. Springer (2012)
11. Caminada, M.: On the issue of reinstatement in argumentation. In: Proceedings of JELIA 2006. pp. 111–123 (2006)
12. Caminada, M.: Semi-stable semantics. In: Proceedings of COMMA 2006. pp. 121–130 (2006)
13. Caminada, M., Gabbay, D.M.: A logical account of formal argumentation. Studia Logica (Special issue: new ideas in argumentation theory) 93(2–3), 109–145 (2009)
14. Dimopoulos, Y., Nebel, B., Toni, F.: Preferred arguments are harder to compute than stable extensions. In: Proceedings of IJCAI 1999. pp. 36–43 (1999)
15. Dimopoulos, Y., Torres, A.: Graph theoretical structures in logic programs and default theories. Journal Theoretical Computer Science 170, 209–244 (1996)
16. Doutre, S., Mengin, J.: Preferred extensions of argumentation frameworks: Query answering and computation. In: Proceedings of IJCAR 2001. pp. 272–288 (2001)
17. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming, and n-person games. Artificial Intelligence 77(2), 321–357 (1995)
18. Dung, P., Mancarella, P., Toni, F.: A dialectic procedure for sceptical, assumption-based argumentation. In: Proceedings of COMMA 2006. pp. 145–156 (2006)
19. Dunne, P.E., Wooldridge, M.: Complexity of abstract argumentation. In: Argumentation in Artificial Intelligence, pp. 85–104. Springer (2009)
20. Dvořák, W., Gaggl, S.A., Wallner, J., Woltran, S.: Making use of advances in answer-set programming for abstract argumentation systems. In: Proceedings of INAP 2011 (2011)
21. Dvořák, W., Järvisalo, M., Wallner, J.P., Woltran, S.: Complexity-sensitive decision procedures for abstract argumentation. In: Proceedings of KR 2012. AAAI Press (2012)
22. Egly, U., Gaggl, S.A., Woltran, S.: Aspartix: Implementing argumentation frameworks using answer-set programming. In: Proceedings of ICLP 2008. pp. 734–738 (2008)
23. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
24. Modgil, S., Caminada, M.: Proof theories and algorithms for abstract argumentation frameworks. In: Argumentation in Artificial Intelligence, pp. 105–129. Springer (2009)
25. Nofal, S., Dunne, P.E., Atkinson, K.: On preferred extension enumeration in abstract argumentation. In: Proceedings of COMMA 2012. pp. 205–216 (2012)
26. South, M., Vreeswijk, G., Fox, J.: Dungine: A java dung reasoner. In: Proceedings of COMMA 2008. pp. 360–368 (2008)