DREDD - A Heuristics-guided Backtracking Solver with Information Propagation for Abstract Argumentation

Matthias Thimm
Institute for Web Science and Technologies,
University of Koblenz-Landau
thimm@uni-koblenz.de

Abstract

We give a short overview on the DREDD solver for abstract argumentation problems under grounded, complete, stable, and preferred semantics. The solver implements a DPLL-like approach to exhaustive search by iteratively trying out possible acceptability values for the arguments until a valid labelling is found or backtracking is needed. The search order is guided by domain-independent heuristics that aim at minimising backtracking steps and information propagation is used to infer acceptability values once certain decisions are made.

1 Introduction

An abstract argumentation framework $AF$ is a tuple $AF = (Arg, \rightarrow)$ where $Arg$ is a set of arguments and $\rightarrow$ is a relation $\rightarrow \subseteq \text{Arg} \times \text{Arg}$. For two arguments $A, B \in \text{Arg}$ the relation $A \rightarrow B$ means that argument $A$ attacks argument $B$.

Semantics are given to abstract argumentation frameworks by means of extensions [3] or labellings [2]. In this work, we use the latter. A labelling $L$ is a function $L : \text{Arg} \rightarrow \{\text{in}, \text{out}, \text{undec}\}$ that assigns to each argument $A \in \text{Arg}$ either the value $\text{in}$, meaning that the argument is accepted, $\text{out}$, meaning that the argument is not accepted, or $\text{undec}$, meaning that the status of the argument is undecided. Let $\text{in}(L) = \{A \mid L(A) = \text{in}\}$ and $\text{out}(L)$ resp. $\text{undec}(L)$ be defined analogously. A labelling $L$ is called conflict-free if for no $A, B \in \text{in}(L)$, $A \rightarrow B$.

Arguably, the most important property of a semantics is its admissibility. A labelling $L$ is called admissible if and only if for all arguments $A \in \text{Arg}$

1. if $L(A) = \text{out}$ then there is $B \in \text{Arg}$ with $L(B) = \text{in}$ and $B \rightarrow A$, and
2. if $L(A) = \text{in}$ then $L(B) = \text{out}$ for all $B \in \text{Arg}$ with $B \rightarrow A$,

and it is called complete (CO) if, additionally, it satisfies

3. if $L(A) = \text{undec}$ then there is no $B \in \text{Arg}$ with $B \rightarrow A$ and $L(B) = \text{in}$ and there is a $B' \in \text{Arg}$ with $B' \rightarrow A$ and $L(B') \neq \text{out}$.

The intuition behind admissibility is that an argument can only be accepted if there are no attackers that are accepted and if an argument is not accepted then there has to be some reasonable grounds. The idea behind the completeness property is that the status of an argument is only undec if it cannot be classified as in or out. Different types of classical semantics can be phrased by imposing further constraints. In particular, a complete labelling $L$

- is grounded (GR) if and only if $\text{in}(L)$ is minimal,
• is preferred (PR) if and only if \(\text{in}(L)\) is maximal, and
• is stable (ST) if and only if \(\text{undec}(L) = \emptyset\).

All statements on minimality/maximality are meant to be with respect to set inclusion. If \(L\) is a complete/grounded/preferred/stable labelling then \(\text{in}(L)\) is also called the corresponding complete/grounded/preferred/stable extension.

Given an abstract argumentation framework \(\text{AF} = (\text{Arg}, \rightarrow)\) and a semantics \(\sigma\) (either GR, CO, ST, or PR) we are interested in the following computational problems [8, 4]:

**SE-\(\sigma\):** Compute a single \(\sigma\)-extension of \(\text{AF}\).

**EE-\(\sigma\):** Enumerate all \(\sigma\)-extensions of \(\text{AF}\).

**DC-\(\sigma\):** For a given argument \(A\), decide whether \(A\) is in at least one \(\sigma\)-extension of \(\text{AF}\).

**DS-\(\sigma\):** For a given argument \(A\), decide whether \(A\) is in all \(\sigma\)-extensions of \(\text{AF}\).

The DREDD solver supports solving the above-mentioned four computational problems wrt. to **grounded, complete, stable, and preferred** semantics. It is a reimplementation of the solver HEUREKA [5] and written in the C programming language. It implements the DPLL-algorithm (Davis-Putnam-Logemann-Loveland) backtracking algorithm from SAT solving [1, Chapter 3] and uses domain-independent heuristics to guide the search order.

In the remainder of this system description, we give a brief overview on the architecture of DREDD (Section 2) and conclude in Section 3.

### 2 Solver architecture

The DPLL-algorithm (Davis-Putnam-Logemann-Loveland) is a standard exhaustive search procedure that is used—at least in its most general form—throughout all areas of general problem solving and search, e. g., most modern SAT solvers use this algorithm in one form or the other [1]. Moreover, there are also solvers for abstract argumentation problems that make use of this algorithm, see e. g. [5, 7]. Algorithm 1 depicts a very abstract version of this algorithm for the purpose of determining a single preferred labelling in abstract argumentation (problem SE-PR from before). As in [7], we use an extended definition of a labelling to allow for arguments to have an explicit label “unlabelled” (\(\text{unlab}\)) in order to iteratively define a proper labelling. In each step of the algorithm, an unlabelled argument is selected (line 4) and a label for this argument is determined (line 5). If an argument has not been labeled before the label \(\text{in}\) is chosen. If the algorithm had to backtrack once before to this step, the label \(\text{out}\) is chosen, otherwise the label \(\text{undec}\) is chosen (note that these details are omitted in the abstract depiction in Algorithm 1). Afterwards the labelling \(L\) is updated by setting the label of \(A\) to the chosen label (line 6). Then labels of other arguments are inferred, if possible (line 7). For example, if \(A\) has been labelled \(\text{in}\), all arguments attacked by \(A\) are labelled \(\text{out}\), see also [7]. During this propagation step, we check also for mislabeled arguments. For example, if \(A\) has been labelled \(\text{in}\) but already has an attacker labelled \(\text{undec}\), we have a contradiction and need to backtrack (indicated in lines 8 and 9). Backtracking is performed by unlabelling all arguments whose label had been inferred or explicitly set up-to the first argument where we still have to try another label. Once all arguments are labeled and we have not found in issue, we return the labelling as the solution (line 10). Note that, as we always try to label an argument \(\text{in}\) first, we obtain a labelling with subset-maximal \(\text{in}\)-labeled arguments, i. e. a preferred labelling.
Algorithm 1 Abstract backtracking algorithm for determining a preferred labelling

**Input:** \( \mathcal{AF} = (\mathcal{Arg}, \rightarrow) \) AAF

**Output:** \( L \) a preferred labelling

1. \( L \leftarrow \) all arguments unlab
2. Stack unlabeledArguments = Arg
3. while unlabeledArguments is not empty do
4. \( A \leftarrow \) unlabeledArguments.pop()
5. \( \text{lab} \leftarrow \) nextLabel(A)
6. \( L \leftarrow \) set \( A \) to \( \text{lab} \)
7. \( L \leftarrow \) propagate(A,lab)
8. if mislabeling detected then
9. backtrack
10. return \( L \)

Figure 1: The argumentation framework from Example 1

**Example 1.** Consider the abstract argumentation framework \( \mathcal{AF} = (\mathcal{Arg}, \rightarrow) \) depicted in Figure 1 and assume that arguments are processed in alphabetical order. A labelling \( L \) is initialised by setting the labels of arguments \( A, \ldots, F \) to unlab. In a first iteration argument \( A \) is selected and labelled in (in line 6 of Algorithm 1). This information is propagated through the framework (in line 7) resulting in arguments \( B \) and \( D \) receiving the label out (because they each have an attacker that is labelled in). Afterwards argument \( C \) receives the label in because its only attacker \( B \) has been labelled out. Now we encounter a mislabeling as \( C \) attacks the argument \( A \), which is labelled in. Due to this the algorithm backtracks (in line 9) to its last decision and labels arguments \( C, B, D, A \) again unlab. In the next iteration of the main loop of the algorithm, argument \( A \) is selected again but now labelled out (line 6). Information is propagated, resulting in \( B \) and \( D \) receiving label in (because their only attacker is labelled out), and argument \( C \) receiving the label out (because it has an attacker with label in). We found no mislabeling so the algorithm proceeds with the next iteration and selects \( E \) to be labelled in (in line 5). Information is propagated and argument \( F \) is labelled out accordingly. We found no mislabeling and—as all arguments are labelled—the algorithm terminates and returns the labelling \( L \) with \( L(B) = L(D) = L(E) = \text{in} \) and \( L(A) = L(C) = L(F) = \text{out} \). Observe that \( L \) is indeed a preferred labelling of \( \mathcal{AF} \).

Algorithm 1 is used in DREDD for other problems with small variations. For example, tasks pertaining to stable semantics do not consider the label undec during search. In order to decide DC-CO the argument under question is initially labeled in and the algorithm returns Yes if a labelling can be found. As a final example, for DC-ST the algorithm runs twice, first to check whether stable labellings exist at all and a second time to check whether there is a labelling.
where the argument under question is labelled in. Problems that can be solved in polynomial time (i.e., all tasks pertaining to grounded semantics, as well as SE-CO and DS-CO) are solved with a dedicated algorithm conceptually identical to the one described in [6].

In order to determine the search order—i.e., the order in which arguments are inserted into the stack in line 2 of Algorithm 1—DREDD makes use of several domain-independent heuristics, in the same way as its predecessor Heureka [5]. These heuristics are based on the topological structure of the argumentation graph and aim at processing those arguments first that are unlikely to require backtracking steps further along the run of the algorithm. For example, arguments with high degree are usually processed at an early stage as a mislabeling of those is (usually) quickly discovered.

3 Summary

We presented DREDD, a heuristics-guided backtracking solver for various problems in abstract argumentation. At its core, DREDD implements the DPLL-algorithm for general problem solving and makes use of domain-independent heuristics to determine the search order of arguments. The source code of DREDD is available at http://taas.tweetyproject.org while an executable Docker file can be downloaded from https://cloud.docker.com/u/matthiasthimm/repository/docker/matthiasthimm/taas-dredd.

References