Solving Set Optimization Problems by Cardinality Optimization with an Application to Argumentation

Wolfgang Faber\textsuperscript{1} and Mauro Vallati\textsuperscript{1} and Federico Cerutti\textsuperscript{2} and Massimiliano Giacomin\textsuperscript{3}

Abstract.
Optimization—minimization or maximization—in the lattice of subsets is a frequent operation in Artificial Intelligence tasks. Examples are subset-minimal model-based diagnosis, nonmonotonic reasoning by means of circumscription, or preferred extensions in abstract argumentation. Finding the optimum among many admissible solutions is often harder than finding admissible solutions with respect to both computational complexity and methodology. This paper addresses the former issue by means of an effective method for finding subset-optimal solutions. It is based on the relationship between cardinality-optimal and subset-optimal solutions, and the fact that many logic-based declarative programming systems provide constructs for finding cardinality-optimal solutions, for example maximum satisfiability (MaxSAT) or weak constraints in Answer Set Programming (ASP). Clearly each cardinality-optimal solution is also a subset-optimal one, and if the language also allows for the addition of particular restricting constructs (both MaxSAT and ASP do) then all subset-optimal solutions can be found by an iterative computation of cardinality-optimal solutions. As a showcase, the computation of preferred extensions of abstract argumentation frameworks using the proposed method is studied.

1 INTRODUCTION

In Artificial Intelligence, the task of set optimization, in the sense of finding a set that is minimal or maximal with respect to set inclusion, frequently occurs. There are famous examples such as Circumscription \cite{circumscription} or Model-based Diagnosis that involve set minimization. Computing preferred extensions of abstract argumentation frameworks \cite{argumentation} is an example that involves set maximization.

Often, set optimization is an element that creates difficulties in implementation and representation. For example, McCarthy had to resort to Second-order logic for defining Circumscription of First-order theories \cite{McCarthy1963}. Also for computing preferred extensions, relatively sophisticated techniques are required, for instance QBFs rather than propositional formulas \cite{McCarthy1963}.

There is another notion of set optimization, finding a set that has minimal or maximal cardinality, which has more readily available system support nowadays. The most prominent examples of languages and systems that support cardinality optimization are MaxSAT, Constraint Programming, and Answer Set Programming (ASP).

In this paper, we show how set optima can be computed by a general algorithm that employs cardinality optimizing subroutines, provided that the underlying languages allow for expressing simple constraints. Algorithm MCSes in Figure 2 of \cite{McCarthy1963}, which computes Minimal Correction Sets\textsuperscript{4} of a propositional formula, bears a few similarities to our algorithms. However, it is also different in several respects, most notably it is formulated for solving one particular problem only, assumes propositional formulas as the representation formalism, and does not employ a cardinality optimization oracle explicitly. To the best of our knowledge a general method applicable in a variety of representation formalisms and for unspecific set optimization problems settings has not been proposed in the literature before. We develop two instantiations of the general method, for MaxSAT and for ASP, and show that they are suitable.

There are two more recent software tools that also support computing set optimization problems when the underlying language is ASP: \textit{asprin} \cite{asprin} and D-FLAT\textsuperscript{2} \cite{D-FLAT2}. The scope of the tool \textit{asprin} is actually reasoning with preferences, but as a special case one can express preferences such that only set optimal solutions remain. The required preferences come with the predefined library of \textit{asprin}. The underlying algorithms of \textit{asprin} are very different from those presented in this paper. D-FLAT\textsuperscript{2} builds on dynamic programming and exploits tree decomposition in order to solve set optimization problems and is therefore also very different from the method that we will present in this paper. In the realm of ASP, the tool \textit{metasp} \cite{metasp} can be seen as a predecessor of \textit{asprin}, which does not seem to be maintained anymore. It relies on reification of rules and exploits a programming pattern known as saturation for set optimization, which is also very different from the method described in this paper.

We then turn our attention to a showcase application, computing preferred extensions of abstract argumentation frameworks. Dung’s theory of abstract argumentation \cite{argumentation} is a unifying framework able to encompass a large variety of specific formalisms in the areas of nonmonotonic reasoning, logic programming and computational argumentation. It is based on the notion of argumentation framework ($\textit{AF}$), consisting of a set of arguments and a binary attack relation between them. Arguments can thus be represented by nodes of a directed graph, and attacks by arcs. The nature of arguments is left unspecified: it can be anything from logical statements to informal natural language text. For instance, \cite{Cerutti2015} shows how argumentation can be efficiently used for supporting critical thinking and intelligence analysis in military-sensitive contexts.

Different \textit{argumentation semantics} declare the criteria to dete-

\textsuperscript{4} It is worth noticing that algorithms exploiting minimal correction sets have been proposed for computing argumentation semantics extensions, in particular for semi-stable, ideal, and eager semantics \cite{Eager2005}, but not for preferred semantics which is our main test-case in this paper.
mine which arguments emerge as “justified” among conflicting ones, by identifying a number of extensions, i.e. sets of arguments that can “survive the conflict together”. In [12] four “traditional” semantics were introduced, namely complete, grounded, stable, and preferred semantics. For a complete overview of subsequently proposed alternative semantics, the interested reader is referred to [3].

The main computational problems in abstract argumentation include decision—e.g. determine if an argument is in all the extensions prescribed by a semantics—and construction problems, and turn out to be computationally intractable for most argumentation semantics [13]. In this paper we focus on the extension enumeration problem, i.e. constructing all extensions for a given AF: its solution provides complete information about the justification status of arguments and allows for solving the other problems as well.

Our general method allowed for the definition and implementation of two novel algorithms for enumerating preferred extensions: \texttt{prefMaxSAT} (based on a MaxSAT solver) and \texttt{prefASP} (based on an ASP solver). Both are evaluated using benchmarks from the International Competition on Computational Models of Argumentation (ICMCA2015). We report on a variety of experiments: the first focuses on \texttt{prefASP} and starts with comparing the use of different solver configurations for \texttt{prefASP}, followed by a comparison of \texttt{prefASP} to \texttt{asprin} and D-FLAT2, and eventually comparing \texttt{prefASP} to the dedicated argumentation solver ASPARTIX-V. Eventually we compare \texttt{prefASP} and \texttt{prefMaxSAT} to the ICMCA2015 competition winner \texttt{CegarX}. The experiments show that despite their conceptual simplicity, our software tools are competitive with the best available ones.

## 2 ABSTRACT METHODOLOGY

The proposed methodology applies to a variety of knowledge representation formalisms, we therefore consider an abstract setting. We define a knowledge base \( K \) to be associated with a set \( \sigma(K) \) of solutions, and we assume that each \( s \in \sigma(K) \) is a set. We also use a set restriction operator \( _{\alpha} \) such that \( s_{\alpha} = s \cap \alpha \), the idea being that \( s_{\alpha} \) identifies the solution elements that are relevant for optimization. We also assume a composition operator \( K_1 \circ K_2 \) to be present that allows to compose two knowledge bases \( K_1 \) and \( K_2 \). We next define a few optimization criteria for solutions of knowledge bases.

### Definition 1
Let \( K \) be a knowledge base and \( R \) be a set (of elements occurring in solutions of \( K \)). Define:

- \( S^\text{max}_{R}(K) = \{ s \mid s \in \sigma(K), \exists s' \in \sigma(K) : s' \mid_{R} \supset s \mid_{R} \} \)
- \( S^\text{min}_{R}(K) = \{ s \mid s \in \sigma(K), \exists s' \in \sigma(K) : s' \mid_{R} \subset s \mid_{R} \} \)
- \( C^\text{max}_{R}(K) = \{ s \mid s \in \sigma(K), \exists s' \in \sigma(K) : |s' \mid_{R}| > |s \mid_{R}| \} \)
- \( C^\text{min}_{R}(K) = \{ s \mid s \in \sigma(K), \exists s' \in \sigma(K) : |s' \mid_{R}| < |s \mid_{R}| \} \)

While \( S^\text{min}_{R}(K) \) and \( S^\text{max}_{R}(K) \) occur in diverse applications of knowledge representation and reasoning, it often happens that the computational complexity of these tasks increases (under standard assumptions) compared to \( S^\text{min}_{R}(K) \) and \( S^\text{max}_{R}(K) \). For example, deciding whether \( \sigma(K) = \emptyset \) is co-NP-complete (in the size of \( K \)) if \( K \) is represented using a propositional formula and \( \sigma(K) \) is the set of satisfying assignments, where each assignment is represented as the set of true variables. In this setting, computing \( S^\text{max}_{R}(K) \) is then \( \Sigma^P_{2} \)-hard, as showing the optimality of a solution may require exponentially many co-NP checks, while \( C^\text{max}_{R}(K) \) is in \( \Delta^P_{2} \), requiring at most a polynomial number of co-NP checks. Note that this does not necessarily have practical consequences, because at the moment all known algorithms to solve these problems require at least exponential time.

There are also representational repercussions. Still assuming \( K \) to be a propositional formula, one cannot find a propositional formula of polynomial size that encodes any of \( S^\text{min}_{R}(K) \), \( S^\text{max}_{R}(K) \), \( C^\text{max}_{R}(K) \), and \( C^\text{min}_{R}(K) \) (if \( NP \neq \Sigma^P_{2} \)), which is currently unknown, but often conjectured. If \( K \) has been modelled using ASP, it is easy to encode \( C^\text{max}_{R}(K) \) and \( C^\text{min}_{R}(K) \) because of the availability of weak constraints (or optimization constructs). In fact, one can use ASP also for encoding \( S^\text{min}_{R}(K) \) and \( S^\text{max}_{R}(K) \), because ASP can express all problems in \( \Sigma^P_{2} \). We will discuss this further in Section 3.

In this paper, we relate \( S^\text{min}_{R}(K) \) to \( C^\text{max}_{R}(K) \) (and \( S^\text{max}_{R}(K) \) to \( C^\text{min}_{R}(K) \)). We first observe that each cardinality optimal solution is also subset optimal.

### Observation 1
For any knowledge base \( K \) and set \( R \), \( C^\text{max}_{R}(K) \subseteq S^\text{min}_{R}(K) \) and \( C^\text{min}_{R}(K) \subseteq S^\text{max}_{R}(K) \).

This observation holds because if any \( s \in C^\text{max}_{R}(K) \) were not in \( S^\text{min}_{R}(K) \), then there would be some \( s' \in \sigma(K) \) such that \( s' \mid_{R} \supset s \mid_{R} \) and clearly \( |s' \mid_{R}| > |s \mid_{R}| \) then holds (and symmetrically for minimization).

This implies that when the task is to compute one subset optimal solution, one can instead safely compute one cardinality optimal solution. When, however, the computational task involves an enumeration of all subset optimal solutions, one is faced with incompleteness, as not all subset optima are cardinality optimal.

### Example 1
Let \( K_1 \) be such that \( \sigma(K_1) = \{ \{ a, b \}, \{ b, c \} \} \) and let \( R_1 = \{ a, b, c \} \). Then \( C^\text{max}_{R_1}(K_1) = \{ \{ a \}, \{ b \} \} \).

This can be overcome by an iterative approach, in which first cardinality optimal solutions are computed. In the next stage, the knowledge base is extended in a way that it no longer admits the solutions already found or any subsets (for maximization) or supersets (for minimization) thereof.

### Definition 2
Given a knowledge base \( K \), a set \( R \), and a set \( S \subseteq \sigma(K) \), let \( N^\text{max}_{S}(K, R, S) \) denote a knowledge base such that \( \sigma(K \circ R, S) = \sigma(K) \setminus \{ s \mid s' \mid_{R} \subseteq s \mid_{R} \wedge s \in S \} \).

Symmetrically, let \( N^\text{min}_{S}(K, R, S) \) denote a knowledge base such that \( \sigma(K \circ R, S) = \sigma(K) \setminus \{ s \mid s' \mid_{R} \supset s \mid_{R} \wedge s \in S \} \).

It depends on the formalism used for the knowledge base, whether \( N^\text{max}_{S}(K, R, S) \) and \( N^\text{min}_{S}(K, R, S) \) can be created, and in particular whether they can be represented in a concise way. It also depends on the formalism whether there is a uniform way of encoding \( N^\text{max}_{S}(K, R, S) \) and \( N^\text{min}_{S}(K, R, S) \), or whether one has to rely on a representation that depends on the structure of \( K \).

The iterative approach is formalized for subset maximality in Algorithm 1 and for subset minimality in Algorithm 2. Note that practical algorithms will usually not collect all solutions in the output because of space considerations, but rather output them immediately as they are computed.

### Theorem 1
For a knowledge base \( K \) and set \( R \), Algorithm 1 computes \( S^\text{max}_{R}(K) \), and Algorithm 2 computes \( S^\text{min}_{R}(K) \).
Algorithm 1 Enumerating $S_R^{max}(K)$ by means of $C_R^{max}(K)$

1: Input: $K, R$
2: Output: $S_R^{max}(K)$
3: $K_i := K$
4: $S := \emptyset$
5: $S_i := C_R^{max}(K_i)$
6: while $S_i \neq \emptyset$ do
7:     $S := S \cup S_i$
8:     $K_i := K_i \setminus N^C(K_i, R, S_i)$
9:     $S_i := C_R^{max}(K_i)$
10: end while
11: return $S$

Algorithm 2 Enumerating $S_R^{min}(K)$ by means of $C_R^{min}(K)$

1: Input: $K, R$
2: Output: $S_R^{min}(K)$
3: $K_i := K$
4: $S := \emptyset$
5: $S_i := C_R^{min}(K_i)$
6: while $S_i \neq \emptyset$ do
7:     $S := S \cup S_i$
8:     $K_i := K_i \setminus N^C(K_i, R, S_i)$
9:     $S_i := C_R^{min}(K_i)$
10: end while
11: return $S$

Proof (Sketch) We show that Algorithm 1 computes $S_R^{max}(K)$. Symmetric arguments prove that Algorithm 2 computes $S_R^{min}(K)$.

We first observe that each assignment of variable $S_i$ contains only elements of $S_R^{max}(K)$. When variable $S_i$ is first initialized in line 5 of Algorithm 1, Observation 1 guarantees the claim. For each further assignment, by construction only $s \in \sigma(K)$ are assigned, and any such $s$ is such that $s' \in \sigma(K) : s' \not\subseteq s \not\subseteq s \not\subseteq \emptyset$ (otherwise $|s'| > |s|R|$ would hold). It is also clear that the algorithm terminates (if $\sigma(K)$ is finite).

Now observe that each $s \in S_R^{max}(K)$ is assigned once to $S_i$ in Algorithm 1. Indeed, the first assignment contains all elements in $S_R^{max}(K)$ that are of maximum cardinality, the next iteration contains all elements in $S_R^{max}(K)$ of the next-highest cardinality, and so forth down to the elements of $S_R^{max}(K)$ of least cardinality in the last assignment. In this way, all elements of $S_R^{max}(K)$ will be contained in $S$ when Algorithm 1 terminates.

It should be pointed out that Algorithms 1 and 2 also work when instead of $C_R^{max}(K_i)$ (or $C_R^{min}(K_i)$) any non-empty subset thereof is assigned to $S_i$ in lines 5 and 9. In particular, for the instantiation of the framework using MaxSAT that will be discussed in Section 3.1, many MaxSAT solvers calculate only one solution, rather than all.

Let us note that the number of subcalls to $C_R^{max}(K_i)$ (resp., $C_R^{min}(K_i)$) is at most $|S_R^{max}(K)|$ (resp., $|S_R^{min}(K)|$). The cardinality of these sets can be exponential in $K$. In the worst case means that in the worst case an exponential number of knowledge bases $N^C(K_i, R, S_i)$ (or $N^C(K_i, R, S_i)$) are composed to $K_i$, which could lead to a use of exponential space. Note however, that this only occurs if there is an exponential number of solutions to be generated by the algorithm. This only occurs if $s_1 \not\subseteq s_2 \not\subseteq \emptyset$ and $s_2 \not\subseteq s_1 \not\subseteq \emptyset$ holds for almost all solutions $s_1$ and $s_2$ of $\sigma(K)$.

Now that there is also a contrast to more traditional algorithms that invoke a co-NP oracle call for each $s \in \sigma(K)$, especially if they run a test on each subset of a found solution. In that setting, the number of subcalls that take exponential time will usually be much greater than $|S_R^{max}(K)|$ (resp., $|S_R^{min}(K)|$). We view this feature of our algorithm as one of the main advantages over more traditional methods.

Algorithm 1 bears some similarities to Algorithm MCSes in Figure 2 of [19]. Algorithm MCSes computes Minimal Correction Sets of a propositional formula. Apart from the fact that is solves a much more specific problem and assumes a specific knowledge representation formalism. In fact, it iteratively increases the cardinality of the (relevant portion of the) solution to be computed and enforces the cardinality by means of formulas thus mimicking the behaviour of a MaxSAT. The clauses that Algorithm MCSes adds follow the same idea of $N^C(K_i, R, S_i)$, and correspond to $N^C(K_i, R, S_i)$ that will be defined in Section 3.1.

3 CONCRETIZATIONS USING MaxSAT AND ASP

We now show how to instantiate the abstract method described in Section 2 using MaxSAT and ASP.

3.1 SAT and MaxSAT

In Boolean satisfiability (SAT) [7], one asks for satisfying assignments of a propositional formula $\varphi$. We will assume that the propositional variables are taken from the infinite set $\mathbb{L}$ and use the standard connectives $\neg, \land, \lor, \Rightarrow$. SAT solvers usually assume that $\varphi$ is represented in conjunctive normal form, as a set of clauses.

In this setting, the terminology of Section 2 is instantiated as follows: $K$ is a propositional formula $\varphi$ over $\mathbb{L}$, or a set of clauses over $\mathbb{L}$. The solutions $\sigma(K)$ are the satisfying assignments of $K$, represented as sets of propositional variables that are true in the respective assignment. The operation $K_i \circ K_j$ is either $K_i \land K_j$ if $K_i, K_j$ are formulae, or $K_i \cup K_j$ if $K_i, K_j$ are sets of clauses. In the sequel, we will assume the clause notation.

In order to encode $C_R^{max}(K)$ and $C_R^{min}(K)$, one can make use of weighted MaxSAT [18]. In weighted MaxSAT, one can assign a numerical weight to some of the clauses (known as soft clauses), and these clauses do not necessarily have to be satisfied. The solutions of a MaxSAT problem are determined among all assignments that satisfy the non-weighed clauses (hard clauses), and are those that maximize the sum of weights of the soft clauses.

Definition 3 Given a set of clauses $K$ and $R \subseteq \mathbb{L}$, we define the following MaxSAT problem

$$C_R^{max}(K) \equiv K \cup \{1 : \{r \mid r \in R\}$$

consisting of hard clauses $K$ and soft clauses $\{1 \mid r \in R\}$ (unit clauses) for each $r \in R$, all with the same positive weight 1 (any other weight could be used, as long it is the same for all soft clauses).

In a similar way,

$$C_R^{min}(K) \equiv K \cup \{1 : \{-r \mid r \in R\}$$

consists of hard clauses $K$ and soft clauses $\{-r \mid r \in R\}$ (unit clauses) for each $r \in R$, again all with the same weight.

It is clear that $C_R^{max}(K)$ corresponds to the solutions of $C_R^{max}(K)$ and $C_R^{min}(K)$ corresponds to the solutions of $C_R^{min}(K)$.

For obtaining $C_R^{max}(K)$ and $C_R^{min}(K)$, one would have to resort to Quantified Boolean Formulae (QBFs), which we do not discuss in detail in this paper.
Example 2 An instantiation of Example 1 would be the
\[ K_{\text{sat}} = \{\{\neg a, c\}, \{\neg b, \neg c\}, \{b, c\}\} \text{, with } \sigma(K_{\text{sat}}) = \{\{a, b\}, \{b, c\}\} \]. Then, with \( R_1 = \{a, b, c\} \), \( \mathcal{C}^\text{max}_{R_1, \text{sat}}(K_{\text{sat}}) = \{\{\neg a, c\}, \{\neg b, \neg c\}, \{b, c\}\} \) where \( h = \{a\}, \{a\}, 1 : \{b\}, 1 : \{c\} \), and the only optimal solution of \( C^\text{max}_{R_1, \text{sat}}(K_{\text{sat}}) \) is \( \{a\} \), which is equal to \( C^\text{max}_{R_1}(K_{\text{sat}}) \).

Finally, we consider how to encode \( \mathcal{N}^\text{sat}(K, R, S) \) and \( \mathcal{N}^\text{sat}(K, R, S) \) in SAT. For \( \mathcal{N}^\text{sat}(K, R, S) \), the idea is to require for each solution in \( S \) that at least an element of \( R \) not in that solution should be true. This inhibits the solution and any subset (restricted to \( R \)) of it. For \( \mathcal{N}(K, R, S) \), we require for each solution in \( S \) that at least one variable in \( R \) should be false. In this way the solution itself and any superset is inhibited.

**Definition 4** Given a set of clauses \( \mathcal{K}, R \subset \mathcal{L}, \text{ and } S \subset \sigma(\mathcal{K}), \) let

\[ \mathcal{N}^\text{sat}_{\text{sat}}(\mathcal{K}, R, S) = \bigwedge_{s \in S} \bigvee_{v \in R \setminus s} v \]

and in a similar way

\[ \mathcal{N}^\text{sat}_{\text{sat}}(\mathcal{K}, R, S) = \bigwedge_{s \in S} \bigvee_{v \in R \setminus s} \neg v \].

Note that these formulas are in clause form. It is clear that \( \mathcal{N}^\text{sat}_{\text{sat}}(\mathcal{K}, R, S) \) and \( \mathcal{N}^\text{sat}_{\text{sat}}(\mathcal{K}, R, S) \) restrict the solutions in the way required by \( \mathcal{N}^\text{sat}(K, R, S) \) and \( \mathcal{N}^\text{sat}(K, R, S) \).

**Example 3** Continuing Example 3, if \( S_1 = \{a, b\} \), then \( \mathcal{N}^\text{sat}_{\text{sat}}(\mathcal{K}, R, S_1) = \{\{c\}\} \), therefore the only optimal solution of \( \mathcal{C}^\text{max}_{R_1, \text{sat}}(\mathcal{K}_{\text{sat}}) \) is \( \{c\} \).

### 3.2 Answer Set Programming

In Answer Set Programming (ASP) one asks for the answer sets (often also called stable models) of a logic program. The full language specification can be found at https://www.mat.unical.it/aspcomp2013/ASPStandardization, below we provide a brief overview of the concepts relevant to this work.

The basic constructs in ASP logic programs are of the form

\[ b_1 | \ldots | b_n : \neg b_1, \ldots, \neg b_n, \text{not } b_{n+1}, \ldots, \text{not } b_n. \]

where \( 0 \leq k, 0 \leq n \leq k \) and the \( b_i \) and \( b_j \) are function-free first-order atoms. When \( k > 0 \), it is called a rule, otherwise a constraint.

If \( k = 1 \) and \( n = 0 \), the rule is called a fact. The part left of the construct : is called head, the part right of it is called body. Sets of rules and constraints are called programs.

Programs with variables are thought of as shorthand for their ground (variable-free) versions with respect to the Herbrand universe of the program. Answer sets are defined on ground programs: they are Herbrand models of the program, which satisfy an additional stability condition.

The language of ASP consists of quite a lot more constructs. One relevant for this paper is the weak constraint, which takes the form

\[ \sim b_1, \ldots, b_n, \text{not } b_{n+1}, \ldots, \text{not } b_n. \]

An interpretation that satisfies all literals to the right of \( \sim \) will incur a (uniform) penalty. Answer sets of programs with weak constraints are then those answer sets of the weak-constraint-free portion that minimize the penalties incurred by weak constraints.

For ASP, the terminology of Section 2 is instantiated as follows: \( \mathcal{K} \) in this setting is a weak-constraint-free program, and \( \sigma(\mathcal{K}) \) is the set of its answer sets. The operation \( \mathcal{K}_1 \circ \mathcal{K}_2 \) simply is the set union of the two programs \( \mathcal{K}_1 \) and \( \mathcal{K}_2 \).

In ASP, it is possible to encode \( C^\text{max}_{\text{sat}}(\mathcal{K}) \) and \( C^\text{min}_{\text{sat}}(\mathcal{K}) \) by means of weak constraints, similar to the MaxSAT approach described earlier.

**Definition 5** Given a set of clauses \( \mathcal{K}, R \subset \mathcal{L}, \) we define

\[ \mathcal{C}^\text{max}_{\text{R, sat}}(\mathcal{K}) = \{ \sim \text{not } r. \mid r \in R \} \]

and in a similar way

\[ \mathcal{C}^\text{min}_{\text{R, sat}}(\mathcal{K}) = \{ \sim r. \mid r \in R \}. \]

It is easy to verify that \( \mathcal{C}^\text{max}_{\text{R, sat}}(\mathcal{K}) \) corresponds to the answer sets of \( \mathcal{C}^\text{max}_{\text{R, sat}}(\mathcal{K}) \) and \( \mathcal{C}^\text{min}_{\text{R, sat}}(\mathcal{K}) \) corresponds to the answer sets of \( \mathcal{C}^\text{min}_{\text{R, sat}}(\mathcal{K}) \).

ASP also allows for encoding \( S^\text{R, sat}_{\text{sat}}(\mathcal{K}) \) and \( S^\text{R, sat}_{\text{sat}}(\mathcal{K}) \) in ASP. For \( \mathcal{N}^\text{sat}(\mathcal{K}, R, S) \), one requires for each solution in \( S \) that not all elements of \( R \) outside that solution should be false. This inhibits the solution itself and any subset (restricted to \( R \)) of it. For \( \mathcal{N}^\text{sat}(\mathcal{K}, R, S) \), we require for each solution in \( S \) that not all of its elements in \( R \) should be true. In this way the solution itself and any superset is inhibited.

**Definition 6** Given a set of clauses \( \mathcal{K}, R \subset \mathcal{L}, \) and \( S \subset \sigma(\mathcal{K}), \) let

\[ \mathcal{N}^\text{sat}_{\text{ASP}}(\mathcal{K}, R, S) = \{\sim \text{not } a_1, \ldots, \text{not } a_n. \mid s \in S, R \setminus s = \{a_1, \ldots, a_n\}\} \]

and in a similar way

\[ \mathcal{N}^\text{sat}_{\text{ASP}}(\mathcal{K}, R, S) = \{\sim a_1, \ldots, a_n. \mid s \in S, R \cap s = \{a_1, \ldots, a_n\}\}. \]

Again, it is easy to verify that \( \mathcal{N}^\text{sat}_{\text{ASP}}(\mathcal{K}, R, S) \) and \( \mathcal{N}^\text{sat}_{\text{ASP}}(\mathcal{K}, R, S) \) restrict the answer sets in the way required by \( \mathcal{N}^\text{sat}(\mathcal{K}, R, S) \) and \( \mathcal{N}^\text{sat}(\mathcal{K}, R, S) \).

### 4 METHODOLOGY SHOWCASE: ABSTRACT ARGUMENTATION

In this section we show how to use the proposed methodology for enumerating all preferred extensions of abstract argumentation frameworks. After a short background on abstract argumentation, we introduce a MaxSAT-based solver (\text{prefMaxSAT}) and an ASP-based solver (\text{prefASP}) that employ the methods described in Section 3.

#### 4.1 Background on Abstract Argumentation

An argumentation framework [12] consists of a set of arguments and a binary attack relation between them.

\footnote{In this paper we consider only finite sets of arguments: see [4] for a discussion on infinite sets of arguments.}
Definition 7 An argumentation framework (AF) is a pair $\Gamma = \langle A, \mathcal{R} \rangle$ where $A$ is a set of arguments and $\mathcal{R} \subseteq A \times A$. We say that $b$ attacks $a$ iff $\langle b, a \rangle \in \mathcal{R}$, also denoted as $b \rightarrow a$. The set of attackers of an argument $a$ will be denoted as $a^+ \triangleq \{ b : b \rightarrow a \}$, the set of arguments attacked by $a$ will be denoted as $a^- \triangleq \{ b : a \rightarrow b \}$.

Each AF has an associated directed graph where the vertices are the arguments, and the edges are the attacks.

As an intuitive example from [6], let $a$ be the argument “Patient has hypertension so prescribe betablockers;” $b$: “Patient has hypertension so prescribe betablockers;” and $c$: “Patient has emphysema which is a contraindication for betablockers.” Intuitively, assuming that only one treatment is possible at the very same time, $a$ attacks $b$ and vice versa, while $c$ suggests that $b$ should not be the case (c attacks b).

Therefore, let $\Gamma_M = \langle A_M, \mathcal{R}_M \rangle$ such that, $A_M = \{ a, b, c \}$ and $\mathcal{R}_M = \{ \langle a, b \rangle, \langle b, a \rangle \}$. $\Gamma_M$ is depicted in Fig. 1.

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

Definition 8 Given an AF $\Gamma = \langle A, \mathcal{R} \rangle$:

- $\{ a \} \subseteq A$ is a conflict-free set of $\Gamma$ if $\nexists b \in A \text{ s.t. } a \rightarrow b$;
- an argument $a \in A$ is acceptable with respect to a set $S \subseteq A$ if $\forall b \in A \text{ s.t. } a \rightarrow b$;
- the function $F_{\Gamma} : 2^A \rightarrow 2^A$ such that $F_{\Gamma}(S) = \{ a \mid a \text{ is an acceptable w.r.t. } S \}$ is called the characteristic function of $\Gamma$;
- a set $S \subseteq A$ is an admissible set of $\Gamma$ if $S$ is a conflict-free set of $\Gamma$ and every element of $S$ is acceptable with respect to $S$ of $\Gamma$.

In the AF $\Gamma_M$ of Fig. 1, $\{ a \}$ is an admissible set because it is conflict-free (there is no such attack $\langle a, a \rangle$) and each element of the set (i.e. $a$) is defended against the attack it receives (i.e. $a$ is attacked by $b$, but in turn, $b$ is attacked by $a$).

An argument semantics $S$ prescribes for any $\Gamma$, $\Gamma_M$ a set of extensions, denoted as $E_S(\Gamma)$, namely a set of sets of arguments satisfying the conditions dictated by $S$. Here we recall the definition of the grounded semantics, denoted as $\mathcal{G}\mathcal{R}$, and of the preferred semantics, denoted as $\text{PR}$.

Definition 9 Given an AF $\Gamma = \langle A, \mathcal{R} \rangle$:

- $\{ a \} \subseteq A$ is the grounded extension of $\Gamma$, if $S$ is the least (w.r.t. set inclusion) fixed point of the characteristic function $F_{\Gamma}$;
- $\{ a \} \subseteq A$ is a preferred extension of $\Gamma$, i.e. $S \in E_{\text{PR}}(\Gamma)$, if $S$ is a maximal (w.r.t. $\subseteq$) admissible set of $\Gamma$.

While $\{ a \}$ is an admissible set for $\Gamma_M$, it is not a preferred extension. In fact, $\{ c, a \}$ is also an admissible set which contains $\{ a \}$. Since there are no admissible supersets of $\{ c, a \}$, it is therefore maximal and thus a preferred extension, the only one for $\Gamma_M$.

### 4.2 Admissible Extensions in SAT and ASP

As discussed in [5, 11] the search for admissible sets can be encoded using propositional logic formulae.

\[
\bigwedge_{a \in A} \left( \bigvee_{b \in a^-} \neg v(b) \right) \land \left( \bigvee_{b \in a^+} \bigvee_{c \in b^-} v(c) \right)
\]

The models of $\text{admsat}_A$ correspond to the admissible sets of $\Gamma$.

For the example AF $\Gamma_M$, $\text{admsat}_M$ is

\[
\begin{align*}
&\langle a \rangle \land (\langle a \rangle \lor \langle v(a) \lor v(c) \rangle) \\
&\land (\langle b \rangle \lor (\neg \langle a \rangle \land \neg \langle c \rangle)) \\
&\land (\langle v(b) \lor (\langle v(b) \land \bot \rangle) \land \\
&\langle v(c) \lor \top \rangle) \land (\langle v(c) \lor \top \rangle)
\end{align*}
\]

For ASP, an encoding for admissible extensions is rather straightforward, see [14, 11].

Definition 10 Given an AF $\Gamma = \langle A, \mathcal{R} \rangle$, for each $a \in A$ a fact

\[
\text{arg}(a).
\]

can be created and for each $\langle a, b \rangle \in \mathcal{R}$ a fact

\[
\text{att}(a, b).
\]

can be created (this corresponds to the apx file format in the ICCMA competition). Together with the program

\[
\begin{align*}
&\text{in}(X) : \neg \text{not out}(X), \text{arg}(X). \\
&\text{out}(X) : \neg \text{not in}(X), \text{arg}(X). \\
&\text{in}(X), \text{in}(Y), \text{att}(X,Y). \\
&\text{defeated}(X) : \neg \text{in}(Y), \text{att}(Y,X). \\
&\text{not_defined}(X) : \text{att}(Y,X), \text{not_defined}(Y). \\
&\text{in}(X), \text{not_defined}(X).
\end{align*}
\]

we form $\text{admsat}_A$ and there is a one-to-one correspondence between answer sets of $\text{admsat}_A$ and admissible extensions.

Different from the SAT encoding, the ASP encoding only changes facts for different AFs, the main program remains equal.

### 4.3 Preferred Extensions via Algorithm 1

We can now use our methodology in order to step from admissible to preferred extensions. Indeed, if $K$ encodes admissible extensions of an AF and $\mathcal{R}$ is the language part encoding the extensions, then $\mathcal{R}^w(K)$ encodes preferred extensions. We can then use Algorithm 1 to compute them. The two concretisations then give rise to two solvers, $\text{prefMaxSAT}$ and $\text{prefASP}$, described in the following.

For $\text{prefMaxSAT}$, given an AF $\Gamma = \langle A, \mathcal{R} \rangle$ and $v : A \rightarrow \mathcal{L}$, we use Algorithm 1 with input $K$ being $\text{admsat}_A$ and input $R$ being the image of $v$ in the following referred to as $\text{img}(v)$. In lines 5 and 9 of Algorithm 1, we use a MaxSAT solver for obtaining all answer sets of $C_{\text{max}}^{\text{img}(v)}(\mathcal{R}, K_i)$. In line 8, $\mathcal{N}_{\text{sat}}(K_i, I(A), S_i)$ is used.

For $\text{prefASP}$, given an AF $\Gamma = \langle A, \mathcal{R} \rangle$, we use Algorithm 1 with input $K$ being $\text{admsat}_A$ and input $R$ being $\{ \text{in}(a) \mid a \in A \}$, in the following referred to as $\mathcal{I}(A)$. In lines 5 and 9 of Algorithm 1, we use an ASP solver for obtaining all answer sets of $C_{\text{max}}^{\text{sat}}(\mathcal{I}, K_i)$. In line 8, $\mathcal{N}_{\text{sat}}(K_i, I(A), S_i)$ is used.

Apart from the encodings and underlying solvers, there is also a difference in the fact that $\text{prefASP}$ computes all cardinality-optimal solutions in one go, while $\text{prefMaxSAT}$ computes one at a time.
5 EXPERIMENTAL ANALYSIS

In order to evaluate the efficiency of the introduced algorithms, we have carried out an experimental analysis where performance is analyzed from different perspectives. After describing the general setup, we first report on an experiment for choosing a parameter setting in the backend solver of prefASP. Next we report on a comparison of prefASP with asprin and D-FLAT². As discussed in the Introduction, these systems also support set optimization in an ASP setting, but use very different underlying algorithms. This is followed by a comparison to dedicated argumentation systems, in which we compare prefASP with ASPARTIX-V, since ASPARTIX-V is also based on ASP. Finally, we compare both prefASP and prefMaxSAT with Cegarix, which is the state-of-the-art solver, in the sense that it won the ICCMA2015 competition. These comparisons are deliberately split up into small pairings, in order to have a crisper picture of the relative performance measure. Indeed, the IPC score depends on the solvers considered in the comparison, while PAR10 and coverage are specific to a single solver. Experiments do not need to be re-run for presenting results together; however, comparisons would be less informative given the changes in the IPC score.

5.1 Experimental Settings

prefMaxSAT has been implemented in C++, and exploits the ASPrin MaxSAT solver [1]. It should be noted that prefMaxSAT can be used with any MaxSAT system supporting the DIMACS format. The ASP-based algorithm prefASP has been implemented as a bash script using basic tools like sed and grep and exploits clingo 4.5.2 [16] as ASP solver. The experiments were conducted on a cluster with computing nodes equipped with 2.5 GHz Intel® Core 2 Quad Processors, 4 GB of RAM and Linux operating system. A cutoff of 900 seconds—wallclock time—was imposed to compute the preferred extensions for each AF. For each involved solver we recorded the overall result: success (if it finds each preferred extension), crashed, timed-out or ran out of memory. In fact, in our experimental evaluation all the unsuccessful runs are due to time-out. Experiments have been conducted on the ICCMA2015 benchmarks [23], which is a set of randomly generated 192 AFs. They have been generated considering three different graph models, in order to provide different levels of complexity. More details can be found on the ICCMA website.

The performance measures reported in this paper are the Penalised Average Runtime and the International Planning Competition (IPC) score. The Penalised Average Runtime (PAR) score is a real number which counts (i) runs that fail to find all the preferred extensions as ten times the cutoff time (PAR10) and (ii) runs that succeed as the actual runtime. PAR scores are commonly used in automated algorithm configuration, algorithm selection, and portfolio construction, because using them allows runtime to be considered while still placing a strong emphasis on high instance set coverage. The IPC score, borrowed from the Planning community, is defined as follows:

- For each valid 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).

\[ \text{IPC score} = \sum_{\text{successful test cases}} \frac{1}{(1 + \log_{10}(T/T^+))} \]

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 second get by default the maximal score of 1.

The IPC score for a system is the sum of its scores over all the valid test cases. It should be noted that the IPC score depends on the ensemble of tested systems, that is, it is a relative measure, which depends on the experimental context. Indeed, in the following, the IPC scores of prefASP vary depending on the different experimental settings. In contrast to this, the PAR10 score of prefASP remains equal, as this is an absolute score.

5.2 Comparison of prefASP Using Different Solver Configurations

Clingo offers several different solver configurations (inherited from the clasp solver used in clingo) which correspond to different heuristic and search setups, see [15] for detailed explanations. As a first analysis, we investigated how robust prefASP is with respect to these configurations, and, as a by-product, we determined the best-performing configuration to be used for comparison with other systems.

Table 1. Comparison of different Clingo solver configurations, that can be exploited by prefASP, on the ICCMA2015 benchmarks. Results are shown in terms of IPC score (maximum achievable is 192.0), percentages of success (% Success) and PAR10.

<table>
<thead>
<tr>
<th>Configurations</th>
<th>IPC score</th>
<th>% Success</th>
<th>PAR10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crafty</td>
<td>183.7</td>
<td>100.0</td>
<td>23.1</td>
</tr>
<tr>
<td>Frumpy</td>
<td>178.6</td>
<td>99.5</td>
<td>75.0</td>
</tr>
<tr>
<td>Jumpy</td>
<td>172.0</td>
<td>99.3</td>
<td>82.3</td>
</tr>
<tr>
<td>Many</td>
<td>177.4</td>
<td>99.5</td>
<td>84.8</td>
</tr>
</tbody>
</table>

Table 1 shows a comparison of the different solver configurations, in terms of IPC score, percentage of successfully analysed AFs and PAR10, on the ICCMA2015 benchmarks. It can be observed that all configurations perform relatively similar to each other, implying that the particular chosen configuration is not critical for the performance of prefASP.

However, there is one winning configuration: the Crafty configuration allows prefASP to enumerate preferred extensions of all the considered AFs, and to provide solutions faster. This configuration is geared towards “crafted” problems, which also makes sense in the context of the considered benchmarks. Therefore, in the rest of the experimental analysis the Crafty configuration will be considered for prefASP.

5.3 Comparison with Existing General Algorithms

We now turn to general tools that allow for easy representation and effective solution of subset optimization problems. To the best of our knowledge, the only tools of this kind are asprin [9] (with its predecessor metasp) and D-FLAT² [8], which we have discussed in the Introduction. Actually, D-FLAT² uses the computation of preferred extensions as an example. When we followed the instruction provided by the
authors\textsuperscript{7}, we were able to compute the preferred extensions of small AFs, but the system was already struggling with resource consumption on medium sized instances. On the ICCMA2015 benchmarks, the system ran out of memory very quickly, and we did not obtain any solutions for any of the ICCMA2015 benchmarks. This is probably due to the fact that ICCMA instances have large tree-width and D-FLAT\textsuperscript{2} relies heavily on tree decomposition. For this reason, in the remainder of this section we focus our comparison on asprin.

As asprin is based on ASP, the most natural comparison is against the ASP implementation of the proposed approach, namely prefASP. For asprin, we used clingo 4.5.2, the same version that is used as a backend for prefASP.

As input to asprin we use the program adnasp\textsubscript{p} in Definition 11 together with the following preference definition:

\[
\text{//preference}\,(p_1,\text{superset})\{ \\
\text{in}(X) : \text{arg}(X) \\
\}\).
\]

which makes asprin compute those answer sets, which are subset maximal for atoms with the predicate in.

Table 2. Comparison of prefASP and asprin, on the ICCMA2015 benchmarks. Results are shown in terms of IPC score (maximum achievable is 192.0), percentages of success (% Success) and PAR10.

<table>
<thead>
<tr>
<th></th>
<th>IPC score</th>
<th>% Success</th>
<th>PAR10</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefASP</td>
<td>191.2</td>
<td>100.0</td>
<td>23.1</td>
</tr>
<tr>
<td>asprin</td>
<td>157.8</td>
<td>100.0</td>
<td>44.9</td>
</tr>
</tbody>
</table>

The results of comparison between asprin and prefASP performed on the ICCMA2015 benchmarks for enumerating preferred extensions are shown in Table 2. Results indicate that the proposed prefASP system is significantly faster: prefASP achieves an IPC score of 191.2 versus 157.8 of asprin. According to the results, prefASP is the fastest system on 187 of the considered AFs. This is also confirmed by the PAR10 scores; on average asprin is about 20 seconds slower than prefASP, while in terms of coverage, both the considered systems are able to successfully analyse all the 192 AFs of the benchmark set. With regard to the observed performance difference, our conjecture is that asprin proves properties for more subsets of each answer set/extension than prefASP has to.

5.4 Comparison with Abstract Argumentation Algorithms Based on the Same Approaches

According to the results of ICCMA2015 \textsuperscript{[23]}, ASPARTIX-V \textsuperscript{[22]} is the ASP-based abstract argumentation solver that showed the best performance in the preferred enumeration track.\textsuperscript{8} Table 3 presents the results of a comparison between prefASP and ASPARTIX-V performed on the ICCMA2015 benchmarks. Presented results indicate that prefASP is faster, both in terms of IPC and PAR10 scores. Remarkably, prefASP is able to successfully analyse a larger number of AFs (100.0\% against 94.0\%).

\textsuperscript{7} D-FLAT\textsuperscript{2} software and instructions have been retrieved from https://github.com/hmarkus/dflat-2 in March 2016.

\textsuperscript{8} We are not aware of any existing solver able to enumerate preferred extensions in a way directly comparable to prefMaxSAT.

At a closer look, it is noticeable that—among ICCMA2015 frameworks—the AFs with a very large grounded extension and many nodes in general\textsuperscript{9} are very challenging for ASPARTIX-V, while the proposed prefASP solver is able to quickly and effectively analyse also such large frameworks.

5.5 Comparison with the State of the Art Solver

In this analysis we compare prefASP and prefMaxSAT with the winner of the the ICCMA2015 track on enumerating preferred extensions, Cegartix \textsuperscript{[10]}. Table 4 shows the performance of considered solvers in terms of IPC score, percentage of successfully analysed AFs and PAR10. prefASP performs significantly better than prefMaxSAT. This is possibly due to the fact that each preferred extension results from an execution of the MaxSAT solver; and a final run is needed in order to demonstrate that no other extensions exist. Therefore, the number of MaxSAT calls is exactly the number of preferred extensions plus one. The generated MaxSAT formulæ tend to be large on the considered benchmarks; therefore, the added overhead can be remarkable.

<table>
<thead>
<tr>
<th></th>
<th>IPC score</th>
<th>% Success</th>
<th>PAR10</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefASP</td>
<td>161.7</td>
<td>100.0</td>
<td>23.1</td>
</tr>
<tr>
<td>prefMaxSAT</td>
<td>115.3</td>
<td>85.0</td>
<td>1423.8</td>
</tr>
<tr>
<td>Cegartix</td>
<td>188.9</td>
<td>100.0</td>
<td>15.2</td>
</tr>
</tbody>
</table>

Interestingly, the performance of prefASP is comparable to the performance of Cegartix; according to PAR10, prefASP needs on average 8 seconds more to enumerate the preferred extensions. Moreover, by re-running the top solvers that took part in this track of ICCMA2015, we observed that prefASP would have been ranked second. This is an impressive achievement, considering that the described algorithm: (i) is very general, in the sense that it does not exploit any argumentation-specific knowledge; (ii) is very easy to implement, particularly in the ASP configuration; and (iii) has been implemented as a prototype, without attention on software engineering techniques for improving performance.

6 CONCLUSIONS AND FUTURE WORK

We have proposed a general methodology for solving subset optimality problems by means of iteratively solving cardinality optimality problems. This approach is motivated by the availability of efficient
systems that support finding cardinality optimal solutions, namely MaxSAT solvers and ASP solvers supporting weak constraints.

As a methodology showcase we have produced two prototype systems, prefMaxSAT and prefASP, for enumerating preferred extensions of abstract argumentation frameworks. While the algorithms are general and easy to implement, an experimental analysis showed that they are competitive with the state-of-the-art system, which is specialized for this particular problem. On this showcase, our methods also prove higher performance than the existing general methods for computing subset optimal solutions of answer set programs, viz. asprin and D-FLAT’2.

There are quite many opportunities for future work. As discussed in [19], retain information between calls could help the performance, and we might investigate it in future work. However, this impact on the modularity of the concrete approach—i.e., a solver must be integrated in the framework—and can potentially reduce the generality of the overall framework.

Apart from tuning the prototype implementations of prefMaxSAT and prefASP to improve their performance, we intend to apply the methodology also to other application domains. Diagnosis or minimal model computation are immediate candidates. Another possibility is integrating our algorithm into a system like asprin, or one that supports the same input language.

The methodology would also allow for computing $\Sigma^p_1$-hard problems when using ASP, which would be interesting to explore, as it would give rise to alternatives to implementations relying on QBFs. It would also be worthwhile to explore whether the general methodology can be used also with formalisms different from MaxSAT and ASP, which would open entirely new avenues.

Given that there are a few similarities to Algorithm MCSes in [19], a comparison with the CAMUS system described in that paper could be attempted. Looking at the other direction, evaluating whether some implementation techniques in this area, e.g. in [21], can be generalized to our less specialized setting could be pursued as well.

Acknowledgements

The authors would like to acknowledge the use of the University of Huddersfield Queensgate Grid in carrying out this work.

The authors also thank the anonymous reviewers for their helpful comments.

REFERENCES


