Splitting Argumentation Frameworks: An Empirical Evaluation

Ringo Baumann, Gerhard Brewka, Renata Wong

Universität Leipzig, Johannisgasse 26, 04103 Leipzig, Germany,
baumann@informatik.uni-leipzig.de

Abstract. In a recent paper Baumann [1] has shown that splitting results, similar to those known for logic programs under answer set semantics and default logic, can also be obtained for Dung argumentation frameworks (AFs). Under certain conditions a given AF A can be split into subparts A_1 and A_2 such that extensions of A can be computed by (1) computing an extension E_1 of A_1, (2) modifying A_2 based on E_1, and (3) combining E_1 and an extension E_2 of the modified variant of A_2. In this paper we perform a systematic empirical evaluation of the effects of splitting on the computation of extensions. Our study shows that the performance of algorithms may drastically improve when splitting is applied.

1 Introduction

Dung's abstract argumentation frameworks (AFs) [3] are widely used in formal approaches to argumentation. They provide several standard semantics, each capturing different intuitions about how to handle conflicts among (abstract) arguments. This makes them a highly useful tool in argumentation (see for instance Prakken's ASPIC [6] for a typical way of using AFs) and algorithms for computing extensions have received considerable interest.

In a recent paper, Baumann [1] has shown that splitting results, similar to those known for logic programs under answer set semantics [4] and default logic [8], can also be obtained for Dung argumentation frameworks. It turns out that under certain conditions a given AF A can be split into subparts A_1 and A_2 such that the computation of extensions of A can be divided into smaller subproblems: to compute an extension of A one has to (1) compute an extension E_1 of A_1, (2) modify A_2 based on E_1, and (3) combine E_1 and an extension E_2 of the modified variant of A_2.

Given these results, the obvious question is: does splitting an AF really pay off in practice? In this paper we aim to give an empirical answer to this question. We do this by systematically comparing the behavior of an algorithm for computing extensions with and without splitting. Our study shows that the performance of the algorithm indeed may drastically improve when splitting is applied.

Our evaluation is based on an implementation of Caminada's labelling algorithm [5], arguably the standard genuine algorithm for computing extensions.

We focus on preferred and stable semantics. We also include results for grounded semantics, but as this semantics is known to be polynomial an improvement of performance here was never expected, and our results confirm this.

The paper is organized as follows: we start in Sect. 2 with the necessary background on AFs, labellings, splitting and strongly connected components. We then describe in Sect. 3 the algorithms used in our evaluation. Sect. 4 contains the empirical evaluation and thus the main results of the paper. Sect. 5 concludes.

2 Background

2.1 Argumentation frameworks

An *argumentation framework* $\mathcal{A}$ is a pair (A, R), where A is a non-empty finite set whose elements are called *arguments* and $R \subseteq A \times A$ a binary relation, called the *attack relation*.

In the following we consider a fixed countable set $\mathcal{U}$ of arguments, called the *universe*. Quantified formulae refer to this universe and all denoted sets are finite subsets of $\mathcal{U}$ or $\mathcal{U} \times \mathcal{U}$ respectively. Furthermore we will use the following abbreviations. Let $\mathcal{A} = (A, R)$ be an AF, B and B' subsets of A and $a \in A$. Then

1. $(B, B') \mathrel{\bar{\in}} R \Leftrightarrow_{def} \exists b \exists b' : b \in B \wedge b' \in B' \wedge (b, b') \in R$,
2. a is defended by B in $\mathcal{A} \Leftrightarrow_{def} \forall a' : a' \in A \wedge (a', a) \in R \rightarrow (B, \{a'\}) \mathrel{\bar{\in}} R$,
3. B is conflict-free in $\mathcal{A} \Leftrightarrow_{def} (B, B) \mathrel{\bar{\not\in}} R$,
4. $cf(\mathcal{A}) = \{C \,| C \subseteq A, C$ conflict-free in $\mathcal{A}\}$.

The set of all extensions of $\mathcal{A}$ under semantics $\mathcal{S}$ is denoted by $\mathcal{E}_{\mathcal{S}}(A)$. We consider the classical semantics introduced by Dung, namely stable, preferred, complete and grounded (compare [3]).

Definition 1. *Let $\mathcal{A} = (A, R)$ be an AF and $E \subseteq A$. E is a*

1. *admissible extension*[1] *($E \in \mathcal{E}_{ad}(\mathcal{A})$) iff*
   $E \in cf(\mathcal{A})$ and each $a \in E$ is defended by E in $\mathcal{A}$,
2. *complete extension ($E \in \mathcal{E}_{co}(\mathcal{A})$) iff*
   $E \in \mathcal{E}_{ad}(\mathcal{A})$ and for each $a \in A$ defended by E in $\mathcal{A}$, $a \in E$ holds,
3. *stable extension ($E \in \mathcal{E}_{st}(\mathcal{A})$) iff*
   $E \in \mathcal{E}_{co}(\mathcal{A})$ and for every $a \in A \backslash E$, $(E, \{a\}) \mathrel{\bar{\in}} R$ holds,
4. *preferred extension (i.e. $E \in \mathcal{E}_{pr}(\mathcal{A})$) iff*
   $E \in \mathcal{E}_{co}(\mathcal{A})$ and for each $E' \in \mathcal{E}_{co}(\mathcal{A})$, $E \not\subset E'$ holds,
5. *grounded extension ($E \in \mathcal{E}_{gr}(\mathcal{A})$) iff*
   $E \in \mathcal{E}_{co}(\mathcal{A})$ and for each $E' \in \mathcal{E}_{co}(\mathcal{A})$, $E' \not\subset E$ holds.

[1] Note that it is more common to speak about admissible sets instead of the admissible semantics. For reasons of unified notation we used the less common version.

2.2 Labelling-based Semantics

The labelling approach [2, 5] provides an alternative possibility to describe extensions. Given an AF $\mathcal{A} = (A, R)$, a labelling is a total function $L : A \to \{in, out, undec\}$. We use $x(L)$ for $L^{-1}(\{x\})$, i.e. $x(L) = \{a \in A \mid L(a) = x\}$. This allows to rewrite a labelling L as a triple $(in(L), out(L), undec(L))$ which is frequently used. Analogously to $\mathcal{E}_\mathcal{S}(\mathcal{A})$ we write $\mathcal{L}_\mathcal{S}(\mathcal{A})$ for the set of all labellings prescribed by semantics $\mathcal{S}$ for an AF $\mathcal{A}$.

Definition 2. *Given an AF $\mathcal{A} = (A, R)$ and a labelling L of $\mathcal{A}$. L is called a complete labelling ($L \in \mathcal{L}_{co}(\mathcal{A})$) iff for any $a \in A$ the following holds:*

1. *If $a \in in(L)$, then for each $b \in A$ s.t. $(b, a) \in R$, $b \in out(L)$,*
2. *If $a \in out(L)$, then there is a $b \in A$ s.t. $(b, a) \in R$ and $b \in in(L)$,*
3. *If $a \in undec(L)$, then there is a $b \in A$ s.t. $(b, a) \in R$ and $b \in undec(L)$ and there is no $b \in A$ s.t. $(b, a) \in R$ and $b \in in(L)$.*

Now we are ready to define the remaining counterparts of the extension-based semantics in terms of complete labellings.

Definition 3. *Given an AF $\mathcal{A}$ = (A,R) and a labelling $L \in \mathcal{L}_{co}(\mathcal{A})$. L is a*

1. *stable labelling ($L \in \mathcal{L}_{st}(\mathcal{A})$) iff $undec(L) = \emptyset$,*
2. *preferred labelling ($L \in \mathcal{L}_{pr}(\mathcal{A})$) iff for each $L' \in \mathcal{L}_{co}(\mathcal{A})$, $in(L) \not\subset in(L')$,*
3. *grounded labelling ($L \in \mathcal{L}_{gr}(\mathcal{A})$) iff for each $L' \in \mathcal{L}_{co}(\mathcal{A})$, $in(L') \not\subset in(L)$.*

Theorem 1. *[5] Given an AF $\mathcal{A}$. For each $\sigma \in \{co, st, pr, gr\}$,*

1. *$E \in \mathcal{E}_\sigma(\mathcal{A})$ iff $\exists L \in \mathcal{L}_\sigma(\mathcal{A}) : in(L) = E$ and*
2. *$|\mathcal{E}_\sigma(\mathcal{A})| = |\mathcal{L}_\sigma(\mathcal{A})|$ holds.*

This theorem will be used to make the splitting results applicable for our algorithm.

2.3 Splitting Results

Baumann [1][2] showed that, under certain conditions, the computation of the extensions of an AF $\mathcal{A}$ can be considerably simplified: one splits the AF $\mathcal{A}$ into two subframeworks $\mathcal{A}_1$ and $\mathcal{A}_2$, computes an extension E_1 of $\mathcal{A}_1$, uses E_1 to reduce and modify $\mathcal{A}_2$, computes an extension E_2 of the modified and reduced version of $\mathcal{A}_2$ and then simply combines E_1 and E_2. We briefly recall the relevant definitions as they are crucial for the algorithms to be discussed later.

Definition 4. *Let $\mathcal{A}_1 = (A_1, R_1)$ and $\mathcal{A}_2 = (A_2, R_2)$ be AFs such that $A_1 \cap A_2 = \emptyset$. Let $R_3 \subseteq A_1 \times A_2$. We call the tuple $(\mathcal{A}_1, \mathcal{A}_2, R_3)$ a splitting of the argumentation framework $\mathcal{A} = (A_1 \cup A_2, R_1 \cup R_2 \cup R_3)$.*

[2] The full version is available at http://www.informatik.uni-leipzig.de/∼baumann/.

Definition 5. *Let $\mathcal{A} = (A, R)$ be an AF, A' a set disjoint from A, $S \subseteq A'$ and $L \subseteq A' \times A$. The (S, L)-reduct of $\mathcal{A}$, denoted $\mathcal{A}^{S,L}$ is the AF*

$$\mathcal{A}^{S,L} = (A^{S,L}, R^{S,L})$$

where

$$A^{S,L} = \{a \in A \mid (S, \{a\}) \not\subseteq L)\}$$

and

$$R^{S,L} = \{(a, b) \in R \mid a, b \in A^{S,L}\}.$$

Definition 6. *Let $\mathcal{A} = (A, R)$ be an AF, E an extension of $\mathcal{A}$. The set of arguments undefined with respect to E is*

$$U_E = \{a \in A \mid a \notin E, (E, \{a\}) \not\subseteq R\}.$$

It can be checked that in case of $\sigma \in \{co, st, pr, gr\}$, U_E equals $undec(L)$, where L is the unique σ - labelling s.t. $in(L) = E$ holds (compare Theorem 1).

Definition 7. *Let $\mathcal{A} = (A, R)$ be an AF, A' a set disjoint from A, $S \subseteq A'$ and $L \subseteq A' \times A$. The (S, L)-modification of $\mathcal{A}$, denoted $mod_{S,L}(\mathcal{A})$, is the AF*

$$mod_{S,L}(\mathcal{A}) = (A, R \cup \{(b, b) \mid a \in S, (a, b) \in L\}).$$

We now present the splitting theorem in both extension-based and labelling-based semantics style. The labelling-based notation can be easily obtained by using the original extension-based splitting results (1.(a), 2.(a)), Theorem 1 and the observation below Definition 9.

Theorem 2. *($\sigma \in \{st, pr, co, gr\}$) Let $\mathcal{A} = (A, R)$ be an AF which possesses a splitting $(\mathcal{A}_1, \mathcal{A}_2, R_3)$ with $\mathcal{A}_1 = (A_1, R_1)$ and $\mathcal{A}_2 = (A_2, R_2)$.*

1. *(a)* $E_1 \in \mathcal{E}_\sigma(\mathcal{A}_1) \wedge E_2 \in \mathcal{E}_\sigma(mod_{U_{E_1}, R_3}(\mathcal{A}_2^{E_1, R_3})) \Rightarrow E_1 \cup E_2 \in \mathcal{E}_\sigma(\mathcal{A})$
   (b) $L_1 \in \mathcal{L}_\sigma(\mathcal{A}_1) \wedge L_2 \in \mathcal{L}_\sigma(mod_{undec(L_1), R_3}(\mathcal{A}_2^{in(L_1), R_3}) \Rightarrow$
   $\exists!\ L \in \mathcal{L}_\sigma(\mathcal{A}) : in(L) = in(L_1) \cup in(L_2)$
2. *(a)* $E \in \mathcal{E}_\sigma(\mathcal{A}) \Rightarrow E \cap A_1 \in \mathcal{E}_\sigma(\mathcal{A}_1) \wedge E \cap A_2 \in \mathcal{E}_\sigma(mod_{U_{E \cap A_1}, R_3}(\mathcal{A}_2^{E \cap A_1, R_3}))$
   (b) $L \in \mathcal{L}_\sigma(\mathcal{A}) \Rightarrow \exists!\ L_1 \in \mathcal{L}_\sigma(\mathcal{A}_1) : in(L_1) = in(L) \cap A_1 \wedge$
   $\exists!\ L_2 \in \mathcal{L}_\sigma(mod_{undec(L) \cap A_1, R_3}(\mathcal{A}_2^{in(L) \cap A_1, R_3})) : in(L_2) = in(L) \cap A_2$

2.4 Splittings and Strongly Connected Components (SCC)

To generate a splitting we use the related graph-theoretic concept of *strongly connected components*. A directed graph is strongly connected if there is a path from each vertex to every other vertex. The SCCs of a graph $\mathcal{A}$ ($SCC(\mathcal{A})$ for short) are its maximal strongly connected subgraphs. Contracting every SCC to a single vertex leads to an acyclic graph. It is well-known that an acyclic graph induces a partial order on the set of vertices. Based on this order every SCC-decomposition can be easily transformed into a splitting. The most obvious

possibility is to take the union of the initial nodes of the decomposition ($= \mathcal{A}_1$) and the union of the remaining subgraph ($= \mathcal{A}_2$).
The following figure exemplifies the idea. We sketch three different splittings, namely S_1, S_2 and S_3. Note that these are not all possible splittings.

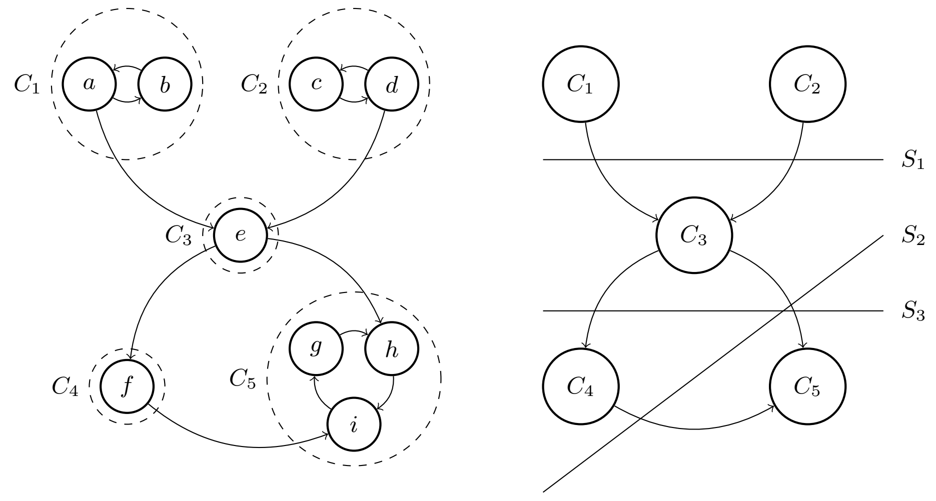

Fig. 1. SCCs and Splittings

3 Algorithms

Our implementation is based on the labelling algorithms for grounded, preferred and stable semantics in [5] and on the standard Tarjan algorithm for computing strongly connected components from [7]. For the latter we refer the reader to the original paper. We briefly describe the former to make the paper more self-contained.

3.1 Labelling Algorithms

The grounded labelling (L_{gr}) is generated as follows: all arguments which are not attacked are assigned the label *IN*. The next step is to assign the label OUT to all those arguments that are attacked by at least one of the arguments just labeled *IN*. We continue assigning the label *IN* to any argument having all of its attackers labeled OUT. The iteration stops when no further assignment can be made. The set $undec(L_{gr})$ is the set of arguments from A which were not labeled during the iteration.

In order to present the algorithms for preferred and stable labellings, further terminological explanations are in place.

Definition 8. *Given an AF $\mathcal{A} = (A, R)$ and a labelling $L \in \mathcal{L}_\sigma(\mathcal{A})$, an argument $a \in A$ is*

1. *legally IN iff x is labeled IN and $\forall b : (b, a) \in R$, b is labeled OUT,*
2. *legally OUT iff x is labeled OUT and $\exists b : (b, a) \in R$ and b is labeled IN,*
3. *illegally IN iff it is not legally IN,*
4. *illegally OUT iff it is not legally OUT,*
5. *super-illegally IN iff it is illegally IN and $\exists b : (b, a) \in R$ and b is legally IN or UNDEC.*

The algorithm for computing all preferred labellings (Algorithm 1) starts by assigning to all arguments the label *IN* (labelling L_{IN}), and initializing an empty set in which candidate labellings are to be stored. Then, by way of the main procedure *find_labellings* arguments that are *illegally IN* in L_{IN} are identified. To each of these arguments a procedure called *transition_step* is applied, by which the label of the given argument is changed from *IN* to *OUT*. If such an argument whose label has been changed from *IN* to *OUT* or if any argument(s) it attacks is *illegally OUT*, it will be relabeled as *UNDEC*. Thus we have obtained a new labelling which contains one less *IN*-argument. Then the entire process repeats again by passing any new labelling onto the main procedure, and the process continues until an acceptance or rejection condition is met. A labelling which does not have any argument which is *illegally IN* will be added to the candidate labellings, unless at any previous stage in the recursion it is detected that a better labelling has been found, i.e. a labelling with a larger *in*-set is already contained in *candidate_labellings*. If such a labelling with a larger cardinality of the *in*-set exists, the current labelling will not be processed.

In order to avoid the situation in which incomplete labellings are being generated by any incorrect assignment of labels, the algorithm is designed to always extract first those arguments that are *super-illegally IN*, i.e. arguments having at least one attacker *legally IN* or *UNDEC*, whenever we try to extract arguments that are *illegally IN*.

The algorithm for computing all stable labellings is obtained by rewriting line 1.5 of the algorithm for preferred labellings to read "**if** $undec(L) \neq \emptyset$ **then return**". If the set of arguments labeled *UNDEC* in a labelling is not empty, i.e. it violates the requirement for a stable labelling, the labelling will not be further processed.

3.2 Computation of Splitting

Our splitting algorithm consists of two parts: The first part (Algorithm 2) is executed prior to the first call of a labelling algorithm for a semantics and computes $\mathcal{A}_1$, $\mathcal{A}_2$ and the set R_3. The second part (Algorithm 3) is executed after receiving an extension from the labelling algorithm. The tuple $\mathcal{A}_2$ is then modified in accordance with the extension.

The first task is set to look for all the initial arguments (A_1) of our framework ($\mathcal{A}$). We use the set of *strongly connected components* returned by the Tarjan algorithm. The algorithm starts by introducing a Boolean variable *scc_attacked* which will be initialized to *false* for every SCC in $SCC(\mathcal{A})$. Given an SCC, once an argument in this SCC is attacked by some argument in another SCC, the

Algorithm 1: Computation of Preferred Labellings

```
input : $L_{IN} = (in(L_{IN}) = A, out(L_{IN}) = \emptyset, undec(L_{IN}) = \emptyset)$
1.1 $candidate\_labellings := \emptyset$
1.2 $find\_labellings(L_{IN})$
1.3 PROCEDURE $find\_labellings(L)$
1.4 begin
1.5     if $\exists L' \in candidate\_labellings : in(L) \subset in(L')$ then return;
1.6     if $L$ does not contain an argument illegally IN then
1.7         foreach $L' \in candidate\_labellings$ do
1.8             if $in(L') \subset in(L)$ then
1.9                 $candidate\_labellings := candidate\_labellings - \{L'\}$
1.10        $candidate\_labellings := candidate\_labellings \cup \{L\}$
1.11        return;
1.12    else
1.13        if $L$ has an argument that is super-illegally IN then
1.14            $x :=$ some argument that is super-illegally IN in $L$
1.15            $find\_labellings(transition\_step(L, x))$
1.16        else
1.17            foreach $x$ that is illegally IN in $L$ do
1.18                $find\_labellings(transition\_step(L, x))$
```

variable will be set to *true* and the execution of the algorithm for this SCC stops. Then the algorithm starts processing the next SCC. Only if *scc_attacked* remains *false*, which means that the corresponding SCC is not attacked, will all the arguments of this SCC be added to A_1. This way of splitting corresponds to S_1 in Fig. 1.

The splitting operation described above may result in subframeworks which differ a lot in size. We also provide a possibility to equalize the cardinalities along the partial ordering dictated by $SCC(\mathcal{A})$. The algorithm, called *optimize*, accepts the already computed arguments of A_1 and adds new ones under certain conditions. The first criterion used is the cardinality of A_1. Since the addition of new arguments relies on the partial order, it may not always be possible. Therefore, choosing 45% as a starting condition for equalization was an attempt to optimally equalize the numbers of arguments on the one hand, and on the other not to slow down the splitting process unnecessarily. Another condition limits the number of arguments added to A_1 by imposing a relative restriction on the added SCC's cardinality, i.e. if $|SCC| + |A_1| > |A| * 60\%$, the SCC will not be accepted. The algorithm runs recursively until no further arguments can be added (i.e. when $|optimal\_set| = |A_1|$). This way of splitting corresponds to S_3 in Fig. 1.

On the basis of the set A_1 we can then compute the sets R_1, A_2, R_2 as well as the set of attacks along which the framework is split (R_3). The pseudo code for these operations is not included here due to their obvious simplicity.

The processing of the tuple $\mathcal{A}_1$ by a labelling algorithm may return an extension as part of a labelling, if it exists. This extension (E_1) will in turn be used for modifying the AF $\mathcal{A}_2$ in the second part of the splitting algorithm. We start with the set A'_2 which is A_2 minus all the arguments in A_2 that are attacked by E_1, and we call it the modified set of A_2.

Next we apply the second algorithm on E_1, starting with an empty set, in order to compute a reduced set of undefined arguments (U_{E_1}). Note that we are not concerned with all the undefined arguments as stated in Definition 6, but only with those that are sources of an attack in R_3. Whenever an argument is a source of an attack in R_3, if it neither is an element of the extension E_1 nor is attacked by E_1, it will be added to the set U_{E_1}.

We then proceed to the final step in the modification of the AF $\mathcal{A}_2$. Given U_{E_1}, for every argument of A'_2 which is attacked by U_{E_1}, a loop is added. By this addition, we have modified the set R_2. We call this modified set R'_2, and now we can define $\mathcal{A}'_2$ as the tuple (A'_2, R'_2). With the given definition, $\mathcal{A}'_2$ is to be processed by a labelling algorithm.

4 Experimental Results

Our evaluation of the runtime for grounded, preferred and stable semantics is based on the sampling of 100 randomly generated frameworks. The tests were performed on a Samsung P510 notebook with a Pentium Dual Core Processor, CPU speed: 2.0 GHz, CPU Caches: 32 KB (L1) and 1024 KB (L2), RAM: 2 GB. We focused in our experiments on frameworks where the number of attacks (n) exceeds the number of arguments (m) by a factor between 1.5 and 3.

The reasons for this restriction are as follows. First of all, even leaving execution times aside[3], by further increasing the number of attacks the probability of generating frameworks consisting of a single SCC grows, thus rendering the experiment inconclusive as splitting has no effect on AFs with a single SCC. For example, initial tests showed that if 500 or more attacks (n) are given for 100 arguments (m), then almost all of the randomly generated frameworks will consist of only a single SCC and no effect of splitting is to be expected.

On the other hand, choosing an n smaller than m would not lead to significant differences in execution time between AFs with and without splitting as execution times tend to be fast under such conditions anyway.

With the above limitations in mind, a total of 100 examples were collected, with 20 examples extracted from each of the following m/n combinations: 10/30, 50/100, 100/175, 200/375 and 500/750. A brief description of the results obtained will be presented below together with a tabular summary of statistical

[3] For example, our preliminary testing showed that for AFs with 100 arguments, if 200 attacks are specified, the percentage of frameworks with runtime over 3 *min* for preferred semantics without splitting was about 70%.

Algorithm 2: Computation of Splitting, part 1

```
input  : set of strongly connected components ($SCC(\mathcal{A})$)
output: $A_1$, $R_1$, $A_2$, $R_2$, $R_3$

2.1  PROCEDURE $compute\_A_1(SCC(\mathcal{A}))$
2.2  begin
2.3      foreach SCC $\in SCC(\mathcal{A})$ do
2.4          $scc\_attacked := false$
2.5          loop:
2.6          foreach $a \in$ SCC do
2.7              foreach $b$ s.t. $(b, a) \in R$ do
2.8                  if $b \notin$ SCC then
2.9                      $scc\_attacked = true$
2.10                     break loop;
2.11         if $scc\_attacked = false$ then  add SCC to $A_1$
2.12     return $A_1$

2.13 PROCEDURE $optimize(SCC(\mathcal{A}), A_1)$
2.14 begin
2.15     $optimal\_set := A_1$
2.16     $illegal\_attacks := false$
2.17     foreach SCC $\in SCC(\mathcal{A})$ do
2.18         if $|A_1| < |A| * 0.45$ then
2.19             pick an $a \in$ SCC
2.20             if $a \notin A_1$ and $|A_1| + |\text{SCC}| < |A| * 0.6$ then
2.21                 $illegal\_attacks = false$
2.22                 loop:
2.23                 foreach $a \in$ SCC do
2.24                     foreach $(b, a) \in R$ do
2.25                         if $b \notin A_1$ and $b \notin$ SCC then
2.26                             $illegal\_attacks = true$
2.27                             break loop;
2.28                 if $illegal\_attacks = false$ then  add SCC to $A_1$
2.29     if $|A_1| < |A| * 0.45$ and $|optimal\_set| \neq |A_1|$ then
2.30         optimize($SCC(\mathcal{A}), A_1$)
2.31     return $A_1$
```

Algorithm 3: Computation of Splitting, part 2

```
     input  : an extension of $A_1$ ($E_1$), $A_2$, $R_1$, $R_2$, $R_3$
     output: $\mathcal{A}'_2 = (A'_2, R'_2)$
3.1  $compute\_modified\_A_2(E_1, A_2, R_3)$
3.2  $compute\_U_{E_1}(E_1, R_1, R_3)$
3.3  $compute\_modified\_R_2(U_{E_1}, R_2, R_3)$

3.4  PROCEDURE $compute\_modified\_A_2(E_1, A_2, R_3)$
3.5  begin
3.6      $A'_2 := A_2$
3.7      foreach $a \in E_1$ do
3.8          foreach $(a, b) \in R_3$ do
3.9              if $b \in A'_2$ then remove $b$ from $A'_2$
3.10     return $A'_2$

3.11 PROCEDURE $compute\_U_{E_1}(E_1, R_1, R_3)$
3.12 begin
3.13     $U_{E_1} := \emptyset$
3.14     foreach $(a, b) \in R_3$ do
3.15         if $a \notin E_1$ and $(E_1, \{a\}) \bar{\notin} R_1$ then add $a$ to $U_{E_1}$
3.16     return $U_{E_1}$

3.17 PROCEDURE $compute\_modified\_R_2(U_{E_1}, R_2, R_3)$
3.18 begin
3.19     $R'_2 := R_2 - \{(x, y) | (x, y) \in R_2$ and $(x \notin A'_2$ or $y \notin A'_2)\}$
3.20     foreach $a \in U_{E_1}$ do
3.21         foreach $(a, b) \in R_3$ do add $(b, b)$ to $R'_2$
3.22     return $R'_2$
```

data for each combination. Each table contains average-runtime results (in milliseconds) and gain-in-time results (in %)[4] for the grounded, preferred and stable semantics. Under "average runtime", the first column contains results from executing without splitting, the second from executing with non-optimized splitting and the third from executing with optimized splitting. Under "gain in time", minimal, maximal and average gain results, each in relation to non-optimized and optimized splitting, are distinguished.

The 10/30 combination was the only case in which we experienced no runtime that was over 3 *min*.[5] Thanks to the low number of arguments we were given a possibility of structural analysis. Although 20 examples is a small sample size, we were able to distinguish 4 characteristics based on the structure of the framework and the corresponding difference in runtime between executions without and with splitting. The analysis below applies to the preferred and stable semantics as the execution of the grounded semantics did not show any difference.

First, in 3 cases out of 20 a single SCC was generated. As splitting has no effect on AFs consisting of a single SCC, there was no runtime improvement for all 3 semantics. However, no noticeable runtime delay in relation to the splitting process was recorded either.

Second, 3 further examples had the form of a single argument SCC attacking a large SCC. Here we recorded no improvement or only a slight improvement in the runtime when splitting was applied: 0-20%.

Third, yet 3 further cases consisted of a single argument SCC with a self-loop attacking a large SCC. The only difference regarding the single argument between this form and the previous one was that we now had a loop attack. However in terms of runtime the gap was significant. In the second case it was between 68-71% for preferred semantics and between 99-100% for stable semantics.

And last, 11 of the random AFs had the form of a larger SCC attacking a single argument SCC, a single argument SCC with a self-loop or two SCCs; or the form of two SCCs, with at least one attack each, attacking a single SCC. The difference in execution without and with splitting ranged here between 80-99% for preferred semantics and between 59-100% for stable semantics.

The limited data suggest that splitting can render computation significantly faster for frameworks with certain characteristics. It seems that the most relevant are those AFs having one or more SCCs, each with at least one attack (i.e. a single argument SCC with a loop or an SCC with at least 2 arguments), attacking one or more SCCs whose structure in itself is not relevant.[6]

[4] For convenience, in the presented data we use "0 *ms*" to mean "close to 0 *ms*" and "100%" to mean "close to 100%".

[5] It comes as no surprise since the computation of preferred labellings for an AF with 10 arguments and 100 attacks takes around 260,000 *ms*

[6] An additional test on an AF of 10 arguments, of which 9 constituted an SCC with 81 attacks and all 9 attacked the 10th argument, recorded a 90% runtime difference for both preferred and stable semantics. This additional result lies nicely within the ranges of the previously obtained 80-99% and 59-100% respectively. A further test of a single argument with a self-loop attacking each argument of an SCC with 9 argu-

In general we obtained an average acceleration of 60% for both types of splitting in comparison to an execution without splitting. It is partly due to the fact that for the 10/30 combination both non-optimized splitting and optimized splitting usually overlap, which in turn is a result of the existence of large SCCs that limits the possibility of having different splittings. In no case was the execution with splitting slower than the one without.

Table 1. Evaluation results for 10 arguments and 30 attacks

m = 10	**average runtime (in ms)**			**gain in time (in %)**					
n = 30	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg.**	**avg./op**
grounded	1	1	1	0	0	0	0	0	0
pref.	3871	886	890	0	0	99	99	60	61
stable	1040	267	262	0	0	100	100	59	60

The runtimes for the 50/100 combination were very diversified: from 1 *ms* (for stable) and 2 *ms* (for preferred semantics) to 381,512 *ms* (preferred)[7] and 4,456 *ms* (stable). The grounded labelling was computed at the speed of 1-3 *ms* in each case, no improvement nor delay was recorded for executions with splitting in comparison to those without.

In 9 out of the 20 cases, the computation time for preferred and stable labellings without and with splitting was very short (below 20 *ms*). No significant difference was observed. The time gain for these cases was given as 0%, which had a negative effect on the average gain in time as shown in Table 2: it dropped to only 26-29%. Note that the maximal gain in time for both semantics was at 99%.

For the stable semantics we observed dramatic improvements in cases where no labellings existed. Through splitting of the framework, the time needed to find the first argument of the *undec* set, hence breaking the execution of the labelling algorithm, was at times very short. In 8 out of 15 cases where no labelling existed, the execution times lay below 20 *ms* which as mentioned above had 0% gain. Among the remaining 7 cases, 2 recorded an improvement of 99%, the rest between 17-75%. In none of the 20 examples was the execution without splitting faster than the one with splitting. Neither significant improvement nor delay was found for optimized splitting as compared to regular splitting.

ments and 81 attacks showed a 90% runtime difference for preferred semantics and 100% for stable semantics. The performance was evidently better than the previously obtained result for preferred semantics (68-71%). Having removed the loop attack we obtained a runtime of 1 *ms* for preferred and stable semantics, both with and without splitting. Again, these results are also in compliance with the ones obtained in the sample test using 20 examples.

[7] This example had already been included in the data before the imposition of the 3-minute limit, and so this is the only example with a runtime above 3 *mins*.

Table 2. Evaluation results for 50 arguments and 100 attacks

m = 50	**average runtime (in ms)**			**gain in time (in %)**					
n = 100	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg.**	**avg./op**
grounded	2	2	2	0	0	0	0	0	0
pref.	35860	23237	23352	0	0	99	99	29	26
stable	663	487	480	0	0	99	99	29	29

Some 40% of the frameworks generated with 100 arguments and 175 attacks had a computation time of at least 3 *min* for the preferred semantics without splitting. They were not taken into consideration for the reason stated at the beginning of this section. In the collected examples, the runtimes varied from around 20 *ms* to slightly below 40,000 *ms*. No stable labelling existed in 19 out of the 20 examples. In 9 out of these 19 examples, we obtained an improvement of 90-100% for the stable semantics and 0-50% for the remaining 10. No slow down due to the process of splitting was noticeable.

Here, for the first time, we recorded a significant improvement in runtime when the optimized version of splitting was applied. It was 13% for the preferred semantics and 5% for the stable semantics, both of which were better than the non-optimized variant. On average, an execution with splitting was better than one without splitting by 56-69% for the preferred semantics and by 60-65% for the stable semantics.

Table 3. Evaluation results for 100 arguments and 175 attacks

m = 100	**average runtime (in ms)**			**gain in time (in %)**					
n = 175	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg.**	**avg./op**
grounded	2	2	2	0	0	0	0	0	0
pref.	8335	3701	2502	0	0	93	99	56	69
stable	499	297	262	0	0	100	99	60	65

The computation time for preferred and stable labellings without splitting in frameworks of 200 arguments and 375 attacks was in general above 15 *ms*, thus making a more precise comparison possible. All the generated AFs showed a runtime improvement of at least 14% (pref.) and 26% (stable) when the execution with splitting is compared to the execution without splitting. Here too the gain in time reached in some cases 99% for the preferred labellings and 96% for the stable labellings.

With an average runtime of 3 *ms* for the grounded semantics, no difference between execution without and with splitting was found. The computation of stable labellings with applied splitting took on average 56% less time than that without. For the preferred semantics, the gain was somewhat less, it was 45% with optimized splitting and 47% with non-optimized splitting.

Table 4. Evaluation results for 200 arguments and 375 attacks

m = 200	**average runtime (in ms)**			**gain in time (in %)**					
n = 375	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg.**	**avg./op**
grounded	3	3	3	0	0	0	0	0	0
pref.	9333	6531	6296	14	16	99	98	47	45
stable	352	236	222	26	26	96	93	56	56

It was relatively comfortable testing the 500/750 combination since only about 20% of the randomly generated frameworks had a runtime above 3 *min* for preferred labellings without splitting. The execution time was quite steady. The lowest runtime for preferred semantics without splitting was 53 *ms* and 58 *ms* for stable semantics without splitting. The absence of drastic highs and lows was mirrored in all the average runtimes for preferred semantics, which were much lower than the average runtimes measured for 200/375. Here we observed also a steady improvement after splitting was applied. The lowest of which was 35% for preferred semantics and 33% for stable. The upper range was also less drastic with up to 86% for preferred and 97% for stable. The average differences were quite high with 57-61% for preferred labellings and 62-66% for stable. There was a drop in efficiency for the optimized type of splitting as compared to the non-optimized type (by 4% for both preferred and stable labellings). However, in AFs with a runtime above 700 *ms*, the optimized type ran faster than the one without optimization. In no case though was an execution with splitting slower than the one without splitting.

While in frameworks with 200 arguments and lower the grounded semantics did not perform worse after splitting, here we observed a visible slowdown. There was an average loss of 2% in the case of the non-optimized variant and an average loss of 36% in the case of the optimized variant.

Table 5. Evaluation results for 500 arguments and 750 attacks

m = 500	**average runtime (in ms)**			**gain in time (in %)**					
n = 750	**w/o spl.**	**w/ spl.**	**opt. spl.**	**min**	**min/op**	**max**	**max/op**	**avg.**	**avg./op**
grounded	10	10	13	-12	-60	15	-15	-2	-36
pref.	2785	1697	1168	36	35	86	78	61	57
stable	232	120	99	33	47	97	89	66	62

5 Conclusions

Based on our evaluations of 100 randomly generated AFs, we have made the following observations:

1. Among the 100 AFs, we observed an average improvement by 50-51% and by 54% for preferred and stable semantics respectively. The data contained some inconclusive examples which had "marred" the results to some extent.

2. No instance, neither for preferred semantics nor for stable, was found in which the execution with splitting lasted longer than the one without. This shows that the additional overhead introduced by splitting is negligible.
3. The optimized type of splitting did better than the non-optimized type in cases when the AF without splitting had a relatively long runtime. When the runtime was relatively short, the type without optimization usually performed better.
4. Splitting may significantly improve runtime for stable semantics in frameworks where no stable labellings exist. By splitting the framework, we were able to complete the execution of the algorithm a lot faster because it took less time to find a labelling with the *undec* set that was not empty.
5. It seems that there exist certain regularities between the structure of frameworks and the corresponding runtime. Having an SCC with at least one attack (or several SCCs with at least one attack each) attacking the rest of the framework can improve runtime significantly. We especially hope that this will greatly affect computation of large frameworks with large SCCs, which so far we were unable to test due to the required long computation time.

In future work we plan not only to extend our evaluation to larger AFs, we would also like to see whether there is an impact of moving from randomly generated to "natural" argumentation frameworks arising in realistic argumentation scenarios. Moreover, our results together with the theoretical considerations from the beginning of Sect. 4 suggest an advanced version of the algorithm where splitting is (1) performed iteratively on the identified subparts and (2) conditioned on the number of arguments and ratio between arguments and attacks, that is, only if the number of arguments is above a threshold and this ratio is in the "interesting" range splitting is performed.

References

1. Ringo Baumann. Splitting an argumentation framework. In *Proc. LPNMR-11, to appear*, 2011.
2. Martin Caminada. On the issue of reinstatement in argumentation. In *Proc. JELIA-06*, pages 111–123, 2006.
3. Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.
4. Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *ICLP*, pages 23–37, 1994.
5. Sanjay Modgil and Martin Caminada. Proof theories and algorithms for abstract argumentation frameworks. In Iyad Rahwan and Guillermo R. Simari, editors, *Argumentation in Artificial Intelligence*, pages 105–132. Springer, 2009.
6. Henry Prakken. An abstract framework for argumentation with structured arguments. *Argument and Computation*, 1:93–124, 2010.
7. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
8. Hudson Turner. Splitting a default theory. In *Proc. AAAI-96*, pages 645–651, 1996.