Encoding Argumentation Semantics by Boolean Algebra

Fuan PU, Guiming LUO, Nonmembers, and Zhou JIANG, Student Member

SUMMARY In this paper, a Boolean algebra approach is proposed to encode various acceptability semantics for abstract argumentation frameworks, where each semantics can be equivalently encoded into several Boolean constraint models based on Boolean matrices and a family of Boolean operations between them. Then, we show that these models can be easily translated into logic programs, and can be solved by a constraint solver over Boolean variables. In addition, we propose some querying strategies to accelerate the calculation of the grounded, stable and complete extensions. Finally, we describe an experimental study on the performance of our encodings according to different semantics and querying strategies.

key words: argumentation framework, acceptability semantics, Boolean algebra, Boolean constraint solver, logic programming, querying strategies

1. Introduction

Argumentation is a reasoning model based on constructing and comparing arguments about conflicting options and opinions. The most popularly used framework to talk about general issues of argumentation is that of Dung’s abstract argumentation frameworks (AFs) [1], which consists of a set of arguments and a binary relation that represents the conflicting arguments. It offers a series of extension-based semantics for solving inconsistent knowledge by selecting acceptable subsets. Intuitively, an acceptable set of arguments must be in some sense coherent (no attacks between its arguments, i.e., conflict-free) and strong enough (e.g., able to defend itself against all attacking arguments, i.e., admissible). Based on these, various semantics have been established, such as grounded, complete, and stable (see [2] for an overview). However, finding extensions can be a complex procedure when done without any computational help, when the AF contains numerous arguments and attacks [3].

The main goal of this paper is to demonstrate how Boolean algebra can be used for representing and solving such complex problems in the research of argumentation. We propose to represent a subset of arguments by a Boolean vector, attack relations by a Boolean matrix. Moreover, we introduce a series of Boolean operations, such as element-wise operations on Boolean vectors, and the Boolean matrix multiplication operation, which can be used to represent the neutrality, innocuousness and characteristic functions of AFs by Boolean functions, three cornerstones of Dung’s acceptability semantics. We show that with these representations and some interesting properties, each semantics can be equivalently encoded into more than one Boolean constraint model. Then, finding the extensions of a semantics is equivalently to finding the Boolean vectors that satisfy all constraints in one of its Boolean constraint models. To solve these constraint models, in this paper, we employ a Constraint Logic Programming over Boolean variables (CLPB) system, which is an algebraically oriented Constraint Programming solver and has abilities to handle any Boolean expressions with little conversation. In addition, we propose some querying strategies to accelerate the calculation of stable and complete semantics by initializing some Boolean variables based on the grounded extension, which is optionally calculated by iterating the Boolean-algebra-based characteristic function using bit-vector operations. At last, we describe an experimental study on the performance of the proposed encodings according to different semantics and querying strategies.

The rest of this paper is organized as follows. Section 2 presents some basic concepts about AF. Section 3 introduces a Boolean algebra approach to represent these basic concepts. Section 4 encodes Dung’s semantics into Boolean constraint models, which are translated into logic programs based on CLPB in Sect. 5. Section 6 gives some experimental results, and Sect. 7 concludes.

2. Abstract Argumentation Framework

In this section, we briefly outline key elements of Dung’s abstract argumentation frameworks [1]:

Definition 1. An abstract argumentation framework (AF) is a tuple \( \Delta = (X, R) \) where \( X \) is a finite set of arguments and \( R \subseteq X \times X \) is a binary relation, called attack relation.

For two arguments \( x, y \in X \), \((x, y) \in R\) means that \( x \) attacks \( y \), or \( y \) is an attacker of \( x \). Often, we write \((x, y) \in R\) as \( xRy \). An AF can be represented by a directed graph, whose nodes are arguments and edges represent attack relations. We denote by \( R^-(x) \) (respectively, \( R^+(x) \)) the subset of \( X \) containing those arguments that attack (respectively, are attacked by) \( x \in X \), extending this notion in the natural way to sets of arguments, so that for \( S \subseteq X \), \( R^-(S) \equiv \{ x \in X : \exists y \in S \text{ such that } xRy \} \) and \( R^+(S) \equiv \{ x \in X : \exists y \in S \text{ such that } yRx \} \).

The justified arguments are evaluated based on subsets of \( X \) (called extensions) defined under a range of semantics. The arguments in an extension are required to not at-
tack each other (extensions are conflict-free), and attack any argument that in turn attacks an argument in the extension (extensions reinstate or defend their contained arguments). Now, let us formally characterise these two fundamental notions of conflict-free and defend as below:

**Definition 2.** Let \( \Delta = (X, R) \) be an AF, \( S \subseteq X \) and \( x \in X \).

- \( S \) is conflict-free iff \( \exists x, y \in S \) such that \( x R y \) (i.e., \( S \cap R^+(S) = \emptyset \), or, equivalently, \( S \cap R^-(S) = \emptyset \)).
- \( S \) defends \( x \) iff \( \forall y \in X \) if \( y R x \) then \( \exists z \in S \) such that \( z R y \), (i.e., \( R^-(x) \subseteq R^+(S) \)).

Dung introduces a function for an AF, called characteristic function. It applies to some set \( S \subseteq X \) and returns all arguments that are defended by \( S \).

**Definition 3.** Let \( \Delta = (X, R) \) be an AF. The characteristic function of \( \Delta \) is \( F_\Delta : 2^X \rightarrow 2^X \) such that, given \( S \subseteq X \),

\[
F_\Delta(S) = \{ x \in X : S \text{ defends } x \}.
\]

An argument \( x \in X \) is not attacked by a set \( S \subseteq X \) if no argument in \( S \) attacks \( x \) (i.e., \( \forall y \in S \) such that \( y R x \), alternatively, \( x \notin R^+(S) \)). Then, we can define a function, which is applied to some set \( S \subseteq X \), returns all arguments that are not attacked by \( S \). This function, called neutrality function, was introduced in [4]. Similarly, we also define another function, called innocuousness function, which is applied to some set \( S \subseteq X \) and returns all arguments that do not attack \( S \).

**Definition 4.** Let \( \Delta = (X, R) \) be an AF. The neutrality function of \( \Delta \) is \( N_\Delta : 2^X \rightarrow 2^X \) such that, given \( S \subseteq X \),

\[
N_\Delta(S) = \{ x \in X : x \notin R^+(S) \}.
\]

Equivalently, we can write the neutrality function as \( N_\Delta(S) \equiv \overline{R^+(S)} \), where the bar on a set means the complement of the set relative to \( X \). The innocuousness function of \( \Delta \) is \( I_\Delta : 2^X \rightarrow 2^X \) such that, given \( S \subseteq X \),

\[
I_\Delta(S) = \{ x \in X : x \notin R^-(S) \}.
\]

From now on, we will omit the subscript \( \Delta \) from \( F_\Delta, N_\Delta \) and \( I_\Delta \) if there is no danger of ambiguity.

**Proposition 1.** Let \( \Delta = (X, R) \) be an AF. For any \( S \subseteq X \), the following holds: (i) \( S \) is conflict-free iff \( S \subseteq N(S) \); (ii) \( S \) is conflict-free iff \( S \subseteq I(S) \); (iii) \( F(S) = N(N(S)) \).

Now, we can define the extensions, i.e., the justified arguments, of an AF \( \Delta \) under Dung’s acceptability semantics, in terms of the fixpoints (i.e., \( S = F(S) \) and \( S = N(S) \)) or the post-fixpoints (i.e., \( S \subseteq F(S) \) and \( S \subseteq N(S) \)) of the characteristic and neutrality functions as below (for more details see [2]):

**Definition 5 (Acceptability semantics).** Let \( \Delta = (X, R) \) be an AF, and a subset \( E \subseteq X \).

- **CF** \( E \) is a conflict-free extension of \( \Delta \) iff \( E \subseteq N(E) \).
- **ST** \( E \) is a stable extension of \( \Delta \) iff \( E = N(E) \).
- **AD** \( E \) is an admissible extension of \( \Delta \) iff \( E \subseteq N(E) \) and \( E \subseteq F(E) \).
- **CO** \( E \) is a complete extension of \( \Delta \) iff \( E \subseteq N(E) \) and \( E = F(E) \).
- **GR** \( E \) is a grounded extension of \( \Delta \) iff \( E \) is the minimal (w.r.t. \( \subseteq \)) complete extension.

Let \( \sigma \) be an acceptability semantics in \{CF, ST, AD, CO, GR\}, we denote the collection of all \( \sigma \)-extensions of \( \Delta \) by \( E_\sigma(\Delta) \).

A stable extension is a maximal (w.r.t. \( \subseteq \)) conflict-free extension. The existence of stable extensions is not guaranteed. If there exists a stable extension, it must be a complete extension (but not vice versa), and thus it is admissible. Each admissible set is contained in a complete extension. Every AF possesses at least one complete extension. The grounded extension is conflict-free and unique, and always exists, for proofs presented in [1, Thm. 25]. It can be computed by iteratively applying the characteristic function \( F \) from the empty set until we reach a fixed point. We summarise some properties of these semantics from [1, 2, 5] as below:

**Proposition 2.** Let \( \langle X, R \rangle \) be an AF, and \( S, E \subseteq X \), then:

(i) \( S \) is an admissible extension iff \( S \subseteq N(S) \) and \( R^-(S) \subseteq R^+(S) \).

(ii) \( S \) is an admissible extension iff \( S \subseteq F(S) \cap N(S) \).

(iii) \( S \) is an admissible extension iff \( S \subseteq F(S) \cap N(S) \).

(iv) \( S \) is a complete extension iff \( S = F(S) \cap N(S) \).

(v) \( S \) is a complete extension iff \( S = F(S) \cap N(S) \).

(vi) If \( S \) is the (unique) grounded extension, then \( S \) is included in each complete extensions, i.e.,

\[
S \subseteq E, \forall E \in E_{CO}(\Delta)
\]

and arguments that attacked by \( S \) are not included in any complete extension, formally,

\[
R^+(S) \notin E, \forall E \in E_{CO}(\Delta)
\]

They are also hold for stable semantics if \( \Delta \) exists stable extensions, i.e., \( E_{ST}(\Delta) \neq \emptyset \).

Note that most semantics are multi-extension semantics. That is, there is not always an unique extension induced by the semantics. In order to reason with multi-extension semantics, usually, one takes either a credulous or skeptical perspective, i.e., an argument \( x \in X \) is credulously (respectively, skeptically) justified under some semantics \( \sigma \in \{ CF, ST, AD, CO, GR \} \) if \( x \) belongs to at least one (respectively, all) \( \sigma \)-extensions of \( \Delta \).

**Example 1.** Consider the AF \( \Delta = (X, R) \), depicted in Fig. 1, in which \( X = \{ x_1, x_2, x_3, x_4, x_5 \} \) and \( R = \{ x_1 R x_2, x_2 R x_3, x_2 R x_4, x_2 R x_5, x_5 R x_3 \} \). We have the conflict-free extensions \( E_{CF}(\Delta) = \{ \emptyset \} \), \( x_3 R x_2, x_3 R x_4, x_3 R x_3, x_4 R x_5, x_5 R x_3 \} \). The stable extensions \( E_{ST}(\Delta) = \{ \{ x_1, x_4 \} \} \), the admissible extensions \( E_{AD}(\Delta) = \{ \emptyset, \{ x_1 \}, \{ x_3 \}, \{ x_4 \} \} \) and the complete extensions \( E_{CO}(\Delta) = \{ \{ x_1 \}, \{ x_1, x_3 \}, \{ x_1, x_4 \} \} \). The processes to compute the grounded extension for \( \Delta \) are listed as: \( F(\emptyset) = \{ x_1 \}, F(x_1) = \{ x_1 \} \). Therefore, the grounded extension is \( E_{GR}(\Delta) = \{ x_1 \} \).

![Fig. 1 A simple example of AF](image-url)
3. Representing AFs via Boolean Algebra

This section introduces a Boolean algebra approach to represent the basic notations of Dung’s AFs using Boolean matrices and Boolean operations. This is mainly based on the following considerations: (1) Boolean matrices can be easily represented and manipulated in computers; (2) Boolean matrices can describe information in a compact way, e.g., bit vectors and bit matrices; (3) a numerous tools can be used to solve Boolean algebra problems.

Before proceeding let us recall the concept of Boolean algebra. A Boolean algebra is a domain consisting of exactly two elements, 0 and 1, indicating the truth values false and true, and a family of operations over the domain.

Definition 6 (Boolean algebra). A Boolean algebra is a four-tuple $\Lambda = (\mathcal{B}, +, *, \neg)$, in which $\mathcal{B} = \{0, 1\}$ is the Boolean domain, $+$ (logical OR) and $*$ (logical AND) are two binary operations, $\neg$ (logical NOT) is an unary operation on $\mathcal{B}$, and they are defined as, for two Boolean variables $a, b \in \mathcal{B}$,

- $a + b \overset{\text{def}}{=} 0$ if $a$ and $b$ have value 0; otherwise $a + b \overset{\text{def}}{=} 1$;
- $a * b \overset{\text{def}}{=} 1$ if $a$ and $b$ have value 1; otherwise $a * b \overset{\text{def}}{=} 0$;
- $\neg a \overset{\text{def}}{=} 0$ if $a$ is 1, and $\neg a \overset{\text{def}}{=} 1$ if $a$ is 0.

Note that $+$ and $*$ satisfy the idempotent, commutative, absorption, and associative laws, and are mutually distributive. $+$, $*$ and $\neg$ are three basic operations of Boolean algebra. In the following, we also introduce three relational operations for Boolean algebra, which can be built up from the basic operations by composition:

- $a \leq b \overset{\text{def}}{=} \neg a + b$, called logical implication. It is commonly written as $a \rightarrow b$;
- $a \oplus b \overset{\text{def}}{=} (a + b) * (\neg a + \neg b)$, called logical disequivalence (or exclusive OR, XOR). It is 1 if exactly one of $a$ and $b$ is 1.
- $a \equiv b \overset{\text{def}}{=} (a + b) + (\neg a + \neg b)$, called logical equivalence. It is 1 just if $a$ and $b$ have the same value. Thus, it is always interpreted as the equality relation, and can be defined by the complement of XOR: $a \equiv b \overset{\text{def}}{=} \neg (a \oplus b)$ (or, $a \equiv b \overset{\text{def}}{=} \neg a \oplus b$, or $a \equiv b \overset{\text{def}}{=} a \oplus \neg b$).

These relational operations can be viewed as special cases of logical predicates. They usually evaluate to true or false, relying on the conditional relationship between the two operands holds or not. For more details about the Boolean algebra refer to [6].

Now, we present the Boolean matrix and matrix representation of an argument set and an argument graph.

Definition 7. Let $\Delta = (X, R)$ be a finite AF with $X = \{x_1, x_2, \cdots, x_n\}$, and $S \subseteq X$. Then $S$ can be represented by an $n \times 1$ Boolean vector $s \in \mathcal{B}^{\text{vec}}$, whose row indices index the elements in $X$, such that the $i^{th}$ component of $s$, denoted by $s_i$, is defined by

$$s_i \overset{\text{def}}{=} \begin{cases} 1 & \text{if } x_i \in S \\ 0 & \text{if } x_i \not\in S \end{cases}$$

Clearly, $s_i = 1$ indicates the presence and $s_i = 0$ the absence of $x_i$ in the set $S$. If $S = \emptyset$, then all components of $s$ are 0s, denoted by $\mathbf{0}^n$, and if $S = X$, i.e., the universal set, then all components of $s$ are 1s, denoted by $\mathbf{1}^n$.

The attack relation $R$ on $X$ for $\Delta$ can be represented by an $n \times n$ Boolean matrix $A \in \mathcal{B}^{n \times n}$, whose row and column indices index the elements of $X$, respectively, such that the entry $A_{ij}$ is defined by

$$A_{ij} \overset{\text{def}}{=} \begin{cases} 1 & \text{if } x_j R x_i \\ 0 & \text{otherwise} \end{cases}$$

We call $A$ as the attack matrix of $\Delta$. It can be seen that $A$ is the transpose of the adjacency matrix of the argument graph of $\Delta$, and can capture all attack relations of $\Delta$.

Example 2. Consider the AF $\Delta = (X, R)$ shown in Fig. 1. Let $S = \{x_1, x_3\}$, then the Boolean vector representation of $S$ w.r.t. $X$ is $s = [1, 0, 1, 0, 0]^T$. Assume the Boolean vector $t = [1, 0, 0, 1, 0]^T$, then its corresponding argument set is $T = \{x_1, x_4\}$. By definition, the attack matrix $A$ of $\Delta$ is

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

All Boolean vectors of a given length form an element-wise Boolean algebra, that is, any $n$-ary Boolean operations can be applied to $n$-dimensional Boolean vectors at one time. Therefore, we can extend the Boolean operations on Boolean variables to Boolean vectors:

Definition 8. Let $X$ be a finite set with $n$ elements, and $S, T \subseteq X$. Let $s$ and $t$ be the corresponding Boolean vectors of $S$ and $T$, respectively. We define the following element-wise operations over Boolean vectors, for $i = 1, 2, \cdots, n$,

$$[s + t_i]_i \overset{\text{def}}{=} s_i + t_i, \quad [s * t_i]_i \overset{\text{def}}{=} s_i * t_i,$$

$$[\neg s_i]_i \overset{\text{def}}{=} \neg s_i, \quad [s \leq t_i]_i \overset{\text{def}}{=} s_i \leq t_i,$$

$$[s \oplus t_i]_i \overset{\text{def}}{=} s_i \oplus t_i, \quad [s \equiv t_i]_i \overset{\text{def}}{=} s_i \equiv t_i$$

Note that each of the above operations over Boolean vectors actually realizes a set operation. The operations $+$, $*$ and $\neg$ over Boolean vectors realize the set union, intersection and complement operations. It has been shown that every finite Boolean algebra is isomorphic to some power set algebra, thus, set theory and Boolean algebra are essentially the same (see [7]). Table 1 illustrates some properties and axioms as applied to set theory, in which $S \oplus T$ means the symmetric difference of the sets $S$ and $T$, i.e., the set of elements which are in either of the sets and not in their intersection. Since there is one-to-one correspondence between the subsets of $X$ and the Boolean vectors in $\mathcal{B}^{\text{vec}}$, in this paper, we may mix a subset and a Boolean vector whenever it is convenient.
Example 2 (Continued). Now, we can calculate \( R^+(s) = A \odot s = [0, 1, 0, 1]^T \), corresponding to the set \( \{x_2, x_3\} \), which is exactly the set \( R^+(S) \). Similarly, we can compute \( R^-(s) = [0, 0, 0, 1]^T \) (i.e., \( \{x_1\} \)). Further, we have \( N(s) = \neg R^-(s) = [1, 0, 1, 0]^T \) (i.e., \( \{x_1, x_3, x_5\} \]), \( I(s) = \neg R^+(s) = [1, 1, 1, 0]^T \) (i.e., \( \{x_1, x_2, x_3, x_5\} \)), and \( F(s) = [1, 0, 1, 0]^T \) (i.e., \( \{x_1, x_3\} \)).

| Table 1 Set theory and Boolean algebra |
|--------------------------------------|--|--|--|--|
| Set theory                          | Boolean Algebra |
| \( \emptyset \)                     | 0               | \( S \cup T \) | \( s + t \) |
| \( X \)                             | 1               | \( S \subseteq T \) | \( s \leq t \) |
| \( \neg s \)                        | \( \neg s \)    | \( S \oplus T \) | \( s \oplus t \) |
| \( s \oplus t \)                    | \( S = T \)     | \( s \equiv t \) |

Next, let us introduce an operation on Boolean matrix and Boolean vector, which multiplies a Boolean matrix with a Boolean vector, and return a Boolean vector:

**Definition 9.** Let \( A \in \mathbb{B}^{m \times n} \) be a Boolean matrix, and \( s \in \mathbb{B}^{1 \times n} \) a Boolean vector. The multiplication of \( A \) and \( s \), denoted by \( A \odot s \), is a Boolean vector \( t \in \mathbb{B}^n \) defined by

\[
t_i = \sum_{j=1}^{n} A_{ij} \cdot s_j
\]

(3)

The multiplication of a Boolean matrix and a Boolean vector is completely analogous to the numerical matrix multiplication, except that we use the Boolean operations + and * instead of ordinary addition and multiplication, respectively. Similar to the numerical matrix multiplication, we can easily prove that the Boolean matrix multiplication also obeys distributive properties over matrix addition:

**Proposition 3.** Let \( A \in \mathbb{B}^{m \times n} \) be a Boolean matrix, and \( s, t \in \mathbb{B}^n \) two Boolean vectors. It holds that

\[
A \odot (s + t) = A \odot s + A \odot t
\]

In the following, we will present and prove a critical theorem, which provides a basis for characterizing Dung's acceptability semantics by means of Boolean algebra.

**Theorem 1.** Let \( \Delta = (X, \mathcal{R}) \) with \( X = \{x_1, x_2, \cdots, x_n\} \), its attack matrix be \( A \) and \( S \subseteq X \). Assume \( s \) be the Boolean vector representation of \( S \) (w.r.t. \( X \)). Then, the following holds: (i) \( R^+(s) = A \odot s \); (ii) \( R^-(s) = A^T \odot s \).

The proofs of this theorem can refer to our previous work [8, Thm. 1]. One similar result based on numerical matrix multiplication appeared almost simultaneously in the literature [9], but we will utilize it in different ways. Combining this result with Definition 4, we can now write the Boolean algebra representation of the neutrality and innocuousness functions as below:

\[
N(s) = \neg R^-(s) = \neg (A \odot s)
\]

(4)

\[
I(s) = \neg R^+(s) = \neg (A^T \odot s)
\]

(5)

By the relationship between the neutrality function and the characteristic function shown in Proposition 1, then we have

\[
F(s) = N(N(s)) = \neg (A \odot \neg (A \odot s))
\]

(6)

**Example 2** (Continued). Now, we can calculate \( R^+(s) = A \odot s = [0, 1, 0, 1]^T \), corresponding to the set \( \{x_2, x_3\} \), which is exactly the set \( R^+(S) \). Similarly, we can compute \( R^-(s) = [0, 0, 0, 1]^T \) (i.e., \( \{x_1\} \)). Further, we have \( N(s) = \neg R^-(s) = [1, 0, 1, 0]^T \) (i.e., \( \{x_1, x_3, x_5\} \)), \( I(s) = \neg R^+(s) = [1, 1, 1, 0]^T \) (i.e., \( \{x_1, x_2, x_3, x_5\} \)), and \( F(s) = [1, 0, 1, 0]^T \) (i.e., \( \{x_1, x_3\} \)).

4. Encoding Dung’s Acceptability Semantics

In this section, we will encode Dung's various acceptability semantics via Boolean algebra. Each semantics is encoded as a finite array of Boolean expressions based on a vector of Boolean variables. The goal is to find an assignment to the vector of all Boolean variables so that each of these Boolean expressions evaluates to true, i.e., be satisfied. This is a typical Boolean Constraint Satisfaction Problem (BCSP), in which each Boolean expression is seen as a Boolean constraint. If no such satisfying assignment exists, then these Boolean constraints has no solution.

Let \( \Delta = (X, \mathcal{R}) \) with \( X = \{x_1, x_2, \cdots, x_n\} \), its attack matrix be \( A \), and Boolean vector \( s \) be a subset of \( X \).

**Conflict-free Boolean constraints.** Boolean vector \( s \) is conflict-free iff any of the equivalent Boolean constraints below are satisfied:

\[
s \in \mathcal{R}^+(s) = 0 \]  \hspace{1cm} [CF1]

\[
s \leq N(s) \]  \hspace{1cm} [CF2]

\[
s \leq I(s) \]  \hspace{1cm} [CF3]

in which [CF1] follows from the definition of conflict-freeness, [CF2] and [CF3] follow from Proposition 1.

**Stable Boolean constraints.** Boolean vector \( s \) is a stable extension iff any of the equivalent Boolean constraints below are satisfied:

\[
s \equiv N(s) \]  \hspace{1cm} [ST1]

\[
s \oplus R^+(s) \]  \hspace{1cm} [ST2]

where [ST1] follows from the condition ST in Definition 5, and [ST2] can be inferred from [CF1] together with the definition of operation \( \equiv \).

**Admissible Boolean constraints.** Boolean vector \( s \) is admissible iff any of the equivalent Boolean constraints below are satisfied:

\[
\begin{cases}
\text{[CF?] } & \\
\text{[CF]? } & \\
\text{[CF]? } & \\
\text{[CF]? } & \\
\text{[CF]? } & \\
\text{[CF]? } & \\
\end{cases}
\]

\[
\begin{cases}
\text{[AD1]} & \\
\text{[AD2]} & \\
\text{[AD3]} & \\
\text{[AD4]} & \\
\text{[AD5]} & \\
\end{cases}
\]

where [CF?] can be one of any conflict-free Boolean constraints, [AD1] can be obtained from the condition AD in Definition 5, and [AD2], [AD3] and [AD4] follow from Proposition 2. [AD5] can follow from [AD3]:

\[
s \leq N(s) \odot F(s) \implies s \leq N(s) \odot N(N(s))
\]

\[
\implies s \leq (A \odot s) \odot \neg (A \odot N(s))
\]

\[
\implies s \leq (A \odot s) \odot A \odot N(s)
\]

\[
\implies s \leq (A \odot (s + N(s)))
\]
\[ \varepsilon \leq s \leq N(s + N(s)) \]

**Complete Boolean constraints.** Boolean vector \( s \) is complete iff any of the following equivalent Boolean constraints are satisfied:

\[
\begin{align*}
& [\text{CF?}] \\
& s \equiv \forall(s) \quad [\text{C01}] \\
& s \equiv \exists(s) \equiv \forall(s) \quad [\text{C02}] \\
& s \equiv \forall(s + N(s)) \quad [\text{C03}] \\
& s \equiv \exists(s + N(s)) \quad [\text{C04}] \\
& s \oplus \forall(s + N(s)) \quad [\text{C05}]
\end{align*}
\]

where \([\text{CF?}]\) is a conflict-free Boolean constraints, \([\text{C01}]\) can be obtained from the definition of the C0 semantics, \([\text{C02}]\) and \([\text{C03}]\) follow from Proposition 2, the proof of \([\text{C04}]\) can refer to the proof of \([\text{AD5}], \) and \([\text{C05}]\) follows from \([\text{C04}]\) combining with the definition of \( \equiv \).

Now, let us discuss the complexity of the proposed encodings in terms of the number of Boolean operations that are used in these encodings. If an encoding has large number of total Boolean operations, then we may say it has a high complexity. Table 2 lists the number of various Boolean operations of the proposed encodings, which are computed using the fact that a Boolean matrix-vector multiplication requires \( n^2 \) Boolean \(*\) operations and \( n(n - 1) \) Boolean + operations. From the Table 2, we can see that for the same semantics, its encodings may have different complexity. For example, the encoding \([\text{AD4}]\) uses significantly more total operations than other \( \text{AD} \) encodings, thus it has a high complexity, and the same goes for the encoding \([\text{C03}]\) in all \( \text{C0} \) encodings. Strictly speaking, this is a rough way to estimate the complexity of an encoding, as different operations may have different complexity. For instance, the \( \leq \) operation can be considered to be less complex than the \( \oplus \) operation, since the former has less basic operations than the latter does.

Next, let us show how to utilize these Boolean constraints to find the extensions of a given semantics.

**Example 3.** We continue to consider the AF \( \Delta \) in Fig. 1.

| Encodings | Basic operations | Relation operations |
|-----------|------------------|---------------------|
| CF        | \( n(n - 1) \)   | \( n^2 + n \)      |
| CF        | \( n(n - 1) \)   | \( n^2 \)          |
| CF        | \( n(n - 1) \)   | \( n^2 \)          |
| ST        | \( n(n - 1) \)   | \( n^2 \)          |
| ST        | \( n(n - 1) \)   | \( n^2 \)          |
| AD        | \( 2n(n - 1) \)  | \( 2n^2 \)         |
| AD        | \( 2n(n - 1) \)  | \( 2n^2 \)         |
| AD        | \( 2n(n - 1) \)  | \( 2n^2 \)         |
| AD        | \( 3n(n - 1) \)  | \( 3n^2 + 3n \)    |
| AD        | \( 3n(n - 1) \)  | \( 3n^2 + 3n \)    |
| CO        | \( 2n(n - 1) \)  | \( 2n^2 \)         |
| CO        | \( 2n(n - 1) \)  | \( 2n^2 \)         |
| CO        | \( 2n(n - 1) \)  | \( 2n^2 \)         |
| CO        | \( 2n(n - 1) \)  | \( 2n^2 \)         |

Here, we consider to use the stable Boolean constraint model \([\text{ST1}]\) to calculate the stable extensions of \( \Delta \). Assume Boolean vector \( s = [s_1, s_2, s_3, s_4, s_5]^T \), and \( s_i \) is an unknown Boolean variable, then by \([\text{ST1}]\), we have the following constraints:

\[
\begin{align*}
& s_1 \\
& s_2 \\
& s_3 \\
& s_4 \\
& s_5
\end{align*}
\]

Then, computing the stable extensions of \( \Delta \) is to find out all Boolean vectors \( s \), which can satisfy all above Boolean constraints. It can be verified that there merely exists one solution \( s_1 = 1, s_2 = 0, s_3 = 0, s_4 = 1 \) and \( s_5 = 0 \), satisfying these constraints. Hence, \( \Delta \) merely has one stable extension \( s = [1, 0, 0, 1, 0]^T \), i.e., \([x_1, x_4]\).

We have encoded different argumentation semantics into a finite set of Boolean constraints, and each semantics has several encodings. Now, we face two issues: (1) How to solve these Boolean constraints? (2) How the performance of these encodings and which of them should be used? We will address these issues in the following sections.

5. Solving Argumentation by CLPB

In this paper, we choose Constraint Logic Programming over Boolean variables (CLPB) systems\(^3\), the modern Constraint Programming solvers, as our tool to solve argumentation problems. SAT solvers may be another common tools to solve BCSPs, and there are some works dedicated in this area (see [11]–[13]). However, we select CLPB solvers instead of SAT solvers in at least the following critical aspects: (1) CLPB systems are more algebraically oriented than most SAT solvers, and thus they are more suitable for encoding and solving our Boolean algebra problems; (2) CLPB systems provide plentiful operations on Boolean variables, and have abilities to handle any Boolean expressions with little conversion. While most SAT solvers merely take input in CNF, i.e., a logical AND of logical ORs of literals; (3) CLPB systems provide more flexible interfaces, and they can support variable quantification, conditional answers and easy symbolic manipulation of formulas.

5.1 CLPB Systems

CLPB is a declarative formalism for reasoning about propositional formulas. It is an instance of the general CLP scheme that extends logic programming with reasoning over Boolean domains\([14]\). Many Prolog systems (e.g., SWI-Prolog\(^1\), CHIP, SICStus Prolog) are equipped with CLPB systems. The solver contains predicates for checking the consistency and entailment of a constraint w.r.t. previous constraints, and for computing particular solutions to the set of previous constraints. The underlying representation of Boolean functions is based on reduced and ordered Boolean Decision Diagrams (BDD)\([15]\). This representation is very

\[^3\text{http://www.swi-prolog.org/}\]
With these syntaxes, we can easily translate any proposed multiplication of Boolean matrix and vector (see the Eq. (3)) to the Boolean sum operation performance.

Table 3 Syntax of Boolean expressions available in CLPB

| CLPB operation | Description |
|----------------|-------------|
| 0              | false       |
| 1              | true        |
| variable       | an unknown Boolean value in \( \mathcal{B} \) |
| Expr           | logical NOT |
| Expr+Expr      | logical OR  |
| Expr*Expr      | logical AND |
| Expr=:=Expr    | equality (logical equivalence, =) |
| Expr\(\le\)=Expr | less or equal (logical implication, \(\le\)) |
| + (Expr)       | logical disequality (exclusive or, \(\not\equiv\)) |
| *(Expr)        | the disjunction of the list \(\text{Exprs}\) |
| *(Exprs)       | the conjunction of the list \(\text{Exprs}\) |

There are two major interface predicates in CLPB systems:

1. sat(Expr): This checks the consistency of the \(\text{Expr}\) w.r.t. the accumulated constraints, and, if the check succeeds, tells the constraint that the \(\text{Expr}\) be satisfied.

2. labeling(Vars): \(\text{Vars}\) is a vector of Boolean variables.

This predicate assigns truth values to \(\text{Vars}\) such that all constraints are satisfied. It can enumerate all solutions by backtracking, but creates choicepoints only if necessary. If no such satisfying assignment exists, it returns false.

5.2 Building CLPB Models by Prolog

In this subsection, we provide a few hints on how we translate a Boolean constraint model of a given semantics into a Prolog program. We do this by using SWI-Prolog, which provides a comprehensive free Prolog environment, and is equipped with a CLPB system (see [16] for a system description). It can be easy to extend our Prolog programs into CHIP and SICStus Prolog, since SWI-Prolog has a high compatibility of syntax with these two ProLog environments.

Figure 2 reports the Prolog codes for element-wise operations on Boolean vectors, and the Boolean matrix-vector multiplication. For each predicate, we provide a clear and concise description before its definition using comments (e.g., line 1, 4 and 7, etc.). First, the predicate name is printed, followed by the parameters in italics. The parameter name indicates the type of the parameters. Parameters are preceded by a mode indicator in \([+, -, ?]\), which means the parameter is an input, an output, and both input and output parameter, respectively. The predicates \(\text{bnot}\), \(\text{bmul}\), \(\text{bleq}\) and \(\text{beqv}\) correspond to the Boolean operations \(\neg, \times, \leq\) and \(\equiv\) on Boolean variables, respectively. Using these predicates and combining with the built-in predicate \(\text{maplist}\)^1, we can implement the element-wise operations on Boolean vectors. For example, we can use the code in line 16 to do the element-wise multiplication operation on two Boolean vectors \(\text{V1}\) and \(\text{V2}\), and store the result into the Boolean vector \(\text{V3}\). Predicate \(\text{bmul}\) (in line 15) is used to compute the sum of the element-wise products of two Boolean vectors. Predicate \(\text{mv mul}\) (in line 21 and 22) realizes the multiplication of Boolean matrix and vector based on the predicate \(\text{bmul}\) by tail recursion[^1]. The predicate \(\text{mv mul}\) can be also seen as the implementation of the Boolean function \(R^s(s)\) when its input parameter \(\text{Matrix}\) is the attack matrix.

Figure 3 shows the implementations of the neutrality function (line 2), the innocuousness function (line 7) and the characteristic function (line 13) based on Eq. (4), Eq. (5) and Eq. (6), respectively. Here, we also give the implementation of the Boolean function \(R^c(s)\) (see line 9) based on the predicate \(\text{mv mul}\) combining with the built-in predicate \(\text{transpose}\). For all these predicates, the input parameter \(\text{AttM}\) is an attack matrix, input parameter \(\text{X}\) and output parameter \(\text{Y}\) are two vectors of Boolean expressions.

Now, using these previously defined predicates, we can easily translate the Boolean constraint encodings of Dung’s acceptability semantics into Prolog programs. Due to the limited space, we merely list the Prolog codes for the constraint models of the encoding [CF2], [AD3] and [C01] in Fig. 4. According to these implementation details, one can easily convert other encodings into logic programs. In each

\[^1\]maplist:Goal, ?List): True if Goal can successfully be applied on all elements of List. maplistGOal, ?List1, ?List2 and maplist:Goal, ?List1, ?List2, ?List3: As maplist/2, operating on pairs/triples of elements from two/three lists.

[^1]: The tail recursion is implemented by an optimization technique with constant stack space, like loops in other languages.
Fig. 3 Prolog codes for the neutrality, innocuousness, and characteristic functions.

\[
\begin{align*}
\text{neutrality}(\text{AttM}, X, Y) : & \% N^+(X) \\
\text{maplist}(\text{sort}, Y, Z). & \\
\end{align*}
\]

Fig. 4 Prolog codes for the constraint models of CF2, AD3, and CO1.

\[
\begin{align*}
\text{conflict_free2}(\text{AttM}, Y) : & \% \text{constraint model [CF2].} \\
\text{Y} & = N(Y) \\
\text{maplist}(\text{sort}, Y, Z). & \\
\text{sat}(Z). & \% \text{Post constraints} \\
\end{align*}
\]

5.3 Querying Solutions

5.3.1 Basic Querying

To find the solutions of a model, we require the help of the interface predicate labeling. For instance, the following query can be used to obtain a solution of the complete constraint model [CO1] (given the attack matrix \text{AttM}),

\[
\text{complete1}(\text{AttM}, X, \text{labeling}(X)).
\]

Then, the system responds with \(X = \langle\text{a vector of Boolean values}\rangle\), representing a complete extension, if it can satisfy all constraints for a certain X. On backtracking, we can enumerate all complete extensions one by one. If Prolog cannot find more assignments to satisfy all constraints, \text{false} will be returned. If we apply this query on the AF in Example 1, then three complete extensions are returned: \(X = [1, 0, 0, 0, 0], X = [1, 0, 1, 0, 0]\) and \(X = [1, 0, 0, 1, 0]\).

If one wants to access all extensions of a given semantics at one time, (s)he can utilize the built-in predicate \text{findall}\(^{1}\), e.g., the following query can find all complete extensions at one time, and bag the results into \text{Exts},

\[
\text{findall}(X, \langle\text{complete1}(\text{AttM}, X, \text{labeling}(X)), \text{Exts}\rangle).
\]

If this query is applied to the AF in Example 1, then it will return \text{Exts} = \([1, 0, 0, 0, 0], [1, 0, 1, 0, 0], [1, 0, 0, 1, 0]\).

5.3.2 Advanced Querying

Prolog systems also allow to previously bind Boolean values to some variables in Boolean vector \(X\) before querying, that is, the so-called \textit{conditional querying}. For instance, let us consider the following query for the AF in Example 1,

\[
\text{complete1}(\text{AttM}, X) :- \\
\text{conflict_free2}(\text{AttM}, X), \\
\text{neutrality}(\text{AttM}, X, Y), Y = N(Y) \\
\text{maplist}(\text{sort}, X, Y). \\
\text{sat}(Z). \text{\% Post constraints}
\]

The more variables in \(X\) are previously specified, the smaller the search space, and thus the less consuming time to find out all possible answers. Based on this point, in the following, we will introduce a novel querying strategy to speed up the solving process of the complete semantics and the stable semantics. Our strategy is based on the fact that the grounded extension is included in all complete extensions, and the arguments attacked by the grounded extension are not contained in any complete extensions, and it is true for stable extensions (if exists) (see Proposition 2(vi)). On the other hand, the grounded extension can be solved in a linear complexity. Therefore, we can construct the queries of complete extensions and stable extensions on the basis of the grounded extension. Our strategy is to previously assign 1s (respectively, 0s) to all variables whose corresponding arguments are in (respectively, attacked by) the grounded extensions, and to keep others unknown. For example, if we consider the AF \(\Delta\) in Example 1 again, the grounded extension of \(\Delta\) is \([x_1]\), and the argument(s) attacked by the grounded extension is \(R^+([x_1]) = \{x_2\}\). Hence, we bind 1 to \(X[1]\), and 0 to \(X[2]\), and then the query for finding all complete extensions can be written as

\[
X = [1, 0, \ldots, 1], \text{complete1}(\text{AttM}, X, \text{labeling}(X)).
\]

\(^{1}\text{http://swish.swi-prolog.org/p/argmat-clpb.pl.}\]
In this section, we report two empirical experiments in order to better understand how well various extensions can be calculated using our proposed encodings. The first empirical experiment is to test the performance of the proposed encodings for enumerating all possible extensions using basic querying. The second empirical experiment is to study the performance effects of using basic querying and advance querying to compute ST and CO semantics. We also compare our approach with a modern solver ArgSemSAT [12], which received the awards of “Second Place” of ICCMA2015 [17].

We consider to execute our tests by using the benchmark argument graphs from [18], where the authors randomly generate AFs according to two parameterized graph models: Kleinberg, and the Barabasi-Albert models. The Kleinberg model generates AFs with an $m \times k$ grid-like structure. They consider two different neighbourhoods, one connecting arguments vertically and horizontally, and the other randomly connecting the arguments diagonally. Table 4 lists the statistics of the Kleinberg graphs over five different test sets with 9, 16, 25, 36 and 49 nodes, and each having 100 cases. For each test set, the average number of edges and extensions of the four semantics are provided. We run the first experiment on Kleinberg graphs mainly based on the following considerations: First, each Kleinberg graph is a strongly connected graph, i.e., all arguments in the graph are attacked, thus for such graphs, their grounded extensions are the empty set, which means that no variable in $X$ can be specified in advance. Hence, the advance querying strategy has little effect on solving Kleinberg graphs. Second, the Kleinberg graphs have high number of extensions, increased massively with respect to the size of the graphs. Such graphs may be suitable for testing the performance of each encoding, as an efficient encoding can encode and organize a large number of solutions in a simple and efficient manner. The Barabasi-Albert model generates arbitrary AFs and inserts for any pair $(x, y)$ an attack from $x$ to $y$ with a given probability. The corresponding statistics are shown in Table 5 over five test sets with 40, 50, 60, 250 and 500 nodes, and with each test having 100 cases. For such AFs, they have small amount ST and CO extensions, and may have non-empty grounded extensions. Thus, the advance querying strategy can be applied, and thus our second experiment is based on this benchmark.

All experiments are executed on computers equipped with 2.4GHz Intel Xeon processors, with 64GB RAM, and the Linux distribution Ubuntu 14.04 (64bits). The timeout for each instance is set to 600 seconds. The average results of ArgSemSAT on the two benchmark graphs are given in Table 4 and Table 5, respectively. ArgSemSAT does not support CF and AD semantics, thus we merely focus on its tests using ST and CO semantics. The time is measured in seconds. The symbol “–” indicates that the whole family

### Table 4

| Nodes $|X|$ | Edges $|E|$ | Number of CF extensions | Number of ST extensions | Number of AD extensions | Number of CO extensions | ArgSemSAT ST | ArgSemSAT CO |
|-------|------|----------------|-------------------------|------------------------|------------------------|------------------------|-------------|-------------|
| 9     | 45   | 20.92          | 6.42                    | 13.47                  | 11.28                  | 0.028                  | 0.045       |
| 16    | 80   | 270.35         | 19.57                   | 112.67                 | 66.10                  | 0.087                  | 0.319       |
| 25    | 125  | 686.75         | 74.30                   | 1946.05                | 556.84                 | 0.355                  | 5.718       |
| 36    | 180  | 35523.70       | 574.29                  | 59450.03               | 11026.94               | 7.321                  | –           |
| 49    | 245  | $>10^7$        | 5573.79                 | 3306880.07             | 312670.90              | –                      | –           |

### Table 5

| Nodes $|X|$ | Edges $|E|$ | Number of ST extensions | Number of CO extensions | ArgSemSAT ST | ArgSemSAT CO |
|-------|------|----------------|------------------------|------------------------|-------------|-------------|
| 40    | 353  | 1              | 1                      | 0.010                  | 0.010       |
| 50    | 480  | 1              | 1                      | 0.012                  | 0.010       |
| 60    | 608  | 1              | 1                      | 0.056                  | 0.047       |
| 250   | 3431 | 1              | 1                      | 0.136                  | 0.133       |
| 500   | 7491 | 1              | 1                      | –                      | –           |

It can be seen that the computation for the grounded extension may play an important role in querying stable and complete extensions. Next, we will introduce an efficient approach to compute the grounded extension based on Boolean algebra, which may be faster than traditional set-based approach, and thus can further speed up the processes to enumerate stable and complete extensions based on the advance querying technique. In set-based approach, the grounded extension can be computed by iterating the characteristic function $F$ starting from the empty set. Thus, we can also compute the grounded extension by iterating the Boolean characteristic function, starting from $s^{(0)} = \emptyset$,

$$s^{(k)} = F(s^{(k-1)}) = \neg(A \odot \neg(A \odot s^{(k-1)})).$$

(7)

An effective way to employ this Boolean-algebra-based approach is to represent the Boolean vector by a bit vector, and the Boolean matrix (an array of Boolean vectors) by an array of bit vectors. A bit vector is an array data structure that can store Boolean variables compactly. It is effective at exploiting bit-level parallelism in hardware to perform element-wise Boolean operations quickly, which are directly supported by the processor. A typical bit vector stores $kw$ Boolean variables, where $w$ is the number of bits in the unit of storage, such as a (32-bit) word, or (64-bit) double word, and $k$ is some nonnegative integer. Hence, the calculation of an element-wise Boolean operation is reduced almost $w$ times, and thus it is true for the total amount of computation of the grounded semantics.

6. Experiments

http://www.dmi.unipg.it/conarg/
has reached the timeout.

The results of the first experiment are shown in Table 6, which reports the average running times for each test set and for each encoding of the CF, ST, AD and CO semantics. Each row in Table 6 shows the number of arguments of a test set and the average results over the test set. The results in bold with gray background indicate the winner for each encoding of each semantics. For each encoding, we can observe that the average run times exponentially increase in the number of nodes, and reach timeout when confronting large graphs (see the line $|X| = 49$, all of them are timeout expect for ST semantics). This may be caused by the fact that the number of the extensions of Kleiber graph grows exponentially.

Now, let us analyze the performance results of these encodings according to each semantics. For the CF semantics, we can see that the gaps among the three encodings, [CF1], [CF2] and [CF3], are slight on each test set (row comparison). One possible reason is that the three encodings have same number of total operations, i.e., $2n^2 + n$ (see Table 2). Nevertheless, the encoding [CF2] has three winners on 3 test sets, thus it can be considered as the winner encoding for CF semantics among the three encodings. For the ST semantics, it can be seen that the performance gap between [ST1] and [ST2] is also slight, and [ST2] is better than [ST1]. Note that both of [ST1] and [ST2] do not timeout on all test sets. This can be explained by the observation from Table 4 that the number of ST extensions is small compared to other semantics. For the AD and CO semantics, we set the [CF7] constraints of the encoding [AD1], [AD2] and [CO1] by [CF2], as it is the winner CF encoding. From Table 6, we can see that [AD2] is far more efficient than other AD encodings, and it is the winner encoding for AD semantics with winning on all test sets. The performance of [AD1], [AD3] and [AD5] are similar, and [AD5] is better than them. Note that [AD4] consumes the most time, and a timeout happens on the test set with 36 nodes, thus it has the worst performance. The reason may be that [AD4] uses the most number of Boolean operations than other AD encodings. For the CO semantics, the winner encoding is [CO5], and the worst is [CO3] (the reason is same as [AD4]). [CO2] and [CO4] have a closer performance, and both are better than [CO1]. From this experiment, we can see that for each semantics, although their encodings are equivalent, they have different performance. The reason might be that they use different numbers of Boolean operations (see Table 2). Some of them are quite inefficient compared to other encodings of the same semantics, such as the encoding [AD4] and [CO3] for AD and CO semantics, respectively. Some of them have similar performance, such as the encoding [CO4] and [CO5]. Therefore, if one intends to realize an efficient argumentation solver using these encodings, the one should notice these differences.

The performance results of the second experiment are depicted in Table 7, where the times are grouped by family of the basic (“*” column) and the advance querying techniques (“◦” column) according to each encoding. The results in bold and gray background indicate the winner between two querying techniques. Note that for the tests using advance querying, the times consist of two parts: the time to compute the grounded extension (as well as the arguments are attacked by the grounded extension), and the time to search answers. Here, the grounded extensions are computed by iterating the Boolean characteristic function using 64-bit double word as the storage unit for Boolean vectors. By this approach, the grounded extension can be solved significantly less than one second. From Table 7, we can easily see that the advance querying technique can significantly improve the problem solving process compared to the basic querying technique.

Note that, in Table 7, [CO1] with basic querying is significantly slower than [CO2], [CO4], and [CO5], while in Table 6, [CO1] is also slower than [CO2], [CO4], and [CO5], but their difference is subtle. This might be caused by the dif-

### Table 6 The average time (over 100 AFs and in seconds) for each proposed encoding using basic querying over Kleiber graphs with 9, 16, 25, 36 and 49 nodes.

| $|X|$ | CF | ST | AD | CO |
|-----|----|----|----|----|
|     | [CF1] | [CF2] | [CF3] | [ST1] | [ST2] | [AD1] | [AD2] | [AD3] | [AD4] | [AD5] | [CO1] | [CO2] | [CO3] | [CO4] | [CO5] |
| 9   | 0.107 | 0.106 | 0.104 | 0.114 | 0.111 | 0.154 | 0.145 | 0.240 | 0.443 | 0.155 | 0.174 | 0.160 | 0.465 | 0.162 | 0.153 |
| 16  | 0.217 | 0.211 | 0.213 | 0.231 | 0.231 | 0.232 | 0.472 | 0.404 | 0.434 | 0.381 | 0.438 | 0.636 | 0.587 | 4.554 | 0.583 | 0.566 |
| 25  | 1.506 | 1.504 | 1.524 | 1.172 | 1.122 | 3.649 | 2.216 | 3.160 | 45.06 | 2.939 | 6.003 | 5.614 | 99.82 | 5.670 | 5.655 |
| 36  | 44.68 | 44.25 | 44.64 | 10.00 | 9.890 | 99.13 | 34.05 | 89.58 | - | 86.47 | 165.4 | 154.2 | - | 153.8 | 153.7 |
| 49  | - | - | - | 128.5 | 125.6 | - | - | - | - | - | - | - | - | - |

### Table 7 The average time (over 100 AFs and in seconds) for finding all stable, complete and grounded extensions over Barabasi-Albert graphs with 40, 50, 60, 250 and 500 nodes. “*” means using basic querying, and “◦” indicates using advance querying.

| $|X|$ | ST | CO |
|-----|----|----|
|     | [ST1] | [ST2] | [CO1] | [CO2] | [CO3] | [CO4] | [CO5] | GR |
| 40  | 1.247 | 0.220 | 1.233 | 0.222 | 4.124 | 0.497 | 1.758 | 0.500 | 115.8 | 9.049 | 1.625 | 0.495 | 1.594 | 0.483 | <0.001 |
| 50  | 4.738 | 0.227 | 4.648 | 0.231 | 14.85 | 0.661 | 3.765 | 0.689 | - | 21.14 | 3.240 | 0.678 | 3.260 | 0.686 | <0.001 |
| 60  | 23.33 | 0.234 | 23.26 | 0.236 | 71.44 | 0.912 | 8.292 | 0.897 | - | 43.53 | 6.522 | 1.010 | 6.542 | 1.042 | <0.001 |
| 250 | - | 0.493 | - | 0.501 | - | 53.07 | - | 56.47 | - | - | 52.81 | - | 52.75 | - | <0.010 |
| 500 | - | 1.240 | - | 1.227 | - | 420.6 | - | 428.3 | - | - | 417.6 | - | 421.7 | - | <0.015 |
ferent structures of the Kleinberg model and the Barabasi-Albert model, where the former is an extension-dense model while the latter is just the opposite.

Next, let us compare our approach with ArgSemSAT on the two benchmark graphs. From the test results on Kleinberg graphs (see Table 4 and Table 6), we can see that ArgSemSAT is slightly superior to our approach on small graph sizes ($\mathcal{X} \leq 25$), while it encounters more timeouts on large graph sizes, e.g., the ST semantics on the instances with $\mathcal{X} = 49$, and the CO semantics on the instances with $\mathcal{X} \geq 36$. In our approach, however, two encodings of the ST semantics can pass all tests, and four CO encodings can pass the tests with $\mathcal{X} \leq 36$. According to the test results on Barabasi-Albert graphs shown in Table 5 and Table 7, we can see that ArgSemSAT is significantly faster than our approach on this benchmark regardless of whether or not using advance querying techniques. The two experiments show that our approach performs better on extension-dense graphs, but worse on extension-sparse graphs than ArgSemSAT does. Overall, our solver is less efficient than ArgSemSAT. There could be several reasons for this phenomenon: (1) The core logic part of our program is written in pure Prolog language. It is a very high-level language that is easy for modeling but it is less efficient than C/C++, a high speed language, which is used in ArgSemSAT; (2) We define a family of compound operations, i.e., the relation operations, in order to make the modeling process more simple and intuitive. However, this may sacrifice some efficiency, compared to the SAT-based approaches, whose input with CNF format merely contains basic Boolean operations. Despite these shortcomings, we believe that there is still room for improving our programs, e.g., resorting to other high speed constraint solvers.

7. Related Work and Conclusion

In this paper, a Boolean algebra approach has been proposed to encode and solve various acceptability semantics for abstract AFs. It provides a more intuitive way to model and characterize argument problems. The first work using matrix approach to solve argumentation is presented in [19]. It considers to use adjacency matrices and their sub-blocks to determine the extensions of Dung’s semantics according to some criteria based on the elementary permutation of matrices. First, our approach does not use sub-blocks and the matrix permutations operation, but use Boolean matrices and Boolean operations to characterize Dung’s semantics. Second, our approach is more intuitive than [19], as our approach is based on Boolean algebra, which has a natural connection with set theory, on which Dung’s semantics are founded. Recently, a Boolean matrix approach was proposed to formalize the basic concepts of AFs [8]. In this paper, we enrich their work to encode Dung’s acceptability semantics. Similar works also appeared in [9] and [20]. These works mainly focus on testing whether a given set of arguments is an extension of Dung’s semantics. However, our approach can be used not only to verify an argument set, but also to find out all possible extensions of a given semantics. Moreover, our approach provides each semantics with at least one encodings, while the works in [9] and [20] just make one encoding for each semantics, and thus our approach has a stronger representation capability than their works.

There are several works similar to our work by considering argumentation problems as a Boolean Constraint Satisfaction Problem (BCSP). The work in [21] expresses argumentation problems as a constraint problems, and solve constraints by Constraint Logic Programming (CLP). It is implemented as a modular meta-interpreter for its logic programs like Dung’s. Hence, the underlying principle is different from our Boolean-algebra-based approach. In [22], [23], the authors propose a mapping from AFs to BCSPs. For each semantics, they merely give one encoding started from Dung’s basic definitions (i.e., the encoding [CF1], [AD1], [ST1] and [CO01] for the four semantics, respectively). Our work provides more than one encoding for each semantics and includes their encodings. Moreover, we also give the performance tests for these encodings, which can be used for selecting the most efficient encoding for each semantics to speed up the searching process. Recently, several SAT-based tools are built to solve argumentation problems, e.g., ArgSemSAT [12] and CoQuiAAS [13]. They encode each semantics into a CNF formula and use SAT solver to enumerate all extensions. However, most modern SAT solvers merely take input in CNF format, and thus, there is usually little room and need for parametrization. Compared to our Boolean algebra approach, in which we encode each semantics as BSCP with a finite set of Boolean formula (constraints), our modeling phase is easier and more intuitive than these SAT-based approaches, as the conversion from a Boolean formula to CNF may lead to an polynomial growth of clauses, or introduce numerous auxiliary variables. Strictly speaking, the CNF formula may be not equivalent to its corresponding Boolean formula, since CNF has additional variables.

There are also some frameworks based on Logic Programming like languages. ASPARTIX [24] is the first work to compute Dung’s acceptability semantics based on Answer Set Programming (ASP). It maps arguments and attack relations into rules of logic program at an object level, and uses the answer-set solver DLV for computing the type of extension specified by the user. However, ASPARTIX is not a Boolean algebra approach.

As far as we know, all these implemented frameworks do not use Boolean-algebra-based characteristic functions to compute the grounded extensions, and do not use the grounded extensions to speed up the search of the stable and complete extensions. In this paper, we mainly concern on how to encode argumentation semantics by Boolean algebra, and on what the performance of these encodings is. Our Boolean algebra approach has been implemented based on CLPB using Prolog language. Compared to some modern solvers, it may be not quite efficient, but it is quite easy and intuitive to model and solve argumentation problems. It
can help us to discover and test new encodings in order to find higher efficient encodings for various argument semantics. We believe that some techniques, which are introduced in our approach, e.g., the advance querying, can be used to improve other modern solvers. There is still a large space to improve our approach. In future study, we mainly concern the following aspects: First, we intend to explore the underlying principles that why one encoding is faster than another, and to find out more efficient encodings based on these principles; Second, we plan to extend our Boolean algebra approach to characterize other semantics, that are not discussed in this paper, such as preferred, semi-stable and stage [2]; Third, we want to consider to use other high speed constraint-based solver, such as BDD, SAT and SMT, in order to achieve a high efficiency for argumentation problems.

Acknowledgements

We sincerely thank the reviewers for their helpful comments and suggestions. The research reported here was supported by the Fund NSFC61572279.

References

[1] P.M. Dung, “On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games,” J. AI, vol.77, no.2, pp.321–357, Sept. 1995.
[2] P. Baroni, M. Caminada, and M. Giacobini, “An introduction to argumentation semantics,” The Knowledge Engineering Review, vol.26, pp.365–410, 12 2011.
[3] G. Charwat, W. Dvorák, S.A. Gaggl, J.P. Wallner, and S. Woltran, “Methods for solving reasoning problems in abstract argumentation–a survey,” Artificial intelligence, vol.220, pp.28–63, 2015.
[4] D. Grossi and S. Modgil, “On the graded acceptability of arguments,” Proc. Twenty-Fourth International Joint Conference on Artificial Intelligence, pp.868–874, IJCAI, 2015.
[5] F. Besnard and A. Hunter, Elements of argumentation, MIT Press, Cambridge, 2008.
[6] J.E. Whitesitt, Boolean algebra and its applications, Courier Corporation, 1995.
[7] S. Givant and P. Halmos, Introduction to Boolean algebras, Springer Science & Business Media, 2008.
[8] F. Pu, J. Luo, and G. Luo, “Some supplementarys to the counting semantics for abstract argumentation,” IEEE 27th International Conference on Tools with Artif. Intelli., pp.242–249, 2015.
[9] E. Hadjisoteriou, “Computing Argumentation with Matrices,” Imperial College Computing Student Workshop, pp.29–36, 2015.
[10] P. Codognet and D. Diaz, “A simple and efficient boolean solver for constraint logic programming,” J. Automated Reasoning, vol.17, no.1, pp.97–128, 1996.
[11] J.P. Wallner, G. Weissenbacher, and S. Woltran, “Advanced SAT techniques for abstract argumentation,” Int’l Workshop on Computational Logic in Multi-Agent Systems, pp.138–154, Springer, 2013.
[12] F. Cerutti, M. Giacobini, and M. Vallati, “ArgSemSAT: Solving argumentation problems using SAT,” COMMA, pp.455–456, 2014.
[13] J.M. Lagrue, E. Lonca, and J.G. Maillé, “CoQuiAAS: A constraint-based quick abstract argumentation solver,” IEEE 27th International Conference on Tools with Artif. Intelli., pp.928–935, 2015.
[14] J. Jaffar and M.J. Maher, “Constraint logic programming: A survey,” The journal of logic programming, vol.19, pp.503–581, 1994.
[15] R. Drechsler and B. Becker, Binary decision diagrams: theory and implementation, Springer Science & Business Media, 2013.

Fuan Pu was born in 1986. His research interests lie in knowledge representation, intelligent information processing, computational argumentation theory, machine learning. He is currently a PHD candidate in School of Software, Tsinghua University. He is a student member of the IEEE and AAAI. Email: Pu.Fuan@gmail.com.

Guiming Luo is with the School of Software, Tsinghua University as a professor. He received his Ph.D degree in 1992 from Institute of Systems Science, Chinese Academy of Sciences and BE degree from University of Science and Technology of China in 1988. His research interests include system identification, signal processing, adaptive control and signal tracking, software reliability methods and model checking. Email: gluo@tsinghua.edu.cn.

Zhou Jiang was born in 1986. He is currently a PhD candidate at the Software School, Tsinghua University. Email: zjiangz12@mails.tsinghua.edu.cn.