# pygarg: A Python Engine for Argumentation

Jean-Guy Mailly

Institut de Recherche en Informatique de Toulouse
CNRS - INP - UT3 - UT1 - UT2J

# pygarg: A Python Engine for Argumentation

Jean-Guy Mailly*
IRIT, Toulouse University, CNRS, INP, UT3, UT1C, Toulouse, France
jean-guy.mailly@irit.fr

* contact author

March 29, 2024

Technical report No. IRIT/RR–2024–02–FR
(version 1)

Institut de Recherche en Informatique de Toulouse
CNRS - INP - UT3 - UT1 - UT2J

# pygarg: A Python Engine for Argumentation

Jean-Guy Mailly*
IRIT, Toulouse University, CNRS, INP, UT3, UT1C, Toulouse, France
jean-guy.mailly@irit.fr

* contact author

March 29, 2024

**Abstract.** Recent advancements in algorithms for abstract argumentation make it possible now to solve reasoning problems even with argumentation frameworks of large size, as demonstrated by the results of the various editions of the International Competition on Computational Models of Argumentation (ICCMA). However, the solvers participating to the competition may be hard to use for non-expert programmers, especially if they need to incorporate these algorithms in their own code instead of simply using the command-line interface. In this paper we described `pygarg`, a Python implementation of the SAT-based approach used in the argumentation solver `CoQuiAAS`. Contrary to `CoQuiAAS` and most of the participants to the various editions of ICCMA, `pygarg` is made to be easy to use even for non-expert programmers. We show how to easily use `pygarg` in other Python scripts as a third-party library before experimentally demonstrating that it can be used in practice to solve large instances.

**Keywords:** Abstract Argumentation, SAT-based Solver, Python Software.

Technical report No. IRIT/RR–2024–02–FR
(version 1)

# Contents

# 1 Introduction

Abstract argumentation [12] provides a simple framework for representing conflicting pieces of information and deducing which of them can be accepted. In his seminal paper, Dung shows how it can be used to represent problems such as non-monotonic reasoning or stable marriage problems. Recent works show various other applications like fair allocation of resources [21] or explainability of (black box) classification models [1]. Despite the generally high complexity of argumentation reasoning [14], recent advancements in SAT-based solving techniques for argumentation [19] have permitted to handle harder instances and problems, as can be seen from the results of the International Competition on Computational Models of Argumentation (ICCMA) [16]. However, from a practical point of view, the solvers participating to the competition may not be easy to use for non-expert programmers. Indeed, the solvers are made to be used via their command-line interface, but it may be complicated to use them inside another piece of software. This may make it difficult for some part of the community to actually implement and test their ideas. This is why we propose `pygarg`, a Python implementation of the SAT-based algorithms initially proposed in `CoQuiAAS` [19],[1] the solver that won the first edition of ICCMA in 2015.[2] While `pygarg` can be used with a command-line interface inspired by the ICCMA requirements, it is also easy to incorporate it as a third-party library in any Python script. We chose Python because:

- it is quite simple to learn, and already widely adopted in some fields (*e.g.* machine learning),
- we can benefit from the `PySAT` library [17] to perform the calls to SAT oracles.

So, anyone in need of solving problems for abstract argumentation can use `pygarg` for problems such as deciding the (credulous or skeptical) acceptability of an argument, computing one or all the extensions, or counting the extensions, for Dung's semantics [12], the semi-stable [8] and ideal semantics [13].

In Section Section 2, we recall basic notions of abstract argumentation. Section Section 3 describes the main elements of the design of `pygarg`, and how to use it either as a command-line tool or in one's own Python script. We show in Section Section 4 that `pygarg` outperforms `PyArg`, the only other easy-to-use Python implementation of abstract argumentation that is available. Finally, we draw some conclusions in Section Section 5.

# 2 Background Notions

We start with a reminder of basic notions of abstract argumentation.

**Definition 1.** *An* abstract argumentation framework *(AF) [12] is a directed graph* $\mathcal{F} = \langle \mathcal{A}, \mathcal{R} \rangle$ *where* $\mathcal{A}$ *is the (finite) set of* arguments *and* $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$ *is the* attack relation*.*

Dung does not assume the finiteness of the set of arguments, however it is

[1] Notice that most of these SAT-based techniques are also incorporated in `Crustabri`, which won several tracks at ICCMA 2023. See https://github.com/crillab/crustabri.

[2] http://argumentationcompetition.org/2015/index.html

the case for our implementation. Notice that the set of argument can be empty.

We say that $a \in \mathcal{A}$ (respectively $S \subseteq \mathcal{A}$) *attacks* $b \in \mathcal{A}$ when $(a, b) \in \mathcal{R}$ (respectively some $a \in S$ attacks $b$). Then, a set of arguments $S$ *defends* an argument $a$ if $S$ attacks all the arguments attacking $a$.

Classical reasoning with AFs is based on the notion of *extensions*, *i.e.* sets of arguments collectivelly acceptable. Such an extension must usually satisfy two basic properties: $S \subseteq \mathcal{A}$ is *conflict-free* if $\forall a, b \in S$, $(a, b) \notin \mathcal{R}$; and $S \subseteq \mathcal{A}$ is *self-defending* if $S$ defends all its elements. We call a set satisfying both these properties *admissible*. We write $\mathbf{cf}(\mathcal{F})$ and $\mathbf{ad}(\mathcal{F})$ for the sets of conflict-free and admissible sets of an AF $\mathcal{F}$. Then, classical Dung's semantics [12] are defined as follows.

**Definition 2.** *Let $\mathcal{F} = \langle \mathcal{A}, \mathcal{R} \rangle$ be an AF. Then, the set $S \subseteq \mathcal{A}$ is:*

- *a* complete *extension if $S \in \mathbf{ad}(\mathcal{F})$ and $S$ contains all the arguments that it defends;*
- *a* preferred *extension if $S$ is a $\subseteq$-maximal admissible set;*
- *a* stable *extension if $S \in \mathbf{cf}(\mathcal{F})$ and $S$ attacks all the arguments in $\mathcal{A} \setminus S$;*
- *a* grounded *extension if $S$ is a $\subseteq$-minimal complete extension.*

We use $\mathbf{co}(\mathcal{F})$, $\mathbf{pr}(\mathcal{F})$, $\mathbf{st}(\mathcal{F})$ and $\mathbf{gr}(\mathcal{F})$ for these sets of extensions. It is well-known [12] that $|\mathbf{gr}(\mathcal{F})| = 1$ for any AF, that $\mathbf{st}(\mathcal{F}) \subseteq \mathbf{pr}(\mathcal{F})$, and preferred extensions also correspond to $\subseteq$-maximal complete extensions. Finally, from all the semantics studied in this paper, only the stable semantics may collapse, *i.e.* for any $\sigma \neq \mathbf{st}$, $\sigma(\mathcal{F}) \neq \emptyset$ for any AF. From the preferred semantics, one can define a "more skeptical" semantics as follows.

**Definition 3.** *Let $\mathcal{F} = \langle \mathcal{A}, \mathcal{R} \rangle$ be an AF. Then, the set $S \subseteq \mathcal{A}$ is an* ideal *extension [13] if it is a $\subseteq$-maximal admissible set among the sets of arguments included in the intersection of all the preferred extensions.*

We write $\mathbf{id}(\mathcal{F})$ the set of ideal extensions of an AF. Similarly to the grounded semantics, the ideal extension is unique for any AF.

Finally, we also focus on one last semantics, which is based on the notion of range. Given an AF $\mathcal{F} = \langle \mathcal{A}, \mathcal{R} \rangle$ and $a \in \mathcal{A}$, we write $a^+ = \{b \in \mathcal{A} \mid (a, b) \in \mathcal{R}\}$ the set of arguments attacked by $a$. We generalize it to sets, with $S^+ = \bigcup_{a \in S} a^+$ for the set of arguments attacked by $S$. The *range* of $S$ is $S^\oplus = S \cup S^+$, *i.e.* the set of arguments which are either members of $S$ or attacked by $S$.

**Definition 4.** *Let $\mathcal{F} = \langle \mathcal{A}, \mathcal{R} \rangle$ be an AF. Then, the set $S \subseteq \mathcal{A}$ is a* semi-stable *extension [8] if $S \in \mathbf{co}(\mathcal{F})$ and the range of $S$ is $\subseteq$-maximal among the ranges of all complete extensions of $\mathcal{F}$.*

We write $\mathbf{sst}(\mathcal{F})$ for the semi-stable extensions. Notice that all stable extensions are semi-stable extensions, and if $\mathbf{st}(\mathcal{F}) \neq \emptyset$ then both semantics coincide, but $\mathbf{sst}(\mathcal{F}) \neq \emptyset$ even when there is no stable extension.

**Example 1.** *Let $\mathcal{F} = \langle \mathcal{A}, \mathcal{R} \rangle$ be the AF shown in Figure 1. Its complete extensions are $\mathbf{co}(\mathcal{F}) = \{\{a_1, a_4, a_6\}, \{a_1, a_3\}, \{a_1\}\}$. Among them, the preferred extensions are $\mathbf{pr}(\mathcal{F}) = \{\{a_1, a_4, a_6\}, \{a_1, a_3\}\}$ (the $\subseteq$-maximal ones), and the grounded extension is $\mathbf{gr}(\mathcal{F}) = \{\{a_1\}\}$ (the $\subseteq$-minimal one). Among the preferred extensions, there is only one stable extension $\mathbf{st}(\mathcal{F}) = \{\{a_1, a_4, a_6\}\}$, which is also the unique semi-stable extension in this case. Finally, since the intersection of the preferred extensions is $\{a_1\}$, we deduce that the ideal extension is $\mathbf{id}(\mathcal{F}) = \{\{a_1\}\}$.*

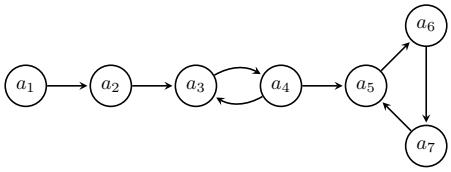


Figure 1: An example of AF $\mathcal{F}$

It is known that reasoning with these semantics is generally hard, with complexity up to the second level of the polynomial hierarchy, depending on the semantics and the exact decision problem [14]. However, various implementations based on SAT solvers have been proposed for reasoning with abstract argumentation, mainly based on the logical encoding by Besnard and Doutre [3]. Describing in details these algorithms is out of the scope of this paper, but we refer the interested reader to [19], since our work is essentially a Python implementation of the SAT-based approach from `CoQuiAAS`.

# 3 Describing pygarg

`pygarg`[3] is an open-source software,[4] implemented in Python, relying on `PySAT` [17] for performing calls to SAT solvers. In this section, we describe how `pygarg` can be used, either as a command-line tool, or as a library integrated to another Python code.

## 3.1 Using pygarg as a Command-line Tool

The source code of `pygarg` is based on five Python files:

- `__main__.py` (in the main package) provides the command-line interface of the software,
- in the `dung` package,
  - `apx_parser.py` provides tools for parsing an APX file [15] (as used notably in the first four editions of ICCMA in 2015, 2017, 2019 and 2021) into an argumentation framework usable by the solvers,
  - `dimacs_parser.py` provides tools for parsing a Dimacs file [18] (as used in the fifth edition of ICCMA in 2023) into an argumentation framework usable by the solvers,
  - `encoding.py` provides the tools used to translate argumentation problems into SAT solving,

[3] `pygarg` is available online: `https://github.com/jgmailly/pygarg`.
[4] Released under the GNU Lesser General Public License version 3.

- `solver.py` provides the functions for solving argumentation reasoning tasks.

Using the `__main__.py` file, one can use a command-line interface reminiscent of the ICCMA requirements, which is described in details in the `README` file of the software. In summary, one can use the command-line options:

- `-p PROBLEM` to define the problem to be solved, with `PROBLEM` being `XX-YY` where `XX` is one of `DC`, `DS`, `SE`, `EE`, `CE` (for credulous acceptability, skeptical acceptability, computing some extension, enumerating extensions and counting extensions) and `YY` must be one of `CF`, `AD`, `ST`, `CO`, `PR`, `GR`, `ID`, `SST` corresponding to $\sigma \in \{\mathbf{cf}, \mathbf{ad}, \mathbf{st}, \mathbf{co}, \mathbf{pr}, \mathbf{gr}, \mathbf{id}, \mathbf{sst}\}$,
- `-fo FORMAT` to define the format of the input file describing an AF, which must be equal to `apx` or `dimacs`,
- `-f FILENAME` to specify the path to the input file,
- `-a ARGNAME` to specify the name of the argument to be checked (for `DC` and `DS` problems).

The output of these commands follows the requirements of ICCMA 2023. This means that (when possible), an extension is provided as a witness for the (non-)acceptability of an argument, in a line starting with `w`. The same syntax is used for providing one (or each) extension of the AF. For instance, if `test.apx` corresponds to the AF from Figure 1, we obtain the result presented at Figure 2.

```
$ python3 main.py -p EE-CO -fo apx -f test.apx
w a1
w a1 a4 a6
w a1 a3
```

Figure 2: Enumerating extensions with `pygarg` on the command-line

In case there is an empty extension, a line with only `w` will be printed.

## 3.2 Using pygarg in Another Python Program

Now we focus on how to use `pygarg` in one's own Python code. The data structures used to represent an AF are simply a list of strings (representing the arguments names), and a list of lists of strings (representing the attacks). For instance, the AF from Figure 1 corresponds to the following structure:

```
args = ["a1", "a2", "a3", "a4", "a5", "a6", "a7"]
atts = [["a1", "a2"], ["a2", "a3"], ["a3", "a4"], ["a4", "a3"],
          ["a4", "a5"], ["a5", "a6"], ["a6", "a7"], ["a7", "a5"]]
```

Instead of manually constructing the list of arguments and attacks, one can use the `parse(filename)` function provided in both `apx_parser.py` and `dimacs_parser.py`. For instance,

```
import apx_parser

args, atts = apx_parser.parse("test.apx")
```

Then, one needs to focus on some functions provided in the file `solver.py`:

- `credulous_acceptability(args, atts, argname, sem)` determines whether the argument `argname` is credulously accepted under the semantics `sem`,
- `skeptical_acceptability(args, atts, argname, sem)` determines whether the argument `argname` is skeptically accepted under the semantics `sem`,
- `compute_some_extension(args, atts, sem)` computes one `sem`-extension,
- `extension_enumeration(args, atts, sem)` enumerates all the `sem`-extensions,
- `extension_counting(args, atts, sem)` counts the number of `sem`-extensions.

In these functions, the `sem` argument must be a string describing the semantics, using the same conventions as the command-line interface (for instance, the complete semantics is described by `"CO"`).

**Example 2.** *Continuing the previous example, Figure 3 shows how we enumerate the extensions of the AF from Figure 1, for the semantics* $\sigma \in \{\mathbf{co}, \mathbf{pr}, \mathbf{gr}, \mathbf{st}, \mathbf{sst}, \mathbf{id}\}$*. Running this script outputs:*

```
CO-extensions: [['a1'], ['a1', 'a4', 'a6'], ['a1', 'a3']]
PR-extensions: [['a1', 'a4', 'a6'], ['a1', 'a3']]
GR-extensions: [['a1']]
ST-extensions: [['a1', 'a4', 'a6']]
SST-extensions: [['a1', 'a4', 'a6']]
ID-extensions: [['a1']]
```

```
import solvers
import apx_parser

args, atts = apx_parser.parse("test.apx")

for sem in ["CO", "PR", "GR", "ST", "SST", "ID"]:
  print(f"{sem}-extensions: ", end='')
  print(solvers.extension_enumeration(args, atts, sem))
```

Figure 3: Enumerating extensions with `pygarg` imported in one's own code

# 4 Experimental Evaluation

In this section, we show that `pygarg` exhibits interesting runtimes in spite of its simple design. In particular, we compare it with `PyArg` [6, 22], the only implementation of argumentation reasoning (as far as we know) with a similar purpose of being easy to use instead of focusing on competition.

## 4.1 Experimental Setting

For these experiments, we consider instances generated following standard graph generation models, namely the Erdös-Rényi model (ER) which generates

graphs by randomly selecting attacks between arguments (with a given probability); the Barabási-Albert model (BA) which provides scale-free networks, *i.e.* a structure in which some nodes have a large number of connections, but in which nearly all nodes are connected to only a few other nodes; and the Watts-Strogatz model (WS) which produces small-world network, *i.e.* graphs with short average path lengths. These models are widely used in abstract argumentation studies, see *e.g.* [16]. The graphs have been generated thanks to the AFBenchGen2 generator [10]. In total, we have generated 9460 AFs almost evenly distributed between the three models (3000 AFs for the WS model and 3230 AFs for the ER and BA model). For each model, the number of arguments varies among $\texttt{Arg} = \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$.[5] The parameters used to generate graphs are as follows: for ER, 19 instances for each (nbArg, pAtt) in $\texttt{Arg} \times \{0.15, 0.2, \ldots, 0.95\}$; for BA, 17 instances for each (nbArg, pCyc) in $\texttt{Arg} \times \{0, 0.05, 0.1, \ldots, 0.9\}$; for WS, 5 instances for each (nbArg, pCyc, $\beta$, $\mathcal{K}$) in $\texttt{Arg} \times \{0.25, 0.5, 0.75\} \times \{0, 0.25, 0.5, 0.75, 1\} \times \{k \in 2\mathbb{N}\ s.t.\ 2 \leq k \leq \texttt{nbArg} - 1\}$. The meaning of the parameters is described in [10]. In the following, we collectively refer to the group of AFs generated using the Erdös-Rényi model (resp. Barabási-Albert model and Watts-Strogatz model) as rER (resp. rBA and rWS).

All the experiments were made on a machine using an Apple M1 chip, under macOS Sonoma 14.0, with 8GB of RAM. For each pair (AF, semantics), a timeout of 10 minutes was set. The instances used in the experiments are available in the GitHub repository of the software.

## 4.2 Results

Now let us describe the results of our evaluation. In the following, we present information about the runtime for solving the enumeration task on the semantics $\sigma \in \{\mathbf{co}, \mathbf{pr}, \mathbf{st}, \mathbf{gr}, \mathbf{sst}, \mathbf{id}\}$. Enumeration has been chosen to demonstrate the capabilities of pygarg since it provides a worst-case evaluation for all the other problems, which can be easily solved if one already knows the set of extensions (but they would actually be solved much faster in most cases with our implementation).

In Table 4, we describe our results for the complete semantics (other semantics are presented in Appendix A), where the left part corresponds to our implementation pygarg, and the right part to the existing tool PyArg. We present the average runtime and the number of timeouts (between parenthesis) depending on the number of arguments (in rows, $\{10, \ldots, 100\}$) in the AFs, for each group of instances (in columns, rER, rBA and rWS). Our goal is to push both pygarg and PyArg to their limits, and to exhibit the difference between both. For this reason, we made the choice to cut the experiments earlier for rBA instances, which are surprisingly harder to solve than other groups. More precisely, with the rBA group, we focus on AFs of size $\{10, \ldots, 50\}$ for pygarg and $\{10, 20, 30\}$ for PyArg. Also, for PyArg, we only go up to 60 arguments for rER and rWS. For larger instances, most runs reach the time limit without providing a solution.

The difference of performance between both approaches is obvious, since pygarg solves much more instances, and much faster than PyArg, for all the groups of instances and all sizes. We notice that enumerating the complete extensions, as presented here, is in practice harder on the instances rBAthan enumerating other kinds of extensions (see Appendix A). This is probably explained by the high number of complete extensions for this type of graphs. The clear difference between pygarg and PyArg is the same for all the other semantics except **gr**,

[5] However, as explained later, for some cases we have restricted the number of arguments to smaller values.

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.2 (0) | 0.2 (0) | 0.3 (0) |
| 20 | 0.6 (0) | 1.2 (0) | 0.6 (0) |
| 30 | 1.5 (0) | 27.4 (0) | 1.4 (0) |
| 40 | 3.1 (0) | 998.1 (0) | 2.9 (0) |
| 50 | 5.6 (0) | 8076.5 (12) | 5.1 (0) |
| 60 | 8.8 (0) | – | 8.3 (0) |
| 70 | 13.2 (0) | – | 12.3 (0) |
| 80 | 20.0 (0) | – | 18.3 (0) |
| 90 | 27.8 (0) | – | 25.5 (0) |
| 100 | 41.5 (0) | – | 35.3 (0) |

(a) pygarg

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 1.1 (0) | 4.9 (0) | 1.1 (0) |
| 20 | 37.6 (0) | 4197.3 (0) | 23.9 (0) |
| 30 | 687.3 (0) | 403537.6 (177) | 345.6 (0) |
| 40 | 12247.4 (0) | – | 4395.3 (0) |
| 50 | 23171.6 (21) | – | 28514.1 (0) |
| 60 | 34393.2 (38) | – | 31452.3 (45) |
| 70 | – | – | – |
| 80 | – | – | – |
| 90 | – | – | – |
| 100 | – | – | – |

(b) PyArg [6, 22]

Figure 4: Average runtime (in ms) depending on the number of arguments, for each set of instances in rER, rBA and rWS, for $\sigma = \mathbf{co}$. The numbers between parenthesis are the number of instances unsolved within the time limit.

where both approaches give similar results.

# 5 Discussion

As far as we know, only one implementation of argumentation reasoning tasks has been done with the same idea of being easy to use for non-expert programmers, namely PyArg [6, 22], that we used in the experimental evaluation. However, as seen previously, the focus of PyArg is not on efficient algorithms, but rather on its graphical interface[6] and various other advanced features like computing explanations of acceptability or structured argumentation, which are not included in the current version of pygarg. For this reason, the algorithms included in this platform are more "naive" (for instance they are not based on SAT solving techniques), and thus they do not scale up as well as SAT-based algorithms. This is not a major problem for the purpose of PyArg, which is visualisation (and one can assume that users interested in visualising graphs do not use large graphs with dozens or hundreds of arguments).

The other implementations that we know are those participating to ICCMA competitions which are optimized for efficient runtime performances, but may be harder to use for non-expert programmers (especially if one needs to use them in one's own code instead of simply relying on a command-line interface).

So, in our short experimental evaluation, we show that pygarg takes best of both worlds: an easy-to-use Python interface that even non-programmers can learn to use, and efficient SAT-based algorithms with good runtime performances.

For future work, we envision various possible directions. A first one would be to replace "naive" SAT-based algorithms by more efficient ones when possible (for instance, the current implementation of skeptical acceptability under the preferred semantics and reasoning with the ideal semantics is based on the enumeration of preferred extensions, but it could benefit from the techniques proposed by [24]). We are also interested in implementing algorithms for other semantics (like the stage semantics [25], or the more challenging semantics based on weak admissibility [2]). Still in line with recent ICCMA competitions, we would like to incorporate techniques for dynamic re-computation of extensions when an AF evolves [4], or approximation algorithms (in the spirit of [11]). Other problems related to extension-based extensions could be added,

[6] See https://pyarg.npai.science.uu.nl.

like counting the number of extensions (not) containing a given argument. The labelling-based [7] counterpart of the problems already implemented could also be added. Finally, more long term projects include the integration of gradual and ranking-based semantics [5], as well as more general abstract argumentation frameworks like Bipolar AFs [9], Strength-based AFs [23] or Incomplete AFs [20].

## References

[1] Leila Amgoud. Non-monotonic explanation functions. In *Proceedings of the 16th European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, ECSQARU 2021*, volume 12897 of *Lecture Notes in Computer Science*, pages 19–31. Springer, 2021.

[2] Ringo Baumann, Gerhard Brewka, and Markus Ulbricht. Shedding new light on the foundations of abstract argumentation: Modularization and weak admissibility. *Artif. Intell.*, 310:103742, 2022.

[3] Philippe Besnard and Sylvie Doutre. Checking the acceptability of a set of arguments. In *10th International Workshop on Non-Monotonic Reasoning (NMR 2004)*, pages 59–64, 2004.

[4] Stefano Bistarelli, Lars Kotthoff, Francesco Santini, and Carlo Taticchi. Containerisation and dynamic frameworks in iccma'19. In *Proceedings of the Second International Workshop on Systems and Algorithms for Formal Argumentation (SAFA 2018) co-located with the 7th International Conference on Computational Models of Argument (COMMA 2018)*, volume 2171 of *CEUR Workshop Proceedings*, pages 4–9. CEUR-WS.org, 2018.

[5] Elise Bonzon, Jérôme Delobelle, Sébastien Konieczny, and Nicolas Maudet. A comparative study of ranking-based semantics for abstract argumentation. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 914–920. AAAI Press, 2016.

[6] AnneMarie Borg and Daphne Odekerken. Pyarg for solving and explaining argumentation in python: Demonstration. In *Computational Models of Argument - Proceedings of COMMA 2022*, volume 353 of *Frontiers in Artificial Intelligence and Applications*, pages 349–350. IOS Press, 2022.

[7] Martin Caminada. On the issue of reinstatement in argumentation. In *Proceedings of the 10th European Conference on Logics in Artificial Intelligence, JELIA 2006*, volume 4160 of *Lecture Notes in Computer Science*, pages 111–123. Springer, 2006.

[8] Martin W. A. Caminada, Walter Alexandre Carnielli, and Paul E. Dunne. Semi-stable semantics. *J. Log. Comput.*, 22(5):1207–1254, 2012.

[9] Claudette Cayrol and Marie-Christine Lagasquie-Schiex. Bipolarity in argumentation graphs: Towards a better understanding. *Int. J. Approx. Reason.*, 54(7):876–899, 2013.

[10] Federico Cerutti, Massimiliano Giacomin, and Mauro Vallati. Generating structured argumentation frameworks: AFBenchGen2. In *Computational Models of Argument - Proceedings of COMMA 2016*, volume 287 of *Frontiers in Artificial Intelligence and Applications*, pages 467–468. IOS Press, 2016.

[11] Jérôme Delobelle, Jean-Guy Mailly, and Julien Rossit. Revisiting approximate reasoning based on grounded semantics. In *Seventeenth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU 2023)*, 2023.

[12] 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.

[13] Phan Minh Dung, Paolo Mancarella, and Francesca Toni. Computing ideal sceptical argumentation. *Artif. Intell.*, 171(10-15):642–674, 2007.

[14] Wolfgang Dvorák and Paul E. Dunne. Computational problems in formal argumentation and their complexity. In *Handbook of Formal Argumentation*, pages 631–688. College Publications, 2018.

[15] Uwe Egly, Sarah Alice Gaggl, and Stefan Woltran. ASPARTIX: implementing argumentation frameworks using answer-set programming. In *Proceedings of the 24th International Conference on Logic Programming, ICLP 2008*, volume 5366 of *Lecture Notes in Computer Science*, pages 734–738. Springer, 2008.

[16] Sarah Alice Gaggl, Thomas Linsbichler, Marco Maratea, and Stefan Woltran. Design and results of the second international competition on computational models of argumentation. *Artif. Intell.*, 279, 2020.

[17] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[18] Matti Järvisalo, Tuomo Lehtonen, and Andreas Niskanen. Design of ICCMA 2023, 5th international competition on computational models of argumentation: A preliminary report (invited paper). In *Proceedings of the First International Workshop on Argumentation and Applications (Arg&App 2023) co-located with 20th International Conference on Principles of Knowledge Representation and Reasoning (KR 2023)*, volume 3472 of *CEUR Workshop Proceedings*, pages 4–10. CEUR-WS.org, 2023.

[19] Jean-Marie Lagniez, Emmanuel Lonca, and Jean-Guy Mailly. Coquiaas: A constraint-based quick abstract argumentation solver. In *27th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2015*, pages 928–935. IEEE Computer Society, 2015.

[20] Jean-Guy Mailly. Yes, no, maybe, I don't know: Complexity and application of abstract argumentation with incomplete knowledge. *Argument Comput.*, 13(3):291–324, 2022.

[21] Jean-Guy Mailly. Abstract argumentation applied to fair resources allocation: A preliminary study. In *Proceedings of the First International Workshop on Argumentation and Applications (Arg&App 2023) co-located with 20th International Conference on Principles of Knowledge Representation and Reasoning (KR 2023)*, volume 3472 of *CEUR Workshop Proceedings*, pages 85–91. CEUR-WS.org, 2023.

[22] Daphne Odekerken, Annemarie Borg, and Matti Berthold. Accessible algorithms for applied argumentation. In *Proceedings of the First International Workshop on Argumentation and Applications (Arg&App 2023) co-located with 20th International Conference on Principles of Knowledge Representation and Reasoning (KR 2023)*, volume 3472 of *CEUR Workshop Proceedings*, pages 92–98. CEUR-WS.org, 2023.

[23] Julien Rossit, Jean-Guy Mailly, Yannis Dimopoulos, and Pavlos Moraitis. United we stand: Accruals in strength-based argumentation. *Argument Comput.*, 12(1):87–113, 2021.

[24] Matthias Thimm, Federico Cerutti, and Mauro Vallati. Skeptical reasoning with preferred semantics in abstract argumentation without computing preferred extensions. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021*, pages 2069–2075. ijcai.org, 2021.

[25] Bart Verheij. Two approaches to dialectical argumentation: admissible sets and argumentation stages. In *Proceedings of the Eighth Dutch Conference on Artificial Intelligence (NAIC'96)*, pages 357–368, 1996.

# A Additional Experimental Results

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.1 (0) | 0.1 (0) | 0.1 (0) |
| 20 | 0.3 (0) | 0.2 (0) | 0.3 (0) |
| 30 | 0.6 (0) | 0.5 (0) | 0.6 (0) |
| 40 | 1.3 (0) | 2.2 (0) | 1.2 (0) |
| 50 | 2.4 (0) | 18.1 (0) | 2.2 (0) |
| 60 | 3.9 (0) | – | 3.6 (0) |
| 70 | 5.7 (0) | – | 5.3 (0) |
| 80 | 8.8 (0) | – | 7.9 (0) |
| 90 | 12.3 (0) | – | 11.0 (0) |
| 100 | 18.0 (0) | – | 15.3 (0) |

(a) pygarg

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.8 (0) | 2.5 (0) | 0.8 (0) |
| 20 | 35.6 (0) | 1369.8 (0) | 23.0 (0) |
| 30 | 681.9 (0) | 350685.4 (56) | 345.6 (0) |
| 40 | 12337.3 | – | 14.897.8 (0) |
| 50 | 23362.9 (21) | – | 28757.6 (2) |
| 60 | 34108.7 (39) | – | 32084.8 (45) |
| 70 | – | – | – |
| 80 | – | – | – |
| 90 | – | – | – |
| 100 | – | – | – |

(b) PyArg [6, 22]

Figure 5: Average runtime (in ms) depending on the number of arguments, for each set of instances in rER, rBA and rWS, for $\sigma = \mathbf{st}$. The numbers between parenthesis are the number of instances unsolved within the time limit.

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.3 (0) | 0.4 (0) | 0.3 (0) |
| 20 | 0.8 (0) | 0.9 (0) | 0.8 (0) |
| 30 | 1.9 (0) | 2.7 (0) | 1.9 (0) |
| 40 | 3.5 (0) | 12.0 (0) | 3.4 (0) |
| 50 | 6.2 (0) | 79.5 (0) | 5.6 (0) |
| 60 | 9.5 (0) | – | 8.9 (0) |
| 70 | 14.0 (0) | – | 13.1 (0) |
| 80 | 21.1 (0) | – | 19.2 (0) |
| 90 | 28.9 (0) | – | 26.7 (0) |
| 100 | 42.2 (0) | – | 37.0 (0) |

(a) pygarg

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.9 (0) | 3.1 (0) | 0.9 (0) |
| 20 | 36.2 (0) | 4251.8 (0) | 23.1 (0) |
| 30 | 683.0 (0) | 379223.3 (85) | 344.5 (0) |
| 40 | 12284.9 (0) | – | 4060.6 (0) |
| 50 | 23251.3 (21) | – | 28652.3 (0) |
| 60 | 34199.1 (38) | – | 31497.1 (45) |
| 70 | – | – | – |
| 80 | – | – | – |
| 90 | – | – | – |
| 100 | – | – | – |

(b) PyArg [6, 22]

Figure 6: Average runtime (in ms) depending on the number of arguments, for each set of instances in rER, rBA and rWS for $\sigma = \mathbf{pr}$. The numbers between parenthesis are the number of instances unsolved within the time limit.

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.1 (0) | 0.1 (0) | 0.1 (0) |
| 20 | 0.7 (0) | 0.7 (0) | 0.6 (0) |
| 30 | 1.9 (0) | 2.5 (0) | 1.6 (0) |
| 40 | 3.5 (0) | 6.4 (0) | 3.4 (0) |
| 50 | 5.9 (0) | 13.8 (0) | 2.5 (0) |
| 60 | 4.9 (0) | – | 2.0 (0) |
| 70 | 5.1 (0) | – | 1.6 (0) |
| 80 | 5.8 (0) | – | 2.4 (0) |
| 90 | 5.4 (0) | – | 3.3 (0) |
| 100 | 5.0 (0) | – | 4.5 (0) |

(a) pygarg

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.1 (0) | 0.1 (0) | 0.1 (0) |
| 20 | 0.4 (0) | 0.2 (0) | 0.4 (0) |
| 30 | 1.2 (0) | 0.6 (0) | 1.1 (0) |
| 40 | 2.6 (0) | – | 2.5 (0) |
| 50 | 4.9 (0) | – | 4.5 (0) |
| 60 | 8.2 (0) | – | 7.6 (0) |
| 70 | – | – | – |
| 80 | – | – | – |
| 90 | – | – | – |
| 100 | – | – | – |

(b) PyArg [6, 22]

Figure 7: Average runtime (in ms) depending on the number of arguments, for each set of instances in rER, rBA and rWS for $\sigma = \mathbf{gr}$. The numbers between parenthesis are the number of instances unsolved within the time limit.

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.4 (0) | 0.3 (0) | 0.4 (0) |
| 20 | 1.0 (0) | 0.6 (0) | 1.0 (0) |
| 30 | 2.3 (0) | 1.3 (0) | 2.3 (0) |
| 40 | 4.3 (0) | 4.1 (0) | 4.1 (0) |
| 50 | 7.6 (0) | 24.3 (0) | 6.8 (0) |
| 60 | 11.5 (0) | – | 10.7 (0) |
| 70 | 16.9 (0) | – | 15.8 (0) |
| 80 | 25.0 (0) | – | 23.1 (0) |
| 90 | 35.0 (0) | – | 32.1 (0) |
| 100 | 51.6 (0) | – | 44.7 (0) |

(a) pygarg

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 1.4 (0) | 8.7 (0) | 2.0 (0) |
| 20 | 54.4 (0) | 10355.4 (2) | 33.7 (0) |
| 30 | 966.8 (0) | 676577.2 (305) | 447.8 (0) |
| 40 | 15413.8 (0) | – | 10165.8 (0) |
| 50 | 26728.1 (21) | – | 36941.1 (0) |
| 60 | 34108.7 (39) | – | 31513.9 (45) |
| 70 | – | – | – |
| 80 | – | – | – |
| 90 | – | – | – |
| 100 | – | – | – |

(b) PyArg [6, 22]

Figure 8: Average runtime (in ms) depending on the number of arguments, for each set of instances in rER, rBA and rWS for $\sigma = \mathbf{sst}$. The numbers between parenthesis are the number of instances unsolved within the time limit.

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.5 (0) | 0.6 (0) | 0.8 (0) |
| 20 | 1.6 (0) | 1.3 (0) | 1.6 (0) |
| 30 | 3.5 (0) | 3.4 (0) | 3.4 (0) |
| 40 | 6.7 (0) | 13.1 (0) | 6.3 (0) |
| 50 | 12.0 (0) | 81.3 (0) | 10.8 (0) |
| 60 | 18.3 (0) | – | 17.1 (0) |
| 70 | 27.0 (0) | – | 25.3 (0) |
| 80 | 39.8 (0) | – | 37.0 (0) |
| 90 | 54.6 (0) | – | 51.0 (0) |
| 100 | 78.4 (0) | – | 68.8 (0) |

(a) pygarg

| nbArg | rER | rBA | rWS |
|---|---|---|---|
| 10 | 0.9 (0) | 3.4 (0) | 0.9 (0) |
| 20 | 36.2 (0) | 5091.4 (0) | 23.165 (0) |
| 30 | 683.7 (0) | 480742.0 (234) | 344.4 (0) |
| 40 | 12299.6 (0) | – | 3782.1 (0) |
| 50 | 23287.5 (21) | – | 28593.5 (0) |
| 60 | 34260.7 (38) | – | 31372.3 (45) |
| 70 | – | – | – |
| 80 | – | – | – |
| 90 | – | – | – |
| 100 | – | – | – |

(b) PyArg [6, 22]

Figure 9: Average runtime (in ms) depending on the number of arguments, for each set of instances in rER, rBA and rWS for $\sigma = \mathbf{id}$. The numbers between parenthesis are the number of instances unsolved within the time limit.

Institut de Recherche en Informatique de Toulouse
CNRS - INP - UT3 - UT1 - UT2J

**ASR – Architecture, Systems and Networks**

RMESS – Networks, Mobile, Embedded, Wireless, Sattelites

SEPIA – Operating systems, distributed systems, from Middleware to Architecture

SIERA – Service IntEgration and netwoRk Administration

T2RS – Real-Time in networks and systems

TRACES – Trace stands for research groups in architecture and compilation for embedded systems

**CISO – HPC, Simulation, Optimization**

APO – Parallel Algorithms and Optimisation

REVA – Real Expression Artificial Life

**FSL – Reliability Systems and Software**

ACADIE – Assistance for certification of distributed and embedded applications

ARGOS – Advancing Rigorous Software and System Engineering

ICS – Interactive Critical Systems

SM@RT – Smart Modeling for softw@re Research and Technology

**GD – Data Management**

IRIS – Information Retrieval and Information Synthesis

PYRAMIDE – Dynamic Query Optimization in large-scale distributed environments

SIG – Generalized information systems

**IA – Artificial Intelligence**

ADRIA – Argumentation, Decision, Reasoning, Uncertainty and Learning methods

LILaC – Logic, Interaction, Language and Computation

MELODI – Methods and Engineering of Language, Ontology and Discourse

**ICI – Interaction, Collective Intelligence**

ELIPSE – Human computer interaction

SMAC – Cooperative multi-agents systems

TALENT – Teaching And Learning Enhanced by Technologies

**SI – Signals and Images**

MINDS – coMputational Imaging anD viSion

SAMoVA – Structuration, Analysis, Modeling of Video and Audio documents

SC – Signal and Communications

STORM – Structural Models and Tools in Computer Graphics

TCI – Images processing and understanding