A Python Script for Abstract Dialectical Frameworks

Ringo BAUMANN [a], Maximilian HEINRICH [a]
[a] {baumann, mheinrich}@informatik.uni-leipzig.de

Abstract. We introduce a Python script for an easy and intuitive calculation of semantics of Abstract Dialectical Frameworks (ADFs) with arbitrary acceptance conditions. In addition, the script enables the evaluation of so-called *single-node formulae* with the help of Kleene's three-valued logic. The experimental results show that we achieve an enormous computational gain in this case.

Keywords. Abstract Dialectical Frameworks, Computation, Semantics

1. Introduction

Abstract Dialectical Frameworks (ADFs) are a powerful tool in the realm of Knowledge Representation. However calculating ADFs semantics without assistence is an effortful task which might complicate research or practical application. In order to facilitate the use of ADFs this paper presents a script for an easy calculation of various ADF semantics. The user just specifies the nodes with their acceptance conditions and the desired semantics. If the acceptance functions match the class of single-node formulae one may use Kleene's three-valued logic [1]. The experimental results show that we achieve an enormous computational gain in this case. We start with a short introduction on ADFs, followed by the presentation of the actual script. At the end we report on a test case generator and our experimental studies. The script, the test case generator as well as additional data can be found at `https://github.com/kmax-tech/ADF`.

2. A Brief Overview on ADFs

Abstract Dialectical Frameworks [2] are a generalization of classical Dung-style Argumentation Frameworks [3] providing more fine-grained modeling capabilities like collective attacks or single support. This expressiveness is achieved via adding acceptance functions (in terms of propositional formulae) to each argument.

Definition 1. *An* ADF *is a tuple* $D = (S, \Phi)$ *where* S *is a set of arguments (statements, nodes) and* $\Phi = \{\varphi_s \mid s \in S\}$ *is a set of propositional formulae.*

For a given ADF we consider three-valued interpretation $v : S \mapsto \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ as well as two-valued interpretation $v' : S \mapsto \{\mathbf{t}, \mathbf{f}\}$. The possible three truth values can be ordered via the so-called *information order* $\leq_i$. The value $\mathbf{u}$ (unknown) represents the $\leq_i$-least element and is thus the value with the lowest amount of information. The $\leq_i$-incomparable values $\mathbf{t}$ (true) and $\mathbf{f}$ (false) contain more information than $\mathbf{u}$, i.e. $\mathbf{u} \leq_i \mathbf{t}$ and $\mathbf{u} \leq_i \mathbf{f}$. We lift the information order to arbitrary interpretations v and w via $v \leq_i w$ if $v(s) \leq_i w(s)$ for each node $s \in S$. Essential for the understanding of ADFs is the so-called *gamma operator*. For a given three-valued interpretation v the operator considers any two-valued completion w of it and returns the consensus of these completions via the meet operator $\sqcap_i$. We have, $\mathbf{t} \sqcap_i \mathbf{f} = \mathbf{u}$ and $\mathbf{u} \sqcap_i \mathbf{t} = \mathbf{u} \sqcap_i \mathbf{f} = \mathbf{u}$. The interpretation $\mathtt{u}$ maps any node to $\mathbf{u}$ and thus represents the least information interpretation.

Definition 2. *Given an ADF $D = (S, \Phi)$. We define $\Gamma_D : \mathcal{V}_3^D \mapsto \mathcal{V}_3^D$ as*

$$\Gamma_D(v) : S \mapsto \{\mathbf{t}, \mathbf{f}, \mathbf{u}\} \text{ with } s \mapsto \sqcap_i \{w(\varphi_s) \mid w \in [v]_2^D\}.$$

Based on the gamma operator several semantics σ can be defined. The following well-known representatives, namely admissible, complete, preferred and grounded semantics specify their σ-interpretations as certain (pre)fixpoints of Γ_D.

Definition 3. *Given an ADF $D = (S, \Phi)$ and $v \in \mathcal{V}_3^D$.*

1. *$v \in adm(D)$ iff $v \leq_i \Gamma_D(v)$,*
2. *$v \in cmp(D)$ iff $v = \Gamma_D(v)$,*
3. *$v \in prf(D)$ iff v is $\leq_i$-maximal in $cmp(D)$ and*
4. *$v \in grd(D)$ iff v is $\leq_i$-least in $cmp(D)$.*

Example 1. *Consider the following situation. It is the middle of the night and one wants to get home (h) as quickly as possible. The only two possibilities to get there are either taking the bus (b) or walking (w). Unluckily, it is not known whether the bus will really run tonight. On the other hand it is important to get a little bit of sleep that night (s), which is only possible if the faster connection with the bus is used. In addition walking is only a preferable option if it is not raining (r), which is luckily the case. The scenario can be formally modeled in the following ADF $D = (\{b, h, r, s, w\}, \{\phi_b = \neg w \wedge b, \phi_h = w \vee b, \phi_r = \bot, \phi_s = h \wedge b, \phi_w = \neg r \wedge \neg b\})$.*

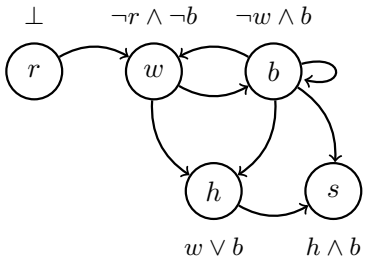


Figure 1. A Knowledge Base for "getting home"

3. A Python Script for ADFs

In order to use the script one has to edit the variables *nodes* and *chooseinterpretations*. The first variable *nodes* is a list which consists itself of list elements. Each of these elements has the form ["n", "φ_n"] where n represents a node and φ_n its corresponding acceptance condition. Both elements have to be of the type string. The acceptance condition has to be notated as a logical formula where logical conjunction is represented as ",", disjunction via ";" and negation as "#". It is further possible to use shorthands for true ("!") and false ("?"). Moreover, parentheses are allowed. The whole process is illustrated in Example 2. The variable *chooseinterpretations* is a list specifying the considered semantics. Possible inputs are "a" for admissible, "p" for preferred, "c" for complete or "g" for grounded semantics. Every input for this field needs to be of the type string. In addition it is possible to use the option "*tri*" enabling the evaluation of formulae w.r.t. Kleene's three-valued logic $\mathcal{K}_3$. This way of calculation is only correct if all used acceptance conditions are *single-node formulae*, i.e. any node appears at most once in an acceptance condition(cf. [4, Definition 7] for more detailed information).

Example 2. *The "getting home" scenario from Example 1 can be entered in the following way:*

```
1 nodes = [["b","#w,b"],["h","w;b"],["r","?"], \
2  ["s","h,b"],["w","#r,#b"]]
3 chooseinterpretations = ["a","c","p"]
```

The output is displayed in the following style:

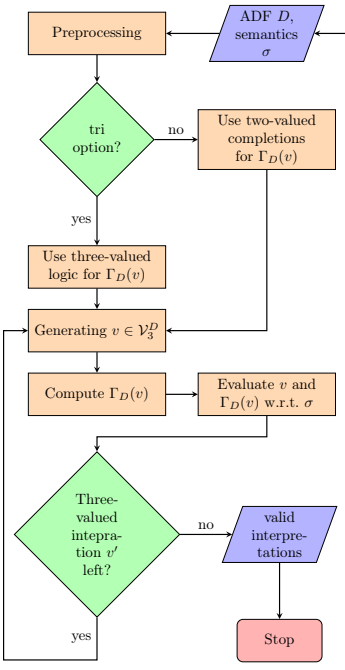
```
4 [['r', [' False '], {}], ['b', [' not ', 'w', ' and ',
5 'b', ''], {'b': [3], 'w': [1]}], ...
6 Admissible Interpretations
7 Nr.1 ['b:False', 'h:u', 'r:False', 's:False', 'w:u']
8 Nr.2 ['b:False', 'h:u', 'r:False', 's:False', 'w:True']
9 ...
```

On the technical side, the input from *nodes* (Example 2, lines 1–3) is first preprocessed into a customized structure respecting Python syntax (Example 2, lines 4–5). During the translation each element from *nodes* is transformed into a list. The first element represents the name of the node and the second one its corresponding acceptance condition. The additional third element is a lookup table storing the position of all nodes in the transformed acceptance condition. The aim of this table is to allow a fast replacing of nodes with truth values of a given interpretation during the evaluation step.

For a given ADF D the script generates systematically each three-valued interpretation v and calculates the corresponding $\Gamma_D(v)$. The way how $\Gamma_D(v)$ is computed depends on whether the *tri* option has been chosen. Anyway, in both cases we use the Python built-in *eval function* for the evaluation of acceptance conditions. If the *tri* option is specified the system calculates $\Gamma_D(v)$ via Kleene's three-valued logic. In this case the values $\mathbf{f}, \mathbf{u}, \mathbf{t}$ are replaced with $0, 0.5, 1$ and the semantics is given via $v(A \wedge B) = \min\{v(A), v(B)\}, v(A \vee B) = \max\{v(A), v(B)\}$

and $v(\neg A) = 1 - v(B)$. Without this option the script systematically generates all two-valued completions w of v. Moreover, the evaluation of any acceptance condition w.r.t. w is stored. We mark a node if a completion returns a different value than the stored ones. A marked node will be mapped to **u**. If all nodes are marked we reached `u` and the calculation is stopped. If not, all completions are computed and we return the values of the nodes according to its completions.

In the next step we compare v and $\Gamma_D(v)$. For admissible interpretations we have to check $v \leq_i \Gamma_D(v)$ and for complete ones $v = \Gamma_D(v)$ is required. For these two semantics the workflow and pseudocode are illustrated by the flowchart and the pseudocode presented in Figure 2. In order to calculate preferred interpretations we first compute all complete ones. After that each complete interpretation v is systematically taken and compared with all other complete interpretations $w \in cmp(D) \setminus \{v\}$. A preferred interpretation is found if $v \leq_i w$ is always false. The calculation of grounded semantics is considerably different. Since Γ_D is $\leq_i$-monotonic and the grounded interpretation is defined as $\leq_i$-least fixpoint we calculate it via iteratively applying the gamma operator starting from $\leq_i$-least interpretation `u` [2].



Require: ADF D and semantics σ
  Preprocessing of Input
  if *tri* option specified **then**
    Use three-valued logic for $\Gamma_D(v)$
  else
    Use two-valued completions for $\Gamma_D(v)$
  end if
  Generate three-valued interpretations $\mathcal{V}_3^D$
  for v in $\mathcal{V}_3^D$ **do**
    Compute $\Gamma_D(v)$
    Evaluate v and $\Gamma_D(v)$ w.r.t. σ
  end for
  return valid interpretations

Figure 2. Flowchart and Pseudocode for Computing Admissible and Complete Interpretations

4. Comparison: Classical Completion Approach vs. Three-valued Approach

In this section we evaluate how much faster ADF semantics can be calculated if the three-valued approach instead of the two-valued completion is used. We therefore wrote a testcase generator creating random instances of ADFs with single-node formulae. The generator works as follows: We fix a set S consisting of n nodes. Regarding the acceptance condition for a single node a we first determine

which nodes of S should occur in its acceptance formula. This happens through simulated coin flips. If a node has been chosen it gets negated with probability of 0.5. The selected nodes are conjoined with conjunction or disjunction, which is again determined through a coin flip. If no nodes are selected at all the acceptance formula is equally likely set to $\top$ or $\bot$. This procedure is repeated for every node in S. The calculation was done with a Ryzen 5 3600 CPU possessing 16 GB RAM. In its current version the script does not support multiple threads meaning that the whole computational process was not parallelized.

For any number of nodes between 1 and 10 we generated and evaluated 100 test instances w.r.t. all considered semantics. The results are displayed in Table 1. The column σ resp. σ-*tri* shows the average times (in seconds) for the classical two-valued completion approach or the three-valued logic approach, respectively. The results were rounded to four digits after the decimal point. The raw data can be found at the mentioned github page. The column speed factor shows the proportion from computation time of the two-valued completion to computation time of the three-valued approach. For the admissible and grounded interpretations the computational time is further illustrated in Figure 3. Please note that the y-axis uses logarithmic scale meaning that the growth in time is exponential.

As can be seen for both approaches the grounded semantics are calculated in very short time. For this semantics we start directly with $\mathtt{u}$ and apply Γ_D repeatedly till a fixpoint is reached. Therefore not every three-valued interpretation has to be generated and evaluated, which increases the speed of computation. In addition for grounded semantics the speed gain w.r.t. the three-valued approach is remarquable. This can be explained because $\mathtt{u}$ is the interpretation, which requires the most two-valued completions. The evaluation of these completions is skipped through the use of the three-valued approach. Admissible, preferred and complete semantics are influenced by the *tri*-option too but it is not comparable with the speed gain in case of grounded semantics. Moreover, the differences in calculation time between the three latter semantics are occurring on the scale of hundreds and thousands of seconds. Note that this observation is independent from the chosen approach. However, this is a bit surprising at least for preferred semantics as it uses the computation of complete semantics as a preprocess. Further experimental studies of this issue will be part of future work. It might be the case that a significant computation differences occurs far beyond the number ten of considered statements. To sum up, the three-valued approach increases the performance in an exponential way for any considered semantics. Nevertheless, except grounded semantics, computing ADFs with more than ten statements seems not to be feasible.

nodes	*adm*	*adm-tri*	speed	*cmp*	*cmp-tri*	speed	*prf*	*prf-tri*	speed	*grd*	*grd-tri*	speed
1	0.0	0.0	1.37	0.0	0.0	1.17	0.0	0.0	1.13	0.0	0.0	1.33
2	0.0002	0.0002	1.31	0.0002	0.0002	1.27	0.0002	0.0002	1.28	0.0001	0.0	1.88
3	0.0012	0.0008	1.51	0.0012	0.0008	1.52	0.0012	0.0008	1.51	0.0002	0.0001	2.79
4	0.0065	0.0036	1.82	0.0065	0.0036	1.81	0.0066	0.0036	1.82	0.0004	0.0001	4.3
5	0.0347	0.015	2.31	0.0347	0.0151	2.3	0.0348	0.0151	2.3	0.0008	0.0001	8.21
6	0.1765	0.0605	2.92	0.1771	0.0605	2.93	0.1749	0.0601	2.91	0.0016	0.0001	13.6
7	0.8953	0.2378	3.77	0.8961	0.2377	3.77	0.8954	0.2376	3.77	0.0039	0.0001	27.57
8	4.3859	0.8921	4.92	4.3854	0.8932	4.91	4.3863	0.8934	4.91	0.0094	0.0002	54.65
9	21.3739	3.3403	6.4	21.3224	3.326	6.41	21.3296	3.3227	6.42	0.0234	0.0002	112.17
10	103.1276	12.3506	8.35	103.0094	12.3051	8.37	103.3044	12.3584	8.36	0.0535	0.0002	215.73

Table 1. Performance Comparison: Classical vs. Three-valued Approach

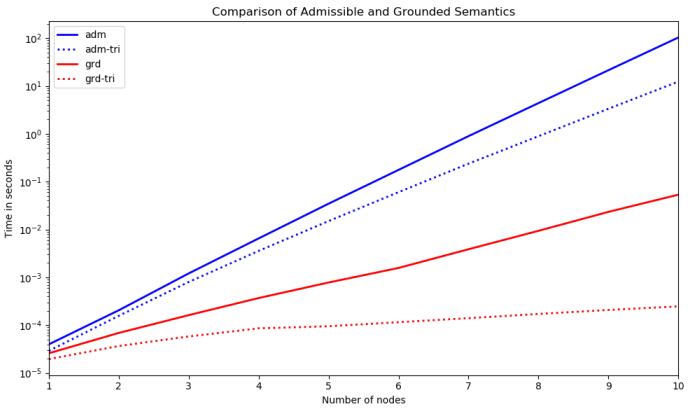


Figure 3. Performance Illustration via Logarithmic Scale

5. Future Work and Conclusion

In this paper we introduced a Python script which enables a convenient calculation of various ADF semantics. Python was choosen because it is a widely used high-level programming language, which enables an easy implementation and is therefore well-suited to test the benefits of a three-valued logic approach. Another system calculating ADFs semantics is DIAMOND [5] relying on ASP encodings. At the moment this project is no longer maintained and lacks the support of three-valued logic. For future work it seems interesting to test the three-valued approach against DIAMOND. Further improvements of our script might involve the implementation of a separate user interface avoiding the direct editing of the script. In addition, it is planned to integrate parallel processing for speeding up the computation. Finally, we plan to support the recently introduced *timed ADFs (tADFs)* via allowing additional temporal shorthands [4, Section 2.2] as acceptance conditions.

Acknowledgement

We thank DFG (406289255) and BMBF (01IS18026B) for funding this work.

References

[1] Wintein S. On All Strong Kleene Generalizations of Classical Logic. Studia Logica. 2016;104(3):503–545.
[2] Brewka G, Ellmauthaler S, Strass H, Wallner JP, Woltran S. Abstract Dialectical Frameworks. An Overview. Journal of Logics and their Applications. 2017 10;4(8):2263–2317.
[3] Dung PM. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. Artificial Intelligence. 1995;77(2):321–358.
[4] Baumann R, Heinrich M. Timed Abstract Dialectical Frameworks: A Simple Translation-Based Approach. In: Proceedings of COMMA; 2020. p. to appear.
[5] Ellmauthaler S, Strass H. The DIAMOND System for Computing with Abstract Dialectical Frameworks. In: Proceedings of COMMA; 2014. p. 233–240.