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

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 $L : \mathsf{Arg} \rightarrow \{\mathtt{in}, \mathtt{out}, \mathtt{undec}\}$ that assigns to each argument $\mathcal{A} \in \mathsf{Arg}$ either the value $\mathtt{in}$, meaning that the argument is accepted, $\mathtt{out}$, meaning that the argument is not accepted, or $\mathtt{undec}$, meaning that the status of the argument is undecided. Let $\mathtt{in}(L) = \{\mathcal{A} \mid L(\mathcal{A}) = \mathtt{in}\}$ and $\mathtt{out}(L)$ resp. $\mathtt{undec}(L)$ be defined analogously. A labelling L is called *conflict-free* if for no $\mathcal{A}, \mathcal{B} \in \mathtt{in}(L)$, $\mathcal{A} \rightarrow \mathcal{B}$.

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

1. if $L(\mathcal{A}) = \mathtt{out}$ then there is $\mathcal{B} \in \mathsf{Arg}$ with $L(\mathcal{B}) = \mathtt{in}$ and $\mathcal{B} \rightarrow \mathcal{A}$, and

2. if $L(\mathcal{A}) = \mathtt{in}$ then $L(\mathcal{B}) = \mathtt{out}$ for all $\mathcal{B} \in \mathsf{Arg}$ with $\mathcal{B} \rightarrow \mathcal{A}$,

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

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

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 $\mathtt{undec}$ if it cannot be classified as $\mathtt{in}$ or $\mathtt{out}$. 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 $\mathtt{in}(L)$ is minimal,

- is *preferred* (PR) if and only if $\texttt{in}(L)$ is maximal, and
- is *stable* (ST) if and only if $\texttt{undec}(L) = \emptyset$.

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

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

SE-σ: Compute a single σ-extension of AF.

EE-σ: Enumerate all σ-extensions of AF.

DC-σ: For a given argument $\mathcal{A}$, decide whether $\mathcal{A}$ is in at least one σ-extension of AF.

DS-σ: For a given argument $\mathcal{A}$, decide whether $\mathcal{A}$ is in all σ-extensions of 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" (`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 `in` is chosen. If the algorithm had to backtrack once before to this step, the label `out` is chosen, otherwise the label `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 $\mathcal{A}$ to the chosen label (line 6). Then labels of other arguments are inferred, if possible (line 7). For example, if $\mathcal{A}$ has been labelled `in`, all arguments attacked by $\mathcal{A}$ are labelled `out`, see also [7]. During this propagation step, we check also for mislabeled arguments. For example, if $\mathcal{A}$ has been labelled `in` but already has an attacker labelled `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 `in` first, we obtain a labelling with subset-maximal `in`-labeled arguments, i. e. a preferred labelling.

Algorithm 1 Abstract backtracking algorithm for determining a preferred labelling

```
Input:   AF = (Arg, →)   AAF
Output:  L               a preferred labelling

L ← all arguments unlab
Stack unlabeledArguments = Arg
while unlabeledArguments is not empty do
    A ← unlabeledArguments.pop()
    lab ← nextLabel(A)
    L ← set A to lab
    L ← propagate(A,lab)
    if mislabeling detected then
        backtrack
return L
```

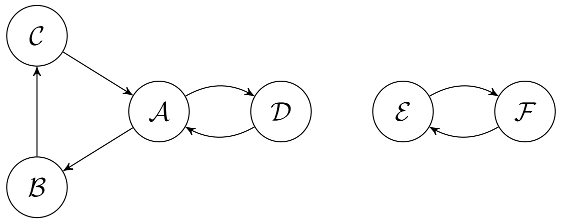

Figure 1: The argumentation framework from Example 1

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

[1] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[2] Martin W.A. Caminada and Dov M. Gabbay. A logical account of formal argumentation. *Studia Logica*, 93(2–3):109–145, 2009.

[3] Phan Minh Dung. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. *Artificial Intelligence*, 77(2):321–358, 1995.

[4] Wolfgang Dvořák and Paul E. Dunne. Computational problems in formal argumentation and their complexity. In Pietro Baroni, Dov Gabbay, Massimiliano Giacomin, and Leendert van der Torre, editors, *Handbook of Formal Argumentation*, chapter 14. College Publications, February 2018.

[5] Nils Geilen and Matthias Thimm. Heureka - a general heuristic backtracking solver for abstract argumentation. In *Proceedings of the 2017 International Workshop on Theory and Applications of Formal Argument (TAFA'17)*, August 2017.

[6] Sanjay Modgil and Martin Caminada. Proof theories and algorithms for abstract argumentation frameworks. In Guillermo Simari and Iyad Rahwan, editors, *Argumentation in Artificial Intelligence*, pages 105–129. Springer Verlag, Boston, MA, 2009.

[7] Samer Nofal, Katie Atkinson, and Paul E. Dunne. Looking-ahead in backtracking algorithms for abstract argumentation. *International Journal of Approximate Reasoning*, 78:265–282, 2016.

[8] Matthias Thimm and Serena Villata. The first international competition on computational models of argumentation: Results and analysis. *Artificial Intelligence*, 252:267–294, August 2017.