Representing and Comparing Large Sets of Extensions of Abstract Argumentation Frameworks

Odinaldo Rodrigues
Departament of Informatics
King's College London
London, UK
odinaldo.rodrigues@kcl.ac.uk

ABSTRACT

In argumentation theory, some reasoning problems involve the enumeration of all *extensions* of an abstract argumentation framework. Abstractly speaking, extensions are simply subsets of a given domain having some special properties. The result of the enumeration is usually presented as a single text file with elements and sets separated by designated delimiters. Neither the elements within each set (a single extension), nor the extensions themselves are presented in any pre-defined order. Events such as the International Competition of Computational Models of Argumentation require the comparison of a large number of enumerations and thus performing the comparisons very efficiently has become very desirable. This paper presents and compares three different alternative representations of extensions, one of which is novel for the argumentation domain, and provides an empirical evaluation of their effectiveness in the comparison of large enumerations. We found that the newly proposed representation can perform the comparisons in a much more memory and time efficient manner than existing solutions.

CCS CONCEPTS

• **Theory of computation → Sorting and searching**.

KEYWORDS

Abstract argumentation frameworks, computation of semantics, verification of extensions

ACM Reference Format:
Odinaldo Rodrigues. 2019. Representing and Comparing Large Sets of Extensions of Abstract Argumentation Frameworks. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3297280.3297394

1 INTRODUCTION

An important area of Artificial Intelligence is the formalisation of human reasoning. *Abstract argumentation frameworks* (AAFs) have emerged as an increasingly important formalism to aid in the automation of this reasoning process. Formally speaking, an AAF is a directed graph $G = \langle A, R \rangle$, where A is a set of *arguments* and $R \subseteq A^2$ is an attack relation on them. *Semantics* for AAFs are given in terms of one or more *extensions* $E \subseteq A$ satisfying special properties.

Computing the extensions of an AAF is an important area of research, but a computationally hard problem [4, 5] and a number of computer programs called *argumentation solvers* have been proposed for this. Since 2015, the International Competition on Computational Models of Argumentation (ICCMA)[1] provides a bi-annual platform for the evaluation of the performance of the solvers against a collection of benchmark problems [9]. Some of these problems consist of checking the inclusion of an argument in one or all of the extensions (a *decision problem*) and the *enumeration* of one or all of the extensions of the AAF under a particular semantics.

The issues involved in the comparison of single extensions are subsumed by the considerations in the comparisons of the enumerations of sets of extensions, therefore this paper concentrates on the latter. The actual definitions of the argumentation semantics are not relevant to the comparison (the reader is referred to [1, 3] for an introduction). It suffices to say that the arguments of an AAF are provided to the solvers as distinct strings of characters and the solvers output extensions as sequences of characters as follows.

A single extension is represented as a sequence of characters of the form $[\mathtt{arg}_1, \mathtt{arg}_2, \mathtt{arg}_3, \ldots, \mathtt{arg}_m]$, where each $\mathtt{arg}_i$ is the sequence of characters representing an argument in the extension; and a set of extensions is represented as a sequence of terms of the form $[\mathtt{E}_1, \mathtt{E}_2, \ldots, \mathtt{E}_n]$, where each $\mathtt{E}_j$ is a single extension as before. In both cases, the sequences of characters are provided as a single text file and neither the arguments in each extension, nor the extensions themselves are presented in any particular order.

As the solvers become more sophisticated, the benchmark problems employed by ICCMA become harder to solve and the set of enumerations harder to verify. To put the competition's task in context, the latest ICCMA evaluated the performance of 14 solvers in 12 enumeration tracks. Each enumeration track consisted of 350 graphs, resulting in the potential verification of 58,800 solutions for the enumeration part alone. Although some solutions are small, others can be very large. For example, the graph

`massachusetts_srta_2014-11-13.gml.50`

has 34,307,712 complete extensions with nearly 1.2 billion elements in total, resulting in a text file over 4.4Gb in size. As a result, providing a tool to compare two enumerations in an efficient manner has become very important for future competitions.

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297394

[1] http://argumentationcompetition.org

In this paper, we are concerned only with the comparison of an enumeration of extensions (a *solution*) against a *master* solution obtained in an appropriate way. How to actually obtain the master solution in the first place is not our concern, but an option would be to "agree" on a solution that satisfies a certain set of constraints and is identical to the results provided by a minimum number of solvers. Although it would be possible to perform the verification of the solutions partially or fully in permanent storage, it would take considerably longer (see Section 7, for a discussion). Thus, for simplicity, we assume that the comparisons are to take place in main memory. We are therefore particularly interested in alternative forms of representation of the solutions, the time needed to perform the comparisons, and their overall memory requirements.

The rest of the paper is structured as follows. In Section 2 we present the general assumptions around which we designed our representations and based our analysis. In Section 3, we consider ordered and non-ordered alternatives for representation of (sets of) extensions. In Section 4, we present our novel approach based on the GNU's Multiple Precision Arithmetic Library (GMP, https://gmplib.org/). In Section 5, we compare it with the alternatives presented in Section 3 using enumerations of the extensions of the AAFs used in the 2nd ICCMA. The main structure of a comparison algorithm using our approach is described in Section 6. Finally, we conclude in Section 7 with a discussion.

2 PRELIMINARIES

The complete enumeration track of the 2nd ICCMA employed 350 AAFs in total. Out of these, the enumeration of the extensions of 78 AAFs could not be performed by any solver [7]. As the solutions to the argumentation problems of the competition have not been published, our considerations were guided by an analysis of 196 solutions of the complete enumeration track that we could generate ourselves employing a number of publicy available solvers (more details are given in Section 5). This represents 72% of the AAFs in the track which could be tackled by at least one solver. For brevity, we will refer to the file names used in the competition by a unique problem identifier (prob id). The correspondence of the prob ids with the original file names along with other general statistics about a selection of interesting graphs is given in Appendix A.

To our knowledge the only other available tool for the comparison of extensions is the one which is part of the benchmarking framework `probo` [2], used in the 1st ICCMA [9]. However, in our tests the tool was not able to cope with the magnitude of the solutions of the 2nd ICCMA.

Since a set of extensions is mathematically a set of sets, the following factors were important in the design of the data representations:

- The number of distinct arguments in the graph: `num_args`
- The number of distinct arguments in a solution: `num_sol_args`
- The number of elements in a solution: `num_sol_elems`
- The cardinality of the largest solution: `num_exts`
- The cardinality of the largest extension: `max_ext_size`

Since it is not our intention to verify the correctness of any solution (we only compare a solution against a master), we do not assume or require access to the original text file describing the graph. Therefore, we cannot assume knowledge of `num_args` either. Although `num_args` determines the maximum cardinality of any solution, in practice, it suffices to know the value of `num_sol_args` instead, which can be obtained by scanning the solution file itself. Similarly, since the number of actual elements in a solution (`num_sol_elems`) has a direct impact on the memory requirements of implementations representing each element as an *object*, we also considered it. Also relevant were the cardinality of the largest solution (as each extension requires space in the solution's internal representation) and the cardinality of the largest extension in a solution (for similar reasons). With all this in mind, we then analysed the 196 solutions we could generate to better understand the boundaries of the verification problem at hand. The solutions of the problems `152` and `157` stood out in this respect by having the maximum values of the above factors (see Table 1). In our discussion, we will often use the factors as functions taking the prob ids as arguments, so for example, we would assert that $\texttt{num\_exts}(152) = 34,307,712$.

Factor	Max Value	Occurring in Prob Id
`num_args`	2,000,000	`157`
`num_sol_args`	1,000,000	`157`
`num_sol_elems`	1,178,276,544	`152`
`num_exts`	34,307,712	`152`
`max_ext_size`	1,000,000	`157`

Table 1: Characteristic values of interest in solutions.

Any implementation will be of course ultimately dependent on the physical hardware of the host machine, but our considerations aimed to optimise resources yielding efficient implementations allowing modern personal computers to compare solutions even beyond the ranges given in Table 1.

We assume that the solutions to be compared are stored as text files in the format described in the Introduction. Without loss of generality, we also assume that each argument descriptor appearing in a solution (i.e., the string representing the argument in the text file) is converted to a unique natural number occupying at most 32 bits, and hence allowing for at most 2^{32} distinct arguments. This number can of course be increased in the future incurring a corresponding increase in the memory footprint of the representations. In the context of the competition, it is more than sufficient.[2] For clarity, we will refer to the data type representing the argument descriptors as `uint32_t` (a 32-bit unsigned integer).

Finally, we assume that the host machine is a computer using a 64-bit addressing mechanism and so pointers to an address in memory occupy 8 bytes. We assume that the comparisons of the solutions is to be made in memory (real and/or virtual) with input from text files but without access to an auxiliary external disk database.

Although our considerations are made specifically for the `C++` programming language, they should apply equally to most modern object-oriented languages.

3 CHOOSING APPROPRIATE CONTAINERS

As we mentioned, solutions to the enumeration problems are provided as simple sequences of characters. These sequences of characters represent sets of sets of arguments. In each sequence, neither

[2]The largest number of distinct arguments in any solution of the 2nd ICCMA was $1{,}000{,}000 \leq 2^{20}$.

the arguments nor the extensions are ordered in any particular way. Because of the potential number of extensions and their cardinality, it becomes unfeasible to compare two solutions without ordering their extensions and elements first.

At first, it may be tempting to choose a C++ associative container such as `set` to represent extensions and solutions, since its operations naturally map to the mathematical concept of *set*, the elements in the container are ordered, and it has a built-in equality comparison operator. Using this container, an extension would be represented by the data type `set<uint32_t>`. We could then represent a solution using the data type `set<set<uint32_t>>`. However, the type `set` is normally implemented as a red-black tree [8] and carries a considerable memory overhead per node: one pointer to the node's parent, and two for its left and right children. Let $mem(s, \mathtt{T})$ denote our *estimated* lower-bound memory requirement in bytes for the representation of the solution s using the data type `T`. Since a pointer occupies 8 bytes in a 64-bit machine, and assuming that the data in each node of the red-black tree is `uint32_t` (4 bytes), then the lower-bound memory requirement of a single node representing each element in a solution would be 28 bytes.[3] This would give $mem(s, \mathtt{set<set<uint32\_t>>}) = 28 \times \mathtt{num\_sol\_elems}(s)$.

In order to compare two solutions, we could store the master solution in memory and check the secondary solution as we parse each of its extensions from disk. This would not require storing or ordering the secondary solution and in the best case scenario the first extension read from the secondary solution would already be decisive, but in the worst case scenario, it would require checking every extension of the secondary solution against the extensions in the master solution. For simplicity we will assume that both solutions are fully read from disk and stored in memory. This means we need to double the memory requirements of the representation of a single solution to perform the comparison.[4]

From our analysis of the solutions, we found that the three problems with the highest number of elements in their solutions were `152`, `122` and `118`, in this order. Using `set<set<uint32_t>>`, each of these solutions would require the following memory allocation:

prob id	num_sol_elems	$mem(s, \mathtt{set<set<uint32\_t>>})$	
152	1,178,276,544	32,991,743,232	(~31,463MB)
122	221,836,056	6,211,409,568	(~5,923MB)
118	187,067,232	5,237,882,496	(~4,995MB)

Thus, using `set<set<uint32_t>>` we would need about 63Gb just to represent the two solutions for problem `152`. Although this can be done in computers with a large amount of memory, it is clearly not memory efficient, making it less scalable to future problems that may generate even larger solutions. We must therefore consider other containers with a smaller memory footprint.

If we represent an extension as an ordinary sequence of elements, and solutions as sequences of extensions, we could use the sequence container `vector`, which only requires roughly 3 pointers per *vector* (24 bytes). Including the 4 bytes used in the representation of the arguments in each vector cell, this would require:

$$mem(s, \mathtt{vector<vector<uint32\_t>>}) = 4 \times \mathtt{num\_sol\_elems}(s) + 24 \times \mathtt{num\_exts}(s) + 24$$

[3] This is very platform-dependent and hence only an estimate, but the considerations are similar for all modern programming languages and platforms.

[4] This strategy deserves further investigation, particularly in systems with less memory.

Using this representation, the problems `152`, `122` and `118` would be more manageable (`vv32` abbreviates `vector<vector<uint_32t>>`):

prob id	num_sol_elems(s)	num_exts(s)	$mem(s, \mathtt{vv32})$	
152	1,178,276,544	34,307,712	5,536,491,288B	(~5280MB)
122	221,836,056	7,788,996	1,074,280,152B	(~1024MB)
118	187,067,232	1,594,323	786,532,704B	(~750MB)

As vectors are not naturally sorted, once its elements are inserted, they need to be sorted appropriately for future comparison. In the case of extensions, this is a simple sorting over the type `uint32_t`. Solutions can be sorted using the lexicographical ordering. These algorithms are readily available in C++'s Standard Template Library [6, 8]. We then need to factor in the cost of the sorting operations in addition to the cost of reading and inserting the elements, yielding:

prob id	data type	(insertion+sorting)=total time
152	`vector<vector<uint32_t>>`	(116.62s+37.63s)=154.25s
	`set<set<uint32_t>>`	508.23s
122	`vector<vector<uint32_t>>`	(21.71s+5.66s)=27.37s
	`set<set<uint32_t>>`	76.13s
118	`vector<vector<uint32_t>>`	(16.97s+3.40s)=20.37s
	`set<set<uint32_t>>`	65.20s

The numbers above clearly show that `vector<vector<uint32_t>>` is much more time efficient than `set<set<uint32_t>>`. We have seen that it is also much more memory efficient, so our first conclusion is that we should avoid the use of associative containers and use sequence containers instead.

A more detailed analysis of memory and time requirements of each representation can be found in Figures 1 and 2 and Appendix A. In the next section, we show that the representation of extensions can be improved further.

4 A COMPACT AND EFFICIENT REPRESENTATION OF EXTENSIONS

We now aim to simplify the internal representation of extensions, simultaneously saving on memory and improving the performance of the comparison operations. The basic idea comes from the realisation that in the representations considered in the previous section, each argument belonging to an extension consumes *at least* 4 bytes. If we were to allocate a single bit position for every argument appearing in a solution, then each extension in it could be represented by a natural number whose binary counterpart contained "`1`" or "`0`" in a position depending on whether the corresponding argument belongs (resp., does not belong) to the extension. In this approach, the memory requirement per argument is a fraction of the requirement of previous representations, i.e., 1 bit per argument, but now the size of each extension is determined by the argument in it whose corresponding bit position is the most significant, so there is a trade-off to be considered. Note that we only need to consider the arguments in the solution, which are usually a proper subset of A, except in some extreme scenarios (see [1, 3] for an introduction on argumentation semantics). Example 4.1 illustrates these ideas.

Example 4.1 (From solutions to lists of natural numbers). Consider the conversion of the solution $s = [[\mathtt{a2}, \mathtt{a0}], [], [\mathtt{a1}, \mathtt{a2}]]$ into a list of natural numbers. Each extension e_i of s is first turned into the natural number $n(e_i)$ as follows.

As we parse s, we encounter its first extension e_1 = `[a2,a0]` and set $n(e_1) = 0$. We then identify the argument `a2` as a new argument within s, assigning to it the first available bit position, i.e., 0. This bit position is *set* in e_1's numerical representation $n(e_1)$, which started with the value 0 (empty). This operation is the same as setting $n(e_1) = n(e_1) + 2^0 = 1$, but bit operations are much faster than arithmetic ones and readily available in the class `mpz_class`. We then read `a0`, the second element of e_1, which is again a new element in s and given the bit position 1. e_1's conversion is then completed by setting $n(e_1) = n(e_1) + 2^1 = 3$ (note that 3 is 11 in binary).

We now read the second extension e_2 = [], which is empty, so it gets the value $n(e_2) = 0$. Finally, we read extension $e_3 = [a1, a2]$, whose first element is new and gets bit position 2. This sets $n(e_3) = 0 + 2^2 = 4$. e_3's second element is `a2`, which is known to occupy bit position 0, so $n(e_3) = 4 + 2^0 = 5$, which is 101 in binary.

Therefore, the solution s is converted into the list of natural numbers $[3, 0, 5]$, which is internally represented as a vector of elements of any unsigned integer type, say `uint32_t`. The vector is then sorted in numerical order to allow quick comparison with other solutions, yielding $[0, 3, 5]$.

There is a slight complication with this representation which has to do with the largest unsigned integer that can be represented *natively* in the host machine. In a modern 64-bit computer this is a 64-bit long integer, meaning that any implementation can only natively represent solutions with a maximum number of 64 distinct arguments. We saw in Section 2 that problem 157 had the largest number of distinct arguments in any solution, namely $1,000,000$ arguments. This means that extensions in these solutions may potentially need an integer with as many bits, i.e., $125,000$ bytes long.[5] Fortunately, there are a few readily available libraries that deal with integers of arbitrary precision. We chose GNU's Multiple Precision Arithmetic Library (GMP, https://gmplib.org/), for reasons that we will explain next.

4.1 The GMP Library

The GMP library is a well-established publicly available library for arbitrary precision arithmetic, operating on a variety of numerical types. The precision of GMP's basic data types is only limited by the characteristics of the host machine. For our application, we are especially interested in integers of very large precision. Although GMP is optimised for speed, its memory allocation policy makes it especially suited for our representation as explained below.

GMP's integers are represented as objects of the class `mpz_class` in space dynamically allocated and reallocated. The idea is to divide the sequence of bits of a long integer into a number of contiguous blocks of memory of fixed size called *limbs*. The library ensures that the representation will only require a number of limbs proportional to the most significant bit in the integer. More importantly, the number of limbs will grow automatically to accommodate any increase in the integer being manipulated (in our case, the number of limbs will increase when we read an argument whose bit position is higher than the most significant bit in the integer representing the extension).

[5] This compares much more favourably with the memory requirements of the `vector<uint32_t>` representation: $(4 \times 1,000,000) + 24 = 4,000,024$ bytes.

4.2 Memory Considerations

GMP is not the only available alternative to overcome the limitations of the native integer data types in `C++`. For example, the data type `bitset` works as an array of boolean elements optimised for space. We opted for the type `mpz_class` instead for two reasons. Firstly, the data type `bitset` requires the number of required bits to be known at *compilation* time, but we do not know how many bits we will need to represent a particular extension until we read it. We could use an arbitrarily large number of bits at the outset, but this would be very space inefficient. Secondly, we want each *individual* extension to use only as few bits as necessary to represent it and extensions using the `mpz_class` will only occupy the minimum number of limbs necessary to accommodate the element in the extension associated with the most significant bit position.

It is not easy to fully estimate the memory requirements of the `mpz_class` representation, but it is possible to check at run-time how many limbs have been allocated to each extension and hence how many bytes each extension (and solution) uses. We have done this for the 196 solutions considered and the total number of MBs used by the ones taking longer than one second to compare is given in Appendix A.

Figure 2 also compares the memory requirements of the three data representations for all 196 solutions checked. There is obviously a trade-off between the number of distinct arguments in a solution (which sets an upper-bound on the minimum number of bits needed to represent a single extension) and the total number of extensions in the solution, but the figure shows that in practice the `mpz_class` proved much more memory efficient at representing extensions than the data type `vector<uint_32>`, whereas also being much faster when used to compare solutions (see Figure 1 and Section 5).

4.3 Discussion

The advantanges of representing extensions in this way are very clear: 1) the memory cost per element is very low, particularly in AAFs whose solutions have a relatively small number of distinct arguments; 2) there is no need to order the elements within an extension. We only need to assign each argument in the solution a unique index (i.e., a bit position) and then the binary representation of an extension can be directly and uniquely constructed by setting the bits corresponding to the arguments in the extension; 3) as each extension corresponds to a natural number, the list of extensions within a solution can be easily ordered and hence solutions can be quickly compared. In fact, comparing two solutions s_1 and s_2 with this representation is trivial because we can simply use the vector comparison operator. The worst-case scenario is when the solutions have the same cardinality and differ only on their last extension, but this can still be done in linear time. It is possible to improve on the comparison even further. As the second solution is read, any argument for which a bit position was not reserved by the first solution flags a difference between the solutions and the process can be stopped there without the need to read the remainder of the second solution. We did not implement this in our prototype, because we wanted to force the complete reading of both solutions in our results.

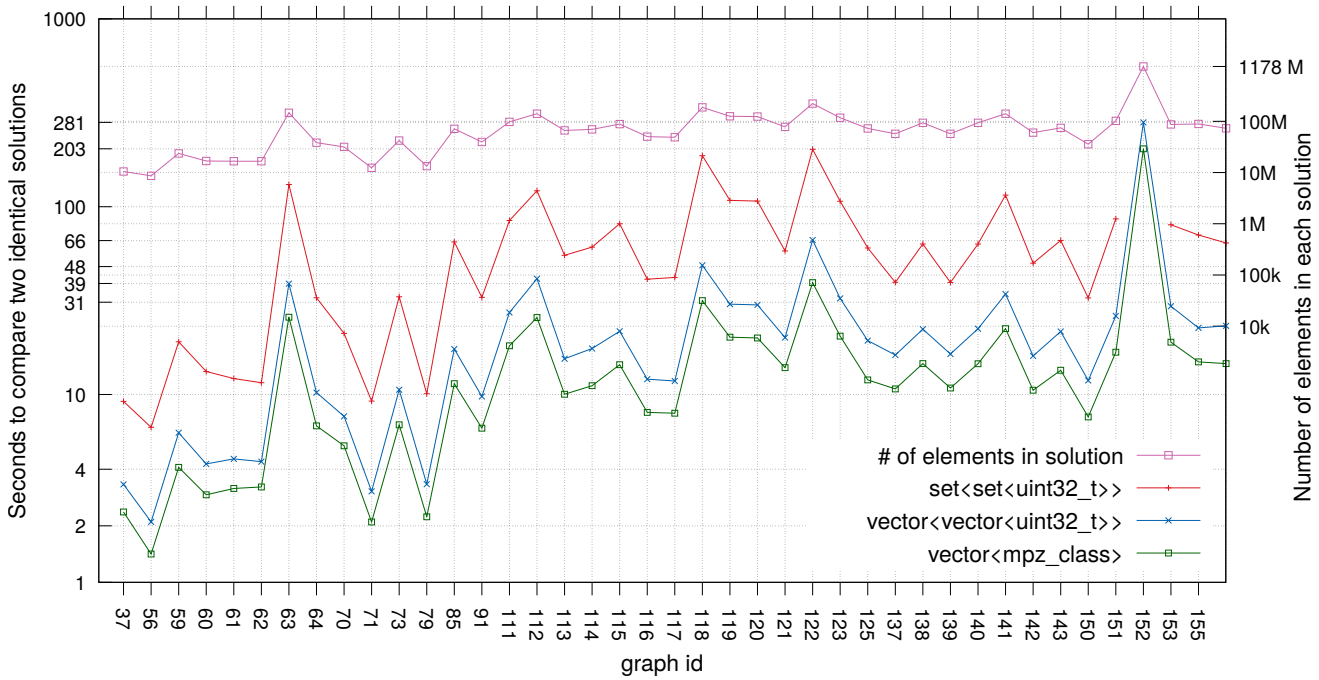

Figure 1: Time to compare pairs of solutions using the three data representations

5 EMPIRICAL ANALYSIS

For our benchmark we chose the solutions of the enumeration of all complete extensions track of the 2nd ICCMA because these solutions contain the largest numbers of sets. However, since the solutions to the AAFs used in the competition have not been published, we restricted the evaluation to the solutions we could obtain ourselves by running a collection of solvers. As mentioned in Section 2, we managed to generate 196 out of the 350 AAFs employed in this track of the competition. 78 AAFs in the track could not be solved by *any* solver, therefore these 196 solutions cover 72% of the graphs employed in the complete enumeration track for which at least one solution was found by any solver.[6] This subset of solutions proved sufficiently large for the purposes of our analysis, offering a reasonably good combination of the likely types of solutions.

Our implementation employed the three data structures `set<set<uint32_t>>`, `vector<vector<uint32_t>>`, `vector<mpz_class>` and analysed the performance and the estimated memory requirements of each on the 196 solutions. We stress that our objective was not to verify the actual *correctness* of the *master* solution, but simply to compare a given solution against *a* solution designated as master. Indeed, we could have evaluated the performance on sets of arguments arbitrarily generated, but using actual solutions allowed us to make our analysis more faithful to the problems used in the competition.

Our test machine had an Intel i7 4790K CPU with 32Gb of memory. The chart in Figure 1 depicts the relative times to compare two identical solutions using the three data representations as well as the overall number of elements in each solution. The y-axis on the left of the chart indicates the time in seconds and the one on the right indicates the number of elements in each solution. To simplify the presentation, we excluded the solutions that took less than 1s to compare. It is easy to see that there is a high correlation between the number of elements in the solution and the time taken to compare the two solutions using all three data representations (> 0.99). Since both y-axes are logarithmic in scale, it is worth emphasising the performance gain using the `mpz_class` representation. The largest solution (problem 152) could be compared in 203s using this representation, against a time of 281s using `vec<vec<uint32_t>>`. Its comparison using `set<set<uint32_t>>` exhausted our computer's memory and could not be completed. The times taken to compare a pair of solutions of the three hardest problems can be found in Table 2, whereas Figure 2 also depicts the estimated memory requirements of the comparison using the three data representations. The figures show the clear advantages of the `mpz_class` representation both in terms of time and memory allocation. Overall, the representation using `vec<mpz_class>` took roughly 2/3 of the time taken by the `vec<vec<uint32_t>>` representation and 1/5 of the time taken by the `set<set<uint32_t>>` representation to compare all solutions.

Prob id	`set<set<uint32_t>>`	`vec<vec<uint32_t>>`	`vec<mpz_class>`
152	-	281.06s	203.27s
122	202.04s	66.44s	39.40s
118	186.91s	48.76s	31.61s

Table 2: Time taken to compare the three hardest solutions.

6 ALGORITHMS

Algorithms 1, 2 and 3 provide the general structure of the implementation. The main program uses three global variables: two vectors of objects of the class `mpz_class`, namely *sol_master* and

[6] Some results found only by a single solver also deserve further scrutiny (see [7]).

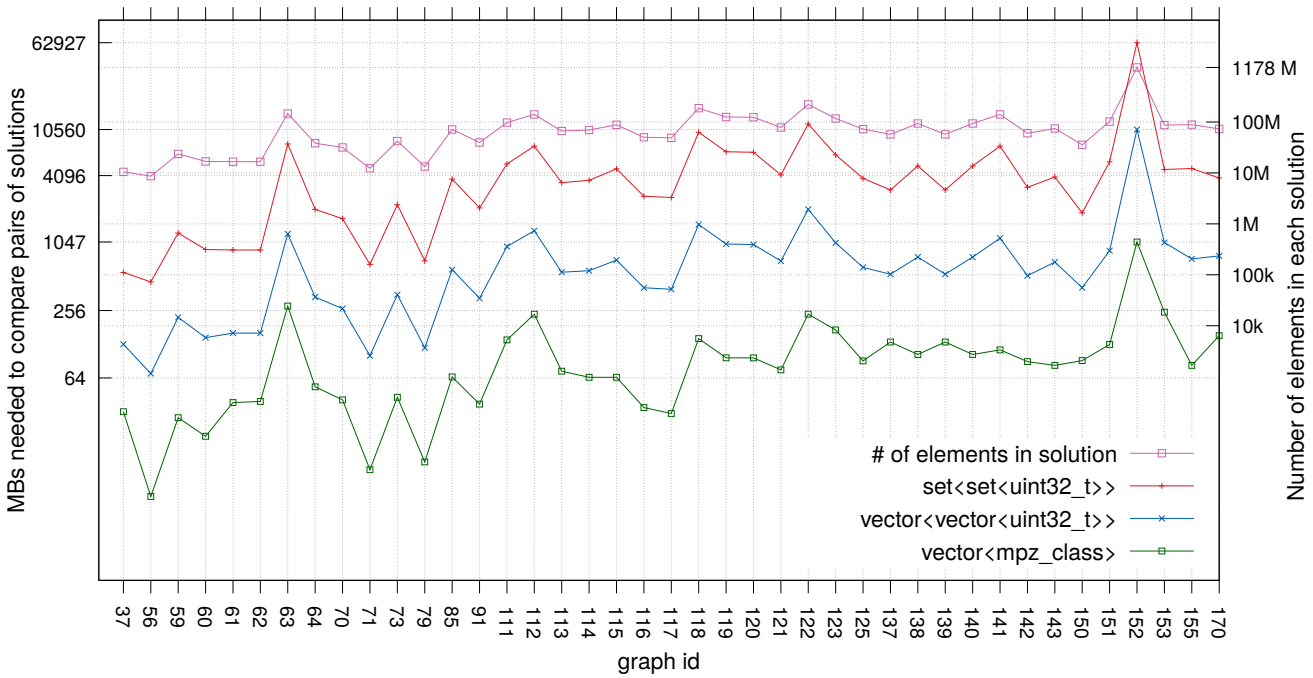

Figure 2: Approximate memory requirements to compare pairs of solutions using the three data representations

sol_target, and the dictionary *index* mapping arguments (strings) into integers of type `unit32_t`, and the two auxiliary functions `parse_solution` (Algorithm 2) and `parse_extension` (Algorithm 3). `parse_solution` receives as input a text file and returns an *ordered* vector of objects of the class `mpz_class` representing the extensions in the file, which are then assigned to the variables *sol_master* and *sol_target*. `parse_solution`, defined in Algorithm 2, works by successively scanning the text file for the delimeters of an extension ("`[`" and "`]`", omitted here). The string within the delimiters is then used by `parse_extension` (Algorithm 3) that performs the main conversion from string to an object of the class `mpz_class`. This conversion simply requires the successive parsing of arguments within the string representing the extension. It uses the global dictionary *index* (line 5). If the argument has not yet been parsed within the solution it is stored in the dictionary assigning it the next available bit position.[7] The index of each argument is used to set the corresponding bit within the long integer variable *mpz_ext*, which is initially set to 0 (representing the empty extension). Once all arguments within the extension are parsed and processed in this manner, *mpz_ext* will contain a unique natural number corresponding to the extension. This is because *index* is a function from arguments to bit positions, and hence `parse_extension` provides a one-to-one mapping between extensions and natural numbers.

Finally, the two vectors *sol_master* and *sol_target* can be compared directly using the vector class' own comparison operator that internally uses the element comparision operator provided by the class `mpz_class` (Algorithm 1, line 5). As we mentioned, this can be optimised further by exiting early with failure when arguments unknown to the first solution are read in the second.

Input : *master* and *target*: text files to compare
Output: The result of the comparison

```
1 Main Program
2     index← ∅;
3     sol_master← parse_solution(master);
4     sol_target← parse_solution(target);
5     if sol_master == sol_target then print "OK";
6     else print "Solutions differ";
7 end
```

Algorithm 1: Compare two solutions given as files

Input : *source* the file with the solution
Output: an ordered vector of long integers

```
1  Function parse_solution(source)
2      str_ext← parse_extension(source)
3      while str_ext ≠ null do
4          int_ext← convert_to_mpz(str_ext)
5          mpz_sol.push_back(int_ext)
6          str_ext parse_extension (source)
7      end
8      sort(mpz_sol);
9      return mpz_sol;
10 end
```

Algorithm 2: Read extensions from a source file

[7] In practice, since the first position is 0, this is just the current number of elements in *index*, which is naturally incremented after insertion.

Input : *str_ext* the string representing the extension
Output: a long integer corresponding to the extension

```
1  Function convert_to_mpz(str_ext)
2      mpz_ext← 0;
3      str_arg← parse_argument(str_ext);
4      while str_arg <> null do
5          if index(str_arg) == null then
6              index.insert(str_arg)
7          end
8          mpz_ext.setbit(index(str_arg));
9      end
10     return mpz_ext;
11 end
```

Algorithm 3: Convert an extension to a natural number

7 CONCLUSIONS

In this paper, we considered three alternative representations of sets of extensions of AAFs (here called *solutions*) provided as unstructured text files. In an abstract sense, solutions are simply sets of sets of elements taken from a domain A. As neither the elements within an extension nor the extensions within a solution are provided in any specific order, comparing large solutions efficiently involves some sorting of elements and extensions.

We first investigated sorted containers of the type `set` in `C++` to represent both extensions and solutions. We found that for this particular application, it does not yield an efficient implementation in terms of time or space. In particular, we were unable to complete the comparison of the solutions with the largest number of extensions (problem `152`) in our test machine with 32Gb of physical memory. This is because the data type `set` in `C++` is usually implemented as a red-black tree [8] and has an overhead of at least three pointers per node (i.e., 24 bytes) to keep track of the tree structure in addition to the space needed for the data itself (another 4 bytes in our case). We then considered the more space efficient `vector` data type using one integer for each element in the extension, with a much smaller memory overhead. Each extension was then inserted into a vector of extensions to represent the solution. One drawback of this representation is that extensions and solutions need to be sorted to yield efficient comparisons of solutions. In our tests, we found that the standard sorting algorithms in `C++`'s STL were adequate for this purpose, achieving better performance than with the use of the data type `set` itself (which does not require post-sorting of solutions and extensions). Using the `vector` data type we were able to complete all comparisons of solutions in a fraction of the original time taken by the implementation using `set`.

Finally, we investigated how to improve on the representation further. Our technique allocates a bit of information for each element in a solution, using the bit "`1`" to represent the membership of an element in an extension and "`0`" otherwise. This representation generates a unique large natural number for each extension, dispensing with the sorting of elements within extensions and making the sorting of extensions within solutions trivial. The difficulty here is that there may be more elements in a solution than the number of bits in the native integer data type of the host machine (which is 64 bits long in modern computers). This limitation was overcome by using the integer data types of arbitrary precision provided by the GNU Multiple Precision Arithmetic Library.

The (long) unsigned integer representation is not guaranteed to be always more space efficient than the representation using vectors of vectors of integers, because a small extension with arguments of high indices can potentially require more bytes this way. However, our empirical tests showed that overall it proved more efficient both in terms of space and time to compare. As an example, comparing two identical (and the largest) solutions for the problem `152` took 203s using the representation (~ 1Gb for both solutions) as opposed to 281s using the `vector` data type (resp., ~ 10Gb).

Collectively, the 196 solutions in our benchmark could be compared in 713s in total using vectors of objects of the `mpz_class` against 1,068s using vectors of vectors of integers. Memory usage was a lot more efficient as well.

Finally, if the two sets of extensions eventually become too large to wholly represent in main+virtual memory, the next best alternative is to store the master solution in memory and check the secondary solution as it is read from disk. If even the master solution will not fit in main memory, then the only alternative is to use disk for the majority of the operations. If sorting cannot be done in memory, then a possibility is to store the solutions as tables of a relational database (again the compact representation of extensions proposed in Section 4 can be used). Since disk operations are relatively slow, the comparison is likely to take a lot longer.

We hope the ideas presented here can be helpful particularly for teams developing argumentation solvers and researchers performing benchmarks. The representation of extensions as natural numbers itself can be used wherever a compact representation of sets of elements of a fixed domain is needed, for instance to represent conflict sets and extensions of argumentation frameworks. This representation is not only compact, but also offers very efficient insertion and membership checking times. The prototypical implementation used in our analysis is available on request.

REFERENCES

[1] P. Baroni, M. Caminada, and M. Giacomin. 2011. An introduction to argumentation semantics. *The Knowledge Engineering Review* 26, 4 (2011), 365âĂŞ410. https://doi.org/10.1017/S0269888911000166

[2] F. Cerutti, N. Oren, H. Strass, M. Thimm, and M. Vallati. 2014. A Benchmark Framework for a Computational Argumentation Competition. In *Proceedings of the 5th International Conference on Computational Models of Argument*. IOS Press, Pitlochry, Scotland, 459–460.

[3] P. M. Dung. 1995. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence* 77 (1995), 321–357.

[4] P. E. Dunne and M. Wooldridge. 2009. Complexity of Abstract Argumentation. In *Argumentation in Artificial Intelligence*, G. Simari and I. Rahwan (Eds.). Springer US, Boston, MA, 85–104. https://doi.org/10.1007/978-0-387-98197-0_5

[5] M. Kröll, R. Pichler, and S. Woltran. 2017. On the Complexity of Enumerating the Extensions of Abstract Argumentation Frameworks. In *Proceedings of the 26th International Joint Conferences on Artificial Intelligence, IJCAI-17*. 1145–1152. https://doi.org/10.24963/ijcai.2017/159

[6] S. B. Lippman, J. Lajoie, and B. E. Moo. 2012. *C++ Primer* (5th ed.). Addison-Wesley Professional.

[7] O. Rodrigues, E. Black, M. Luck, and J. Murphy. 2018. On Structural Properties of Argumentation Frameworks: Lessons from ICCMA. In *Proceedings of the Second International Workshop on Systems and Algorithms for Formal Argumentation, SAFA 2018*. 22–35. http://ceur-ws.org/Vol-2171/

[8] R. Sedgewick. 1998. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Third Edition* (third ed.). Addison-Wesley Professional.

[9] M. Thimm, S. Villata, F. Cerutti, N. Oren, H. Strass, and M. Vallati. 2016. Summary report of the first international competition on computational models of argumentation. *AI Magazine* 37, 1 (2016), 102–104.

A. MEMORY CONSUMPTION (PER SOLUTION) AND TIME TAKEN TO COMPARE PAIRS OF SELECTED SOLUTIONS

In the table below, su stands for `set<set<uint32_t>>`, vu stands for `vector<vector<uint32_t>>`, and vm stands for `vector<mpz_class>`.

Id	Problem	# exts	# elems	Distinct arguments	Megabytes su	Megabytes vu	Megabytes vm	Seconds su	Seconds vu	Seconds vm
37	sembuster_60	1048596	10486160	40	280	64	16.00	9	3	2
56	BA_160_20_3	91854	8636463	117	230	35	2.80	6	2	1
59	BA_60_60_3	927261	23581386	58	629	111	14.15	19	6	4
60	BA_80_50_1	406782	16796160	68	448	73	9.66	13	4	2
61	bw2.pfile-3-02.pddl.2.cnf	743510	16631534	156	444	80	19.32	12	4	3
62	bw2.pfile-3-08.pddl.2.cnf	743510	16631534	156	444	80	19.74	11	4	3
63	bw3.pfile-3-04.pddl.2.cnf	2558258	146860679	297	3921	618	140.35	131	38	25
64	commute.org_20140813_1738.gml.20	991440	38288340	71	1022	168	26.67	32	10	6
70	ferry2.pfile-L2-C1-04.pddl.4.cnf	540207	31658406	137	845	133	20.39	21	7	5
71	ferry2.pfile-L2-C3-01.pddl.1.cnf	151146	12359385	133	330	50	4.86	9	3	2
73	go-taps_20150524_1019.gml.20	708588	42160986	92	1125	177	21.50	33	10	6
79	kanawha-valley-regional-transportation-authority-krt_20150126_1340.gml.20	373248	13343616	59	356	59	5.70	10	3	2
85	south-metro-area-regional-transit_20151216_1704.gml.20	1071630	71314425	93	1904	296	32.70	64	17	11
91	thecomet_20131025_1906.gml.20	612360	39511800	96	1055	164	18.69	32	9	6
111	aircoach_20130716_1324.gml.50	4587840	97476048	55	2602	476	70.00	84	27	18
112	anaheim-resort-transportation_20151217_1213.gml.50	5349240	140676048	65	3756	659	118.68	121	41	25
113	BA_100_40_5	1220346	66414816	80	1773	281	36.79	55	15	10
114	BA_120_50_4	1062882	69618771	100	1859	289	32.44	60	17	11
115	BA_140_30_3	1062882	88219206	104	2355	360	32.44	81	21	14
116	BA_140_30_4	570807	50165406	117	1339	204	17.42	41	12	8
117	BA_160_30_2	505197	48894759	120	1305	198	15.42	42	11	7
118	BA_180_30_2	1594323	187067232	138	4995	750	71.91	186	48	31
119	BA_180_30_4	1062882	125361027	143	3347	502	48.44	108	30	20
120	BA_180_30_5	1062882	123884802	145	3308	496	48.34	107	29	19
121	BA_60_80_1	2480058	77944680	56	2081	354	37.84	58	20	13
122	BA_60_90_4	7788996	221836056	59	5923	1024	118.85	202	66	39
123	BA_80_70_5	2918202	117411069	74	3135	514	85.92	106	32	20
125	el-dorado-transit_20151217_1024.gml.20	1497852	72504612	85	1936	310	45.46	60	19	11
137	ferry2.pfile-L2-C1-03.pddl.1.cnf	2268636	57131690	123	1525	269	66.99	39	16	10
138	ferry2.pfile-L2-C1-04.pddl.6.cnf	1192716	93575280	178	2498	384	51.87	63	22	14
139	ferry2.pfile-L2-C1-05.pddl.1.cnf	2268636	57131690	123	1525	269	66.99	39	16	10
140	ferry2.pfile-L2-C1-09.pddl.6.cnf	1192716	93575280	178	2498	384	51.87	63	22	14
141	ferry2.pfile-L2-C4-05.pddl.1.cnf	1255338	140724702	177	3757	565	57.03	115	34	22
142	ferry2.pfile-L3-C1-07.pddl.1.cnf	1414107	60213747	130	1607	262	44.77	50	16	10
143	fresno-county-rural-transit-agency_20151216_1934.gml.50	2716254	74539521	61	1990	346	41.45	66	21	13
150	irvine-shuttle_20091229_1547.gml.80	3000618	35673102	41	952	204	45.79	32	11	7
151	massachusetts_nrta_2014-12-03.gml.20	2125764	101918574	87	2721	437	63.69	86	26	16
152	massachusetts_srta_2014-11-13.gml.50	34307712	1178276544	63	31463	5280	523.49	N/A	281	203
153	metro-bilbao_20110407_1059.gml.80	8111740	86993570	39	2322	517	123.78	80	29	18
155	yamhill-or-us.gml.20	1358127	88809696	103	2371	369	41.45	70	22	14
157	admbuster_2000000	1	1000000	1000000	26	3	0.24	1	0	0
170	carroll-transit-system_20151202_1957.gml.50	5004804	73107934	52	1952	393	76.37	64	23	14