Representing and Manipulating Large Sequences of Argumentation Labellings

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

ABSTRACT

This paper proposes a canonical ordering of arguments within abstract argumentation labellings and two new types of efficient representations of these labellings for use in applications involving the computation of argumentation semantics. The space requirements of the representations are analysed, benchmarked on a class of hard enumeration problems taken from the International Competition on Computational Models of Argumentation (ICCMA), and compared for efficiency. We found that they both offer significant reductions of the memory representation requirements of large labellings, sometimes of up to 75%. We argue that the new way of looking at labellings provided by one of the representations, i.e., by considering repetitions of segment assignments within labellings, paves the way for investigations of new applications in argumentation theory.

CCS CONCEPTS

• **Theory of computation** → **Data structures design and analysis**;

KEYWORDS

argumentation theory, computation models of argumentation, representation of labellings

ACM Reference Format:
Odinaldo Rodrigues. 2023. Representing and Manipulating Large Sequences of Argumentation Labellings. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-March 31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3555776.3577756

1 INTRODUCTION

Representing large sequences of "tokens" efficiently is critical in any computer science application that needs to store and reason with the sequences in a computer's physical memory simultaneously. In artificial intelligence and argumentation theory, large sequences can appear as the result of the computation done for reasoning problems in the so-called *abstract argumentation frameworks* (AFs), which are presented as a tuple $\langle A, R \rangle$, where A is a set of arguments and R is an attack relation on $A \times A$.

Generically speaking, given an AF $\langle A, R \rangle$, an *enumeration* problem can be viewed as the computation of all subsets of A, whose arguments can all be jointly "accepted" according to some conditions. Those subsets of acceptable arguments are called *extensions*. Obviously, each extension is an element of the power set of A and hence for a given argumentation framework, there could be exponentially many such extensions. During the computation of extensions, we often need to reason with the status of all arguments, and hence enumeration problems involve dealing with finite, but potentially very large sequences of "labels" – tokens that indicate whether arguments are *accepted*, *defeated*, or *undecided* [2, 9]. The assignment of labels to arguments is called a *labelling* [7].

In the specific case of argumentation theory, we would argue that in applications where arguments are mined or generated automatically, resulting in complex or large AFs, large labellings are inevitable and therefore the investigation of efficient internal representations is worthwhile.

In this paper, we propose a canonical ordering of arguments within labellings and two new types of compact representations which still allow direct access to the arguments' labels. The first representation mimics the topological structure of the AF whereas the other packs as many labels as possible per unit of memory. The first representation takes advantage of repetitions of segments naturally occurring within labelling enumerations. By storing just one copy of each repetition and keeping track of the correct segment order, a full labelling can be reconstructed for easy reference. We would argue that the division of the labellings into these types of segments paves the way for new applications in argumentation theory, e.g., the analysis of the repetitions associated with the segments, the investigation of implicit dependencies that may exist between AF's components, and potential simplifications of the AF's themselves.

The second representation is the densest possible whilst still allowing a direct association of labels to arguments. If used in conjunction with the canonical ordering of the first representation, it offers all the benefits with few shortcomings.

We evaluated the memory requirements of both representations on a class of problems taken from ICCMA'17 known to generate a very large number of labellings, compared them against a baseline approach, and then discussed the results of the analysis in detail. Our analysis show that both representation offer significant reductions of the memory requirements, sometimes of up to 75%.

The rest of the paper is structured as follows. In Section 2, we give a brief overview of concepts of argumentation theory needed for the understanding of the representations. In Section 3, we propose

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC '23, March 27-March 31, 2023, Tallinn, Estonia
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577756

a canonical way of ordering arguments within labellings, which we use in our novel representation. In Section 4, we present our baseline representation and the first novel representation, followed by the second proposed representation in Section 5. We then conclude with an experimental analysis and some final remarks in sections 6 and 7, respectively.

2 BACKGROUND

Abstract argumentation frameworks (AFs) were proposed by Dung [8] as a system for reasoning about the acceptability of arguments. Formally, an AF is a directed graph $\langle A, R\rangle$, where A is a *finite* non-empty set of arguments and R is a binary relation on A, called the *attack relation*. When $(x, y) \in R$, we say that x *attacks* y and depict it with an arrow $\longrightarrow$ from x to y. Figure 1 shows the sample AF $\mathcal{A}$ which we will use as our running example.

Based on the relation R, the immediate neighbourhood of an argument identifies the arguments attacking it and the arguments that it attacks. These two concepts are of particular interest.

Definition 2.1. Let $\langle A, R\rangle$ be an AF and $x \in A$. The set of x's attackers x^- is defined as $x^- = \{y \in A \mid (y, x) \in R\}$. The set of arguments that x attacks x^+ is defined as $x^+ = \{y \in A \mid (x, y) \in R\}$.

If $x^- = \varnothing$, then we say that x is an *initial* argument [5]. If $x \in x^-$, then we say that x *self-attacks*. We extend the notion of attackers and attacked arguments to an arbitrary set $X \subseteq A$, as follows: $X^- = \bigcup_{x \in A} x^-$ and $X^+ = \bigcup_{x \in A} x^+$.

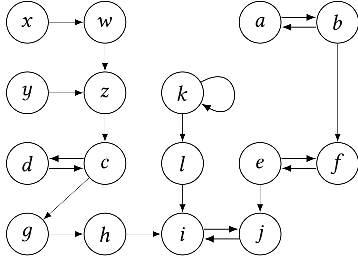

Figure 1: The sample argumentation framework $\mathcal{A}$.

In deciding whether an argument is "accepted" or "defeated", one needs to analyse whether any of the argument's attackers is accepted. If this is the case, the argument is defeated and it is accepted otherwise. Hence, initial arguments (having no attackers) are always accepted. For example, x and y in $\mathcal{A}$, are always accepted. An attacked argument can still persist, as long as all of its attackers are defeated. An example of this in $\mathcal{A}$ is when g is defeated by c, allowing h to be accepted. Arguments in cycles are treated similarly, but this sometimes requires an element of choice. In the case of the argument k of $\mathcal{A}$, we have no option because the argument cannot be accepted and defeated at the same time. In such cases we say that the argument's status is *undecided*. However, in the case of the arguments a and b, which are also involved in a cycle, we can either leave them both undecided, or accept one and the reject the other. This results in three potential outcomes for the arguments in this particular cycle.

The discussion above hints that arguments can be determined recursively following the reverse direction of attack, dealing with cycles as a special case. In Dung's original formulation, the concept of an *extension* is associated with a subset of the set of arguments having some desired special properties. Before we can describe the properties, we need to introduce some underlying concepts.

Let $\langle A, R\rangle$ be an AF. A set $E \subseteq A$ is said to be *conflict-free* if for all elements $x, y \in E$, we have that $(x, y) \notin R$. Moreover, an argument $x \in A$ is said to be *acceptable with respect to* E, if E "protects" it, i.e., for all $y \in x^-$, $E \cap y^- \neq \varnothing$. A set E is then said to be *admissible* if it is conflict-free and all of its elements are acceptable with respect to itself. Extensions are special types of admissible sets.

Definition 2.2 (Complete extensions). An admissible set $E \subseteq A$ is a *complete extension* if and only if E contains all arguments which are acceptable with respect to itself.

The $\subseteq$-minimal complete extension is called the *grounded extension*. A complete extension E is called a *preferred extension* iff E is a $\subseteq$-maximal complete extension; and a preferred extension E is also *stable* if $E \cup E^+ = A$.

Every AF has at least one complete extension, exactly one grounded extension, one or more preferred extensions, and zero or more stable extensions. In this paper we will concentrate on the enumeration of complete extensions because it is the problem whose associated labellings generate the largest amount of data. Our concern is how to represent *all* such labellings efficiently in memory. Even smaller argumentation frameworks can generate a surprisingly large number of complete extensions [16]. The AF of Figure 1 has 21 complete extensions (see Table 1).

During the computation of extensions, we will use the special labels **in**, **out**, or **und** to represent the status of arguments. **in** means the argument is "accepted" (belongs to the extension), **out** means it is "defeated" (it is attacked by an argument belonging to the extension), and **und** means it is "undecided" (neither accepted nor rejected). The association of labels to arguments can be formally captured by a *labelling function*.

Definition 2.3 (Labelling function). A *labelling function* f is a function of the form $f : A \longrightarrow \{\mathbf{in}, \mathbf{out}, \mathbf{und}\}$.

Let dom denote the domain of a function and define $in(f) = \{x \in \text{dom } f | f(x) = \mathbf{in}\}$; $und(f) = \{x \in \text{dom } f | f(x) = \mathbf{und}\}$; and $out(f) = \{x \in \text{dom } f | f(x) = \mathbf{out}\}$. We say that an argument x is *illegally labelled* **in** by f, if $x^- \nsubseteq out(f)$; x is *illegally labelled* **out** by f, if $x^- \cap in(f) = \varnothing$; and x is *illegally labelled* **und** by f, if either $x^- \subseteq out(f)$ or $x^- \cap in(f) \neq \varnothing$. A labelling function is *legal* if it does not have any arguments illegally labelled. Legal labelling functions and extensions can be defined in terms of each other. Any complete extension E can be recovered from a legal labelling function f by setting $E = in(f)$. Conversely, a labelling function f can be defined from an extension E by setting $in(f) = E$; $out(f) = E^+$; and $und(f) = A \backslash (E \cup E^+)$ [6, 7, 17].

In implementations that compute complete extensions, one usually works with labelling functions, henceforth just *labellings*, instead of extensions, since sometimes we need to check the statuses of all attackers of an argument and obtaining these from the extension being constructed is inefficient. Since there are 3 possible labels, we need at least two bits of information to represent the label of each argument, but in most architectures the smallest data type occupies one full byte. A straighforward implementation of a

labelling representation assigns a unique index to every argument and records the labelling as a vector of single bytes, e.g., using the data type `unsigned char` in `C++`.

We now turn to a few graph theoretical concepts which will also prove useful in the discussion of the representations.

Definition 2.4 (Path equivalence relation). Let $\langle A, R\rangle$ be an AF. The path equivalence relation $\sim \subseteq A \times A$ is the relation defined as follows: *i)* For every $x \in A$, $x \sim x$; and *ii)* For every $x, y \in A$, if $x \neq y$ and there is a path from x to y (via $\longrightarrow$) and a path from y to x in R, then $x \sim y$.

Notice that $\sim$ is indeed an equivalence relation, as it is reflexive, symmetric and transitive. $\sim$ induces a partition of the elements of A, dividing it into equivalence classes of significant importance:

Definition 2.5 (Strongly connected component (SCC)). Let $\mathcal{B} = \langle A, R\rangle$ be an AF. A *strongly connected component* of $\mathcal{B}$ is an element of the partition of $\mathcal{B}$ under the path equivalence relation $\sim$. We will denote the set of SCCs of $\mathcal{B}$ by $sccs(\mathcal{B})$.

An SCC S is called *trivial* if and only if it is a singleton set $\{x\}$, for some $x \in A$, and x does not self-attack. All other SCCs are said to be *non-trivial*.

Figure 2 (L) shows AF $\mathcal{A}$'s SCCs, where the trivial SCCs are enclosed by dotted lines, and the non-trivial SCCs are enclosed by dashed lines. It should be easy to see that every argument belongs to a unique SCC (its equivalence class under the path equivalence relation). It will prove useful to have a symbol for this SCC.

Definition 2.6 (SCC of an argument). Given an AF $\langle A, R\rangle$, the SCC to which the argument $x \in A$ belongs will be denoted by the symbol $[\overset{\twoheadrightarrow}{x}]$. Formally, $[\overset{\twoheadrightarrow}{x}] = \{y \in A \mid y \sim x\}$.

Since all elements of a partition are pairwise disjoint, given an AF $\mathcal{B}$, we can construct a directed *acyclic* graph from its attack relation and $sccs(\mathcal{B})$, called $\mathcal{B}$'s *condensation graph*.

Definition 2.7 (Condensation graph of an AF). Let $\mathcal{B} = \langle A, R\rangle$ be an AF and $X, Y \in sccs(\mathcal{B})$. Define the relation $\twoheadrightarrow \subseteq sccs(\mathcal{B})^2$ as $X \twoheadrightarrow Y$ if and only if $X \neq Y$ and there exist $x \in X$ and $y \in Y$ such that $(x, y) \in R$.

Figure 2 (L) shows $\mathcal{A}$'s condensation graph. If $X \in sccs(\mathcal{B})$, then we will use the notation $[X]^-$ to denote the set $\{Y \in sccs(\mathcal{B}) \mid Y \twoheadrightarrow X\}$. Note that X^- is a set of arguments, whereas $[X]^-$ is a set of SCCs.

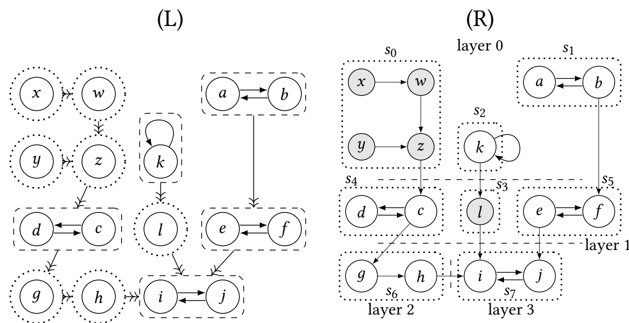

Figure 2: The condensation graph of the AF $\mathcal{A}$ of Figure 1 and its stratification into layers.

Definition 2.8 (Levels of SCCs and arguments). The term $level(S)$ denotes the *level of an SCC S*, which is defined as follows.

<u>For a non-trivial SCC S</u>: If $[S]^- = \emptyset$, then $level(S) = 0$. Otherwise, $level(S) = \max_{X \in [S]^-}\{level(X)\} + 1$.

<u>For a trivial SCC S</u>: If $[S]^- = \emptyset$, then $level(S) = 0$. Otherwise, let $NTA(S) = \{Y \in [S]^- \mid Y \text{ is a non-trivial SCC }\}$ and $TA(S) = \{Z \in [S]^- \mid Z \text{ is a trivial SCC }\}$, then

$$level(S) = \max_{Y \in NTA(S); Z \in TA(S)}\{level(Y) + 1, level(Z)\}.$$

The *level of an argument x* is the level of the SCC to which it belongs. We denote the level of an argument x and the level of an SCC S using the same symbol.

Intuitively speaking, Definition 2.8 arranges trivial and non-trivial SCCs into layers identifying all non-trivial SCCs that can be processed simultaneously, provided the SCCs of previous levels have already been processed. You can think of each non-trivial SCC as a "block". We can also group all trivial SCCs of a layer into a single block which can also be processed following the topological order.

Definition 2.9 (Stratification of an AF into layers). Let $\mathcal{B} = \langle A, R\rangle$ be an AF and $sccs(\mathcal{B})$ its set of strongly connected components. A layer of $\mathcal{B}$ is constituted by the collection of trivial and non-trivial SCCs assigned the same level according to Definition 2.8.

Figure 2 (R) gives the levels of each SCC in $\mathcal{A}$, and also shows $\mathcal{A}$'s stratification into layers. It should be easy to see that the AF has been divided into the 8 blocks $s_0, s_1, \ldots, s_7$.

The significance of the stratification is that, provided layers are processed in topological order, the solutions for each block within a layer can be computed independently and are conditioned by the labellings of the predecessors of the block. Given a fixed assignment of labels to the nodes in previous layers attacking the nodes of a given block, the trivial SCC block will produce a single label assignment for the block, whereas a non-trivial SCC will produce one or more. We will sometimes call these blocks *segments* of the AF. Notice that each label assignment for a given block is also a *segment* of a complete labelling for the entire AF.

Further details of the process of computation of extensions in this way are beyond the scope of this paper, but are discussed in [5, 10]. A new algorithm for computation of extensions taking advantage of the decomposition of AFs into SCCs was proposed in [13], and subsequently the algorithm was implemented in the solver EqArgSolver [12]. [1, 3] made further independent advances for *dynamic* argumentation frameworks.

3 CANONICAL ORDERING OF LABELLINGS

In the previous section, we saw that an AF can be decomposed into an acyclic condensation graph and that we can partition the components of the condensation graph into layers, which can be further divided into self-contained independent blocks.

One of the contributions of this paper is to stress that labellings can benefit from the inherent topological structure of the AF themselves because entire segments repeat themselves multiple times across complete labellings. Understanding how this happens allows not only for benefits in the computation and representation of large

labellings, but also paves the way for potential analyses of the relationships between the blocks. In order to do this systematically, we propose to arrange labellings in a canonical order satisfying the following conditions.

Definition 3.1 (Canonical ordering of arguments). Let $\mathcal{B}=\langle A, T\rangle$ be an AF, and $O = [o_1, o_2, \ldots, o_n]$ a topological ordering of the arguments in A, such that for all i, $o_i = x$ for some $x \in A$, and $o_i = x$ and $o_j = x$ imply $i = j$. Let $O(x) = i$, be the unique index in O, such that $o_i = x$.

For all $x, y \in A$:

(C1) $level(x) < level(y)$ implies $O(x) < O(y)$

(C2) if $level(x) = level(y)$ and $[\overrightarrow{\overrightarrow{x}}] \neq [\overrightarrow{\overrightarrow{y}}]$, then for all $w \in [\overrightarrow{\overrightarrow{x}}]$, either $O(w) < O(z)$, for all $z \in [\overrightarrow{\overrightarrow{y}}]$ or for all $z \in [\overrightarrow{\overrightarrow{y}}]$, $O(z) < O(w)$, for all $w \in [\overrightarrow{\overrightarrow{x}}]$.

(C3) if $level(x) = level(y)$, x belongs to a trivial SCC, and y belongs to a non-trivial SCC, then $O(x) < O(y)$

Notice that Definition 3.1 does not require any specific ordering between elements of the same SCC, between SCCs in the same layer, or between elements of a trivial SCC block of the same layer. (C1) requires the canonical order to respect the topological order. The motivation for this is that solutions for predecessors are needed before the label of an argument can be decided, so it makes sense to present these first. (C2) requires elements of the same block to be grouped together (so we can take advantage of repetitions of solution segments). (C3) is not strictly required, but helps to arrange the solutions in such a way that the segments for the trivial block arguments are always presented first. (C1)–(C3) allow for flexibility. For example, in the arrangement of solutions to the AF $\mathcal{A}$ of Figure 1, we can choose to present the elements of block s_0 in any order, or choose to list the elements of block s_2 before the elements of block s_1, but we are not allowed to interleave the elements of different blocks, and the elements of lower levels must be presented before the elements of higher levels. One potential advantage of respecting the topological ordering is that we can save space in applications where not all layers of the AF need to be considered.

Example 3.2. Consider the arguments of the AF of Figure 1 again. The ordering $O = [x, w, y, z, a, b, k, l, c, d, e, f, g, h, i, j]$, where $O(x) = 1, O(y) = 2, \ldots$ satisfies conditions (C1)–(C3). Notice that $\mathcal{A}$'s segments are ordered in the sequence $[s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7]$ according to O. Given segment solutions $sl_0, sl_1, \ldots, sl_7$ for each s_i, a complete labelling for $\mathcal{A}$ can be obtained by simple concatenation.

As we shall see, if we arrange the labels given to arguments in pre-defined blocks, we will notice that within the set of all labellings, certain sequences of labels repeat themselves (in proportion to the level in which the block appears within the AF). This means we may represent a collection of labellings by storing its unique segment variations only once, as long as we keep track of the combinations of segment variations making up the original labellings.

Let us illustrate the idea with an example. Table 1 lists all complete labellings of the AF $\mathcal{A}$ of Figure 1, using **i** for **in**, **o** for **out**, and **u** for **und**, and one possible canonical arrangement of the arguments. It should be easy to see the repetitions in the labellings. Segment s_0 contains one unique solution that is repeated across all 21 complete labellings. Similarly, there are only three possible

Level	0							1					2		3	
Lab/Arg	x	w	y	z	a	b	k	l	c	d	e	f	g	h	i	j
l_1	i	o	i	o	u	u	u	u	u	u	u	u	u	u	u	u
l_2	i	o	i	o	u	u	u	u	u	u	i	o	u	u	u	o
l_3	i	o	i	o	u	u	u	u	i	o	i	o	o	i	u	o
l_4	i	o	i	o	u	u	u	u	i	o	u	u	o	i	u	u
l_5	i	o	i	o	u	u	u	u	o	i	i	o	i	o	u	o
l_6	i	o	i	o	u	u	u	u	o	i	u	u	i	o	u	u
l_7	i	o	i	o	i	o	u	u	u	u	u	u	u	u	u	u
l_8	i	o	i	o	i	o	u	u	u	u	o	i	u	u	u	u
l_9	i	o	i	o	i	o	u	u	u	u	i	o	u	u	u	o
l_{10}	i	o	i	o	i	o	u	u	u	u	o	i	u	u	o	i
l_{11}	i	o	i	o	i	o	u	u	i	o	u	u	o	i	u	u
l_{12}	i	o	i	o	i	o	u	u	i	o	o	i	o	i	u	u
l_{13}	i	o	i	o	i	o	u	u	i	o	o	i	o	i	o	i
l_{14}	i	o	i	o	i	o	u	u	i	o	i	o	o	i	u	o
l_{15}	i	o	i	o	i	o	u	u	o	i	u	u	i	o	u	u
l_{16}	i	o	i	o	i	o	u	u	o	i	o	i	i	o	u	u
l_{17}	i	o	i	o	i	o	u	u	o	i	o	i	i	o	o	i
l_{18}	i	o	i	o	i	o	u	u	o	i	i	o	i	o	u	o
l_{19}	i	o	i	o	o	i	u	u	u	u	i	o	u	u	u	o
l_{20}	i	o	i	o	o	i	u	u	o	i	i	o	i	o	u	o
l_{21}	i	o	i	o	o	i	u	u	i	o	i	o	o	i	u	o
Block	s_0				s_1		s_2	s_3	s_4		s_5		s_6		s_7	

Table 1: All complete labellings of the AF $\mathcal{A}$ of Figure 1.

s_0	x	w	y	z
sl_{00}	i	o	i	o

s_2	k
sl_{20}	u

s_3	l
sl_{30}	u

s_1	a	b
sl_{10}	u	u
sl_{11}	i	o
sl_{12}	o	i

s_4	c	d
sl_{40}	u	u
sl_{41}	i	o
sl_{42}	o	i

s_5	e	f
sl_{50}	u	u
sl_{51}	i	o
sl_{52}	o	i

s_6	g	h
sl_{60}	u	u
sl_{61}	i	o
sl_{62}	o	i

s_7	i	j
sl_{70}	u	u
sl_{70}	u	o
sl_{70}	o	i

Figure 3: Unique segment solution variations for $s_0, \ldots, s_7$.

variations in solutions to segment s_1, and so forth. Incidentally, this segmentation makes it easier to answer questions about mode of acceptance. For example, the argument x is *skeptically* accepted, since its segment contains only one solution, x is labelled **in** in that solution, and hence all solutions will have the same label for x. We can also easily see that a is accepted in the "majority" of solutions, whereas j has the lowest level of acceptance of all *credulously* accepted arguments (1/7 of the solutions).

In terms of representation, in reality all we need to store is the information in the tables of Figure 3. We can then represent any complete labelling by keeping track of the desired segment solution *order*. For example, l_{21} can be represented as the sequence of segment labellings $[sl_{00}, sl_{12}, sl_{20}, sl_{30}, sl_{41}, sl_{51}, sl_{62}, sl_{70}]$. Despite the potential benefits in the analyses of the interdependencies between segment solutions, there is a non-insignificant overhead in keeping track of the sequences themselves. However, large problems can have millions of labellings, involving a lot of repetitions. For example, the problem `massachusetts_srta_2014-11-13.gml.50` (see idx 41 in Table 3) has 34307712 unique complete labellings and benefits greatly from segmented storage. We discuss the relative costs of various representations in Section 6.

4 REPRESENTATIONS USING SEQUENCE CONTAINERS

In this section we consider two alternative representations using sequences of containers. The first one, which we use as the baseline, is a *continuous* representation in which for an AF $\langle A, T\rangle$, we represent each labelling with a sequence container with $n = |A|$ cells, each large enough to hold an individual label, and a further container to hold all labelling containers.

Let c_v be the number of bytes required to store the infrastructure of a sequence container in the local computer architecture and c_l the number of bytes required to store one label.

In our experiments, we used the `vector` C++ container in a 64-bit testing machine. In this case, $c_v = 24$ (16 bytes for two pointers to the beginning and end of the vector, plus 8 bytes to store the capacity allocated to the vector), and $c_l = 1$ byte (`unsigned char`, the smallest native data type we can use to represent a label).[1]

Figure 4 illustrates how this representation is used.

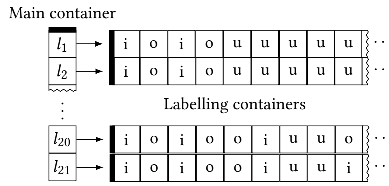

Figure 4: Using continuous sequence containers.

Memory requirement of the representation using continuous sequence containers

Assuming that each labelling has n arguments, and each label needs c_l bytes of memory, a solution will require the following amount of bytes:

$$c_{sol}(n) = c_v + n * c_l \tag{1}$$

Let t be number of unique labellings. The cost in bytes to represent t labellings of size n will be

$$\begin{aligned} c_{allsols}(t, n) &= c_v + (t * c_{sol}(n)) \\ &= c_v + t * (c_v + n * c_l) \end{aligned} \tag{2}$$

Example 4.1 shows the calculation for graph 19 (`BA_60_80_1`), which was part of the "A/3" series in ICCMA'17. This graph has $n = 61$ arguments and $t = 2480058$ unique complete labellings.

Example 4.1 (Graph `BA_60_80_1`*).* Each labelling needs to store one label (= 1 byte) per argument. As each labelling is stored as a vector, the cost per labelling is $c_{sol}(61) = 24 + 61 * 1 = 85$. Now to store all $t = 2480058$ labellings, we need

$$c_{allsols}(2480058, 61) = 24 + (2480058 * 85) = 210804954 \text{ bytes.}$$

The continuous sequence representation potentially wastes a lot of space by storing multiple times segments that repeat themselves across labellings. To avoid this, we can store only the unique segment solutions, and then re-construct the labellings by pointing to the right vector for the right segment in the right order.[2]

[1] Although we can actually pack 4 labels in one byte, we cannot address it directly, unless we use a special technique. This is explored in Section 5.

[2] The solver `EqArgSolver` basically uses this representation, except that smart pointers are used for some of the book-keeping. Although smart points are access-efficient, their use can be costly, because of their intrinsic memory requirements.

Memory requirement of the representation using segmented sequence containers

To start here, we need to see that each segment is a "mini-AF", i.e., the labellings for a segment s_i will need as many bytes as the labellings of an AF with as many arguments as the segment, so we can use (1) and (2) again, where n is replaced by $|A(s_i)|$ — the number of arguments in the segment s_i.

Now let $u(s_i)$ be the number of unique labellings of the segment s_i, then all such labellings can be stored in a vector using $c_{oneseg}(s_i) = c_{allsols}(u(s_i), |s_i|)$ bytes. So for example, for s_4, we need a vector with three vectors, one for each solution sl_{40}, sl_{41} and sl_{42} to the segment.

Let *segs* be the set of all segments. To store all segments $s_i \in segs$, we will need a vector with one position for each segment, where each position stores the vector of vectors above. When using this to "assemble" a complete labelling, we need to address the correct vector for the segment and solution. Equation (3) below gives the cost of storing all segment solutions:

$$sc_{allsegs} = c_v + \textstyle\sum_{s_i \in segs} c_{oneseg}(s_i) \tag{3}$$

However, we still need a way to "assemble" the solutions to the AF as a whole. Each solution can be re-constructed via the sequence of *indices* indicating the appropriate solution for each segment. This can be represented as a vector of an appropriate unsigned integer type, large enough to address all segment solutions.

So our next problem is to determine the maximum number of solutions for any given segment. Out of the solutions we analysed in ICCMA'17,[3] the largest number of unique solutions of any single segment was 826686, again in problem 19 (`BA_60_80_1`) chosen as our running example for this reason. This means that we need at least 24 bits (3 bytes) to address a particular solution for each segment. In C++ the shortest integer containing at least 24 bits is the 4-byte long `unsigned int` (32 bits).

Our conclusion is that each solution using a segmented representation can be safely represented as a vector of elements of the type `unsigned int`. For a solution with n segments, this will cost $c_{onesegsol}(n) = c_v + n * 4$ bytes. In order to store *all* solutions, we will need another vector containing all such vectors, giving the following cost of assembling all solutions:

$$c_{allsegsols} = c_v + (t * c_{onesegsol}(n)) \tag{4}$$

The total cost will be the sum of (3) and (4): $c_{allsegsols} + sc_{allsegs}$.

Example 4.2 (Graph `BA_60_80_1`*).* The graph can be partitioned into five segments whose details are given in Table 2. "uls" stands for the number of unique labellings for a segment.

Now contrast Example 4.2 with Example 4.1. The segmented sequence representation requires only 79.61% of the space requirement of the continuous sequence representation for this graph.

5 USING INTEGERS OF ARBITRARY PRECISION

In Section 4, we saw that the segmented sequence representation is a lot more efficient in a number of cases (especially when the number of unique segment solutions is relatively small). The crucial factor

[3] The hard instances identified in [14].

seg id	len	uls		$c_{oneseg}(id)$
0	7	1	$24 + 1 * (24 + 7 * 1) =$	55
1	4	3	$24 + 3 * (24 + 4 * 1) =$	108
2	1	2	$24 + 2 * (24 + 1 * 1) =$	74
3	47	826686	$24 + 826686 * (24 + 47 * 1) =$	58694730
4	2	2	$24 + 2 * (24 + 2 * 1) =$	76
Total				**58695043**
(A) $c_{allsegsols}$			$24 + 58695043 =$	58695067
$c_{onesegsols}$			$24 + (5 * 4) =$	44
(B) $c_{allsegsols}$			$24 + 2480058 * 44 =$	109122576
Total (A+B)				**167817643**

Table 2: Segmented sequence representation of graph 19.

with the representation is how to efficiently address the solutions for the segments. In some cases, we saw that the addressing requires at least 24 bits, but in practice 32 bits are needed because of the limitations in the size of the basic data types at our disposal. The addressing issue is potentially problematic because we may end up spending more space storing the addresses of the segment solutions than the whole AF solutions themselves.[4]

Tackling both the storage of the addresses and the storage of the unique labellings themselves can be achieved if we use a representation in which the addresses can be embedded in the data structure itself. The basic idea is to represent labellings as long sequences of bits, packing as many labels as possible into the sequence. In "traditional" AFs, we only have the labels **in**, **out** and **und**, and therefore we only need 2 bits per label. Therefore we can pack 4 labels into a single byte, potentially saving of up to 75% of space. We say potentially because we need some extra control information to manage the allocation of space for the long sequences of bits.

Fortunately, there are ready-made solutions to do all the management efficiently and we chose as a case study, GNU's Multiple Precision Arithmetic Library (GMP, https://gmplib.org). For space reasons, we cannot go into the details of its implementation, but for the purposes of the presentation below, it suffices to know that in GMP integers of arbitrary precision are stored as sequences of so-called *limbs*, which are usually the length of a word in the underlying system architecture. In a 64-bit machine, a limb is 64 bits (8 bytes) long. This means that memory is allocated in multiples of 8 bytes and a little overhead is also needed to keep track of the allocation of limbs. For all purposes, we can think of an integer as an aribitrarily long sequence of bits, which we can access independently and hence use as a proxy for labellings, employing two bits per argument. More details about these are discussed next.

Memory cost of the integer representation

As we mentioned, each solution can be encoded as an integer of arbitrary precision and to represent $|A|$ labels, we need $2 * |A|$ bits. We can record the label for a particular argument by setting the appropriate bits in the (long) integer and retrieve the labels of individual argument efficiently by performing bitwise "and" operations with an appropriate mask. The details are beyond the scope of this paper, but the idea was first demonstrated in [14]. In our test machine, we found that each integer object takes 16 bytes, plus the limb allocation, which of course depends on the size of the solution.

With all this in mind, each labelling for a graph with n arguments can be stored as an arbitrary precision integer object requiring the following number of bytes: $c_{mpzsol}(n) = 16 + \lceil (n * 2)/64 \rceil * 8$. We can then use a vector to store all such objects, and define the total cost to represent t labellings of an AF with n arguments as $c_{allmpzsols}(t, n) = 24 + (t * c_{mpzsol}(n))$.

Example 5.1 (Graph `BA_60_80_1`*).* Recall that this graph has 2480058 unique labellings and $n = 61$ arguments. Each labelling will need at least 122 bits, i.e., two 8-byte segments. The cost per labelling is $c_{mpzsol}(61) = 16 + \lceil (2 * 61)/64 \rceil * 8 = 32$ bytes. Now to store all $t = 2480058$ labellings, we need $c_{allmpzsols}(61, 2480058) = 24 + (2480058 * 32) = 79361880$ bytes.

Now contrast Examples 4.1, 4.2 and 5.1. The segmented sequence representation requires 79.61% of the space requirement of the continuous sequence representation for this graph, whereas the integer representation requires only 37.65% of this space.

6 EXPERIMENTAL RESULTS

For comparative evaluation we used the 42 AFs identified in [14], which are amongst the most memory intensive problems in all ICCMA instances. For each graph, we generated all complete labellings and verified the results from at least two solvers. We then decomposed each graph into SCCs, grouped the SCCs into the segments and canonical order described in Section 3, counted the number of unique solutions for each segment, and computed the memory requirements of each representation. A summary of the results can be found in Table 3. The contents of the first four columns are self-descriptive. The column "max" gives the maximum number of unique solution in any segment of the graph; the columns "con", "seg" and "int" give the memory requirements (in MBs) of each of the continuous, segmented and integer representations, respectively. The columns "seg/con" and "int/con" give the ratio between the segmented and integer representations, respectively, with respect to the continuous one. Here a value below 1 indicates saving.

The segmented representation was strictly more economical than the continuous one in 32/42 problems (76%) and required roughly the same in two other problems. In some of the 32 cases, it was signficantly better than the continuous representation, reducing the space requirements by as much as 47.84% (problem 39, `BA_60_90_4`). In problem 41 (`massachusetts_srta_2014-11-13.gml.50`), with the largest number of complete labellings (34307712), the segmented representation saved over 1.5 billion bytes (> 1439 MB). Taking into account that the continuous representation needed nearly 4000MB, the difference may be significant enough to cause difficulties to some solvers due to memory limitations or because of the time it takes to manage the memory itself.

In problem 6 (`sembuster_60`), the segmented representation usage was over twice as high as the continuous one. The structure of graphs in the "Sembuster" domain contains a large number of layers with very small SCCs and hence it is problematic for the segmented version which needs to address all segments to compose each labelling. In any case, graphs in this domain can only be solved for a

[4]In our analysis this only happened in 19% of the cases and in all but one of the cases the effect was not significant. Table 3 has full details.

small number of arguments, because of the combinatorial nature of the solutions they generate (see [15] for a full discussion).

The memory requirements of the integer representation were uniformly better than the other two representations', ranging from 25% to 51% of the requirements of the continuous representation. Although we did not implement this representation fully, we did implement dummy containers, allocated space to represent solutions in all graphs, and recorded the actual memory usage of those containers in all problems. The results are given in Table 3. Further work is required to measure the performance of the representation in terms of reading and setting of labels, but these initial results are very promising, given that the GMP library has been used before to efficiently compare large solutions of enumeration problems [14].

The 2019 version of the solver `EqArgSolver` [12] used a rudimentary implementation of the segmented representation presented in this paper and the solver was adapted to report on the segments and their number of unique solutions. In the complete extensions enumeration track of ICCMA'19 (http://argumentationcompetition.org/2019/results/results-main.html), the solver μ-`toksia` [11] came out in first place, and the solvers `pyglaf` [4] and `EqArgSolver` came up in third and fifth places, respectively. During our analysis, we benchmarked the performance of the solvers in the 42 graphs, giving each of them 30 mins to complete each enumeration of the solutions. `EqArgSolver` managed to compute all solutions correctly, taking no longer than 101 secs in any graph. `pyglaf` took considerably longer in most cases, being faster than `EqArgSolver` only in two instances. Perhaps somewhat surprisingly, μ-`toksia` struggled, being unable to complete 25 instances. In the case of `massachusetts_srta_2014-11-13.gml.50` (the problem with the largest number of solutions), neither `pyglaf` nor μ-`toksia` managed to complete in under 30 mins. The execution times of all solvers in the 42 problems can be found in Table 3.

These results show that performance results can vary greatly, depending on how much time a solver is given to solve a problem and the types of graphs given to it. We speculate that memory management may be causing difficulties to `pyglaf` and μ-toksia in problems with very large solutions.

7 CONCLUSIONS AND FUTURE WORK

In this paper we proposed a canonical ordering of arguments of AFs, which can be used not only in the optimisation of the representation of argument labellings but also in further analyses of properties of the labellings themselves and towards a standard presentation of extensions. Using our canonical ordering, we presented two novel internal representations of labellings. We compared the memory requirements of the new representations with a baseline representation in 42 hard enumeration problems from ICCMA'17, and they were found to offer substantial advantages in problems generating large sets of solutions.

We found that for a large proportion of the graphs analysed (76%), the segmented representation offered savings of up to 47.84% in comparison with the baseline approach. Finally, the representation using integers of arbitrary precision proved the most economical with savings of up to 75% in memory requirements. Taking into account that some problems will inevitably generate large solutions, the analysis shows that our proposals go some way towards supporting the construction of more resilient solvers.

The segmented representation of labellings also opens interesting directions for the investigation of hidden dependencies between AF components and can be used in the solution of some well known problems in argumentation, such as in the decision problems of argument acceptance. Here skeptical and credulous considerations can be performed more directly – and sometimes more efficiently – by simply analysing the unique segment variations.

Further optimisations to the segmented sequence representation are possible. Firstly, instead of storing segment solutions as independent vectors, we can simply store them in a single continuous vector and then compose a complete labelling by making appropriate references to a specific offset within the solutions of the segment. Secondly, we can of course also pack the segment solution representation using the integer representation, reducing the memory requirements even further.

Finally, we found some interesting performance variations in the solvers used in our benchmarks: the solver `EqArgSolver`, which does not rely on a translation of the AF into a problem to be solved by another solver; and the solvers `pyglaf` (circumscription+`GLUCOSE`) and μ-`toksia` (`CryptoMiniSat`). The use of a "pure" solver allowed us to fine-tune all details of the implementation, including memory management using a variation of one of the newly proposed representations. In our analysis, `EqArgSolver` outperformed both `pyglaf` and μ-`toksia` significantly in 40/42 of the problems investigated.

REFERENCES

[1] G. Alfano and S. Greco. 2021. Incremental Skeptical Preferred Acceptance in Dynamic Argumentation Frameworks. *IEEE Intelligent Systems* 36, 2 (2021), 6–12. https://doi.org/10.1109/MIS.2021.3050521

[2] G. Alfano, S. Greco, and F. Parisi. 2019. On Scaling the Enumeration of the Preferred Extensions of Abstract Argumentation Frameworks. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus). ACM, New York, NY, USA, 1147—-1153. https://doi.org/10.1145/3297280.3297393

[3] G. Alfano, S. Greco, and F. Parisi. 2021. Incremental Computation in Dynamic Argumentation Frameworks. *IEEE Intelligent Systems* 36, 6 (2021), 80–86. https://doi.org/10.1109/MIS.2021.3077292

[4] M. Alviano. 2018. The Pyglaf Argumentation Reasoner. In *Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017) (OpenAccess Series in Informatics (OASIcs), Vol. 58)*, R. Rocha, T. C. Son, C. Mears, and N. Saeedloei (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:3. https://doi.org/10.4230/OASIcs.ICLP.2017.2

[5] P. Baroni, M. Giacomin, and G. Guida. 2005. SCC-recursiveness: a general schema for argumentation semantics. *Artificial Intelligence* 168, 1 (2005), 162–210. https://doi.org/10.1016/j.artint.2005.05.006

[6] M. Caminada. 2011. A labelling approach for ideal and stage semantics. *Argument and Computation* 2, 1 (2011), 1–21.

[7] M. Caminada and D. M. Gabbay. 2009. A Logical Account of Formal Argumentation. *Studia Logica* 93, 2-3 (2009), 109–145.

[8] 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.

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

[10] B. Liao. 2013. Toward incremental computation of argumentation semantics: A decomposition-based approach. *Annals of Mathematics and Artificial Intelligence* 67, 3 (2013), 319–358. https://doi.org/10.1007/s10472-013-9364-8

[11] A. Niskanen and M. Järvisalo. 2020. μ-toksia: An Efficient Abstract Argumentation Reasoner. In *Proceedings of the 17th Int. Conference on Principles of Knowledge Representation and Reasoning.* 800–804. https://doi.org/10.24963/kr.2020/82

[12] O. Rodrigues. 2018. EqArgSolver – System Description. In *Theory and Applications of Formal Argumentation*, E. Black, S. Modgil, and N. Oren (Eds.). Springer International Publishing, Cham, 150–158. https://doi.org/10.1007/978-3-319-75553-3_11

[13] O. Rodrigues. 2018. A Forward Propagation Algorithm for the Computation of the Semantics of Argumentation Frameworks. In *Theory and Applications of Formal Argumentation.* Springer International Publishing, Cham, 120–136. https://doi.org/10.1007/978-3-319-75553-3_8

idx	graph name	# sols	# segs	max	con	seg	int	seg/con	int/con	EqArg	pyglaf	μ-toksia
0	admbuster_2000000	1	1	1	1.91	1.91	**0.48**	1.0000	0.2500	18.11	53.95	**2.53**
1	BA_160_20_3	91854	30	12	16.21	12.62	**5.61**	0.7786	0.3459	**0.93**	1.12	20.45
2	ferry2.pfile-L2-C3-01.pddl.1.cnf	151146	48	36	30.85	31.14	**9.23**	1.0095	0.2991	3.15	**2.23**	32.59
3	BA_80_50_1	406782	14	54	40.73	31.04	**15.52**	0.7620	0.3810	**1.51**	7.31	237.87
4	kanawha-valley-regional-transportation-authority-krt_20150126_1340.gml.20	373248	19	12	45.56	35.6	**17.09**	0.7813	0.3750	**1.09**	6.96	155.01
5	BA_60_60_3	927261	13	126	75.17	67.21	**28.30**	0.8942	0.3765	**2.60**	30.75	1253.92
6	sembuster_60	1048596	40	3	84.00	184.01	**32.00**	2.1905	0.3810	**14.01**	130.41	TO
7	BA_160_30_2	505197	29	9	89.13	67.45	**30.83**	0.7568	0.3459	**3.26**	13.52	438.11
8	BA_140_30_4	570807	25	8	89.82	67.5	**30.48**	0.7515	0.3394	**3.69**	16.07	466.51
9	ferry2.pfile-L2-C1-04.pddl.4.cnf	540207	58	32	102.52	131.89	**32.97**	1.2865	0.3216	20.13	**15.04**	171.46
10	thecomet_20131025_1906.gml.20	612360	32	6	112.13	88.77	**37.38**	0.7917	0.3333	**4.41**	17.25	621.95
11	go-taps_20150524_1019.gml.20	708588	32	12	125.02	102.72	**43.25**	0.8216	0.3459	**8.64**	21.03	753.54
12	bw2.pfile-3-02.pddl.2.cnf	743510	38	45	131.18	124.8	**45.38**	0.9514	0.3459	**12.61**	35.21	932.00
13	bw2.pfile-3-08.pddl.2.cnf	743510	38	45	131.18	124.8	**45.38**	0.9514	0.3459	**12.57**	29.03	994.47
14	commute.org_20140813_1738.gml.20	991440	22	18	132.37	105.9	**45.38**	0.8000	0.3429	**4.25**	46.44	1508.26
15	BA_100_40_5	1220346	22	36	145.48	130.35	**55.86**	0.8960	0.3840	**6.95**	54.91	TO
16	BA_120_50_4	1062882	16	243	146.98	89.22	**48.65**	0.6070	0.3310	**3.89**	45.01	TO
17	BA_140_30_3	1062882	29	3	167.25	141.91	**56.76**	0.8485	0.3394	**5.72**	42.00	TO
18	irvine-shuttle_20091229_1547.gml.80	3000618	12	23	191.73	206.04	**91.57**	1.0746	0.4776	**15.76**	468.36	TO
19	BA_60_80_1	2480058	5	**826686**	201.04	160.04	**75.69**	0.7961	0.3765	**11.87**	209.15	TO
20	south-metro-area-regional-transit_20151216_1704.gml.20	1071630	25	378	205.42	126.75	**65.41**	0.6170	0.3184	**2.90**	54.21	1725.85
21	BA_180_30_4	1062882	22	18	207.80	113.53	**64.87**	0.5464	0.3122	**2.48**	45.95	TO
22	BA_180_30_5	1062882	30	10	207.80	145.97	**64.87**	0.7025	0.3122	**7.72**	46.25	TO
23	el-dorado-transit_20151217_1024.gml.20	1497852	20	138	232.84	148.57	**79.99**	0.6381	0.3436	**13.85**	170.76	TO
24	yamhill-or-us.gml.20	1358127	22	324	234.43	145.09	**72.53**	0.6189	0.3094	**23.22**	61.96	TO
25	ferry2.pfile-L3-C1-07.pddl.1.cnf	1414107	46	20	253.54	280.51	**86.31**	1.1064	0.3404	**32.99**	101.93	TO
26	fresno-county-rural-transit-agency_20151216_1934.gml.50	2716254	20	26	261.63	269.41	**103.62**	1.0297	0.3960	**13.98**	353.47	TO
27	ferry2.pfile-L2-C1-04.pddl.6.cnf	1192716	39	32	276.40	204.75	**81.90**	0.7408	0.2963	**22.45**	50.67	1296.97
28	ferry2.pfile-L2-C1-09.pddl.6.cnf	1192716	39	32	276.40	204.75	**81.90**	0.7408	0.2963	**22.67**	52.73	1254.30
29	BA_80_70_5	2918202	11	98	292.22	189.25	**111.32**	0.6476	0.3810	**11.02**	251.53	TO
30	BA_180_30_2	1594323	34	3	311.70	243.28	**97.31**	0.7805	0.3122	**10.85**	97.53	TO
31	massachusetts_nrta_2014-12-03.gml.20	2125764	27	162	312.20	267.61	**113.53**	0.8572	0.3636	**14.68**	156.60	TO
32	ferry2.pfile-L2-C1-03.pddl.1.cnf	2268636	36	64	318.04	363.48	**103.85**	1.1429	0.3265	**47.71**	183.31	TO
33	ferry2.pfile-L2-C1-05.pddl.1.cnf	2268636	36	64	318.04	363.48	**103.85**	1.1429	0.3265	**47.42**	186.93	TO
34	ferry2.pfile-L2-C4-05.pddl.1.cnf	1255338	57	68	329.23	301.7	**95.77**	0.9164	0.2909	**27.18**	68.48	TO
35	carroll-transit-system_20151202_1957.gml.50	5004804	8	129250	381.84	274.81	**152.73**	0.7197	0.4000	**22.57**	912.90	TO
36	aircoach_20130716_1324.gml.50	4587840	17	141	393.78	402.54	**175.01**	1.0222	0.4444	**19.55**	535.18	TO
37	metro-bilbao_20110407_1059.gml.80	8111740	5	1279	487.37	340.43	**247.55**	0.6985	0.5079	**31.02**	1535.28	TO
38	anaheim-resort-transportation_20151217_1213.gml.50	5349240	14	201978	535.65	421.98	**204.06**	0.7878	0.3810	**52.37**	1015.93	TO
39	BA_60_90_4	7788996	5	38088	631.39	329.31	**237.70**	0.5216	0.3765	**7.42**	1599.96	TO
40	bw3.pfile-3-04.pddl.2.cnf	2558258	69	148	1254.03	731.94	**351.32**	0.5837	0.2802	**72.71**	556.14	TO
41	massachusetts_srta_2014-11-13.gml.50	34307712	13	2256	3926.21	2486.74	**1308.74**	0.6334	0.3333	**100.81**	TO	TO

Table 3: Structural and solution information; memory usage (MBs) using continuous sequence, segmented sequence, and integer representations; and execution time (secs) for solvers `EqArgSolver`, μ-`toksia`, and `pyglaf'19`. 'TO' indicates timeout at 1800 secs.

[14] O. Rodrigues. 2019. Representing and Comparing Large Sets of Extensions of Abstract Argumentation Frameworks. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus). ACM, New York, NY, USA, 1154–1161. https://doi.org/10.1145/3297280.3297394

[15] O. Rodrigues, E. Black, M. Luck, and J. Murphy. 2018. On Structural Properties of Argumentation Frameworks: Lessons from ICCMA. In *Proceedings of the 2nd Int. Workshop on Systems and Algorithms for Formal Argumentation.* 22–35.

[16] M. Ulbricht. 2021. On the Maximal Number of Complete Extensions in Abstract Argumentation Frameworks. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning.* 707–711. https://doi.org/10.24963/kr.2021/74

[17] Y. Wu and M. Caminada. 2010. A labelling-based justification status of arguments. *Studies in Logic* 3, 4 (2010), 12–29.