UNIVERSITÉ LIBRE DE BRUXELLES

DÉPARTMENT D'INFORMATIQUE

MEMO-F-403
PREPARATORY WORK FOR THE MASTER THESIS

Data structures for Partially Ordered Sets

Hakim BOULAHYA

Supervised by
Emmanuel FILIOT
Guillermo A. PÉREZ

May 26, 2018

Contents

1 Introduction

1.1 Theoritical context

Data structures play an important role in algorithms complexity. With the objective to improve standard algorithms in automata theory, researchers at ULB have implemented new algorithms to solve important problems therein.

One computer science field that benefits from those new implementation is formal verification of computer systems. Verifiying a system consists in checking for a given model, if it respect some formal specifications. The major issue regarding problems such as model checking is the state-space explostion problem. Verification techiques are based on representing models and specifications using automata, and the combination can lead to an exponential blow-up. To face the state explosion problem, methods that allow to represent larger system using smaller models were developped. A known method is called symbolic model checking and sometimes uses a data structure called Binary Decision Diagram, used to represent boolean functions.

Another well known problem in verification is the synthesis problem. It is, for some model of a controller and an environment, to check if there exists a winning strategy for the controller. In order to develop more efficient algorithms to resolve such problems Filiot et al. proposed in [FJR09] a new approach using antichains. This is possible due to a partial order that exists on the state space of the subset constructions. Since it has been proven that the set is closed for this partial order, antichains are well suited to resolve such problem. Indeed, antichains are a set of incomparable elements, and allow to represent partially ordered set in a more compact way. This is possible through different operations on the antichains. When the set of a partial order is closed, the antichains can represent a bigger set by only keeping either maximal or minimal elements, and other elements can be retrieved using an upper or lower closure, that is all the elements that are either bigger or smaller compared to the elements of the antichains. We define formally those notions in section 2.

The goal of this preparatory work is to motivate the interest that are made in data structures for partially ordered sets, especially antichains, and why we need efficient implementations. It also defines the objectives and recalls the state of the art of the various existing implementations. In this work, we will mainly focus on the usage of partially ordered sets and antichains in automata theory-related problems.

1.2 An application: language universality

An example that highlights the use of antichains in automata theory is the universality problem. It is the problem that for a finite automaton, we want to check if the language of this automaton equivalent to the language of all the words on the alphabet. The universality problem is a classical theoritical problem, and many verification-related problems can be reduced to it.

Let $A = \langle Q_A, \Sigma, q_0, \delta, F_A \rangle$ be a finite automaton, we want to check whether the language of A is universal, that is, $L(A) = \Sigma^*$. The language of A is universal if and only if the language of the complement of A accepts no words, that is, $L(A) = \Sigma^*$ if and only if $L(\bar{A}) = \emptyset$. Therefore, the goal is to find a computation path of the automaton on a word such that the path start in the initial state, and the final target is a non-accepting state.

Let's first define an interesting proprety of non-deterministic automata before explaining the use of antichains for the universality problem. Let q and f be two subsets of states such that $q, f \subseteq Q_A$. When reading a letter σ on the deterministic equivalent A_d, that is $q \xrightarrow{\sigma} f$, it is known that for all sets of states $q' \subseteq q$, the resulting subset f', when computed on A_d such that $q' \xrightarrow{\sigma} f'$, it holds that $f' \subseteq f$.

1.2.1 Concrete example

Let's take the example from [DWDHR06] shown in Figure 1. This example is a non-deterministic finite automata with 4 states where only the last state is non-accepting. The goal is to check if whether or not this automaton is universal. One approach to check universality, is to check if the complementary automaton accepts the empty set. This is actually the standard algorithm that first builds the equivalent deterministric automaton, builds the complement and then check the emptiness of this final construction. The complexity here rely on building the equivalent deterministic which is exponential in size. The idea of the antichain based algorithm is instead of determinize the automaton, to do it implicitly by checking only maximal subsets of states.

Figure 2 shows the equivalent deterministic finite automaton of the one shown in Figure 1. In the specific example of 2, instead on checking each state of the deterministic finite automaton, it is better to check only the maximal path. We refer to maximal path, a path where each subset of states when computing any letter is maximal.

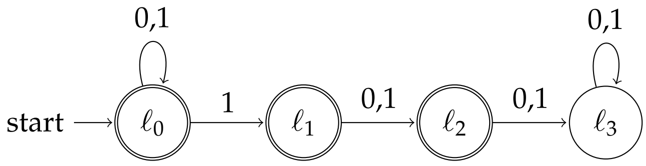

Figure 1: Non-deterministic finite automaton from the family of automata A_k, from [DWDHR06], with $k = 3$.

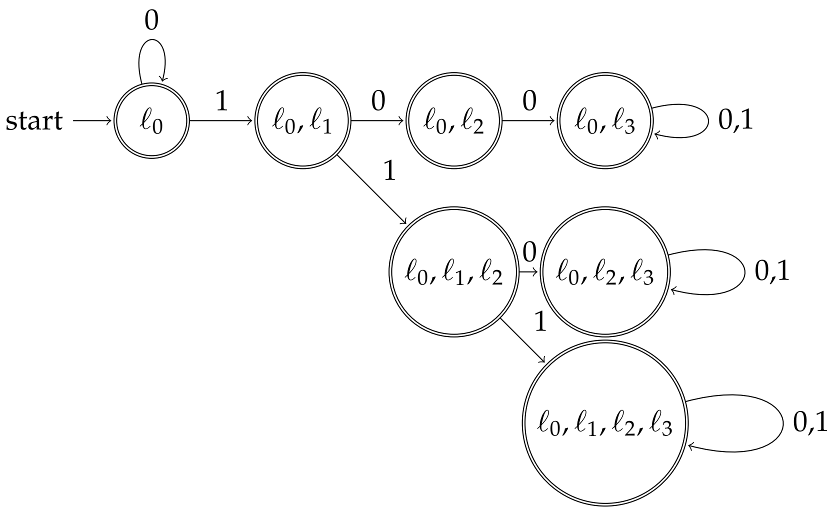

Figure 2: Deterministic automaton equivalent of the non-deterministic automaton shown in Figure 1.

Indeed, the focus is made on checking if the final state of a word computation is non-accepting that is $q \xrightarrow{w} f$ we have that $f \cap F_A = \emptyset$. Because of the proprety of state inclusion, checking only the maximal set of states is sufficient. Let $k = 3$, computing the word 1^k, will lead to a maximal path for this automaton. For any word $w < 1^k$ of size k such that $w = \sigma_1..\sigma_k$ with $\sigma_i \in \Sigma$, the path of computation will be included in the path of 1^k, *i.e.* the subset of states q_i will be included in p_i, that is $q_i \subseteq p_i$, corresponding respectively to the subsets of states when reading the i^{th} letter of the word w and 1^k. Since the objective is to find the final state as a non-accepting, if the final state is non-accepting for 1^k, that is $p_k \cap F = \emptyset$ then for the word w, since $q_k \subseteq p_k$, it holds that $q_k \cap F = \emptyset$. This method allow to bypass some computation, *i.e.* checking the intersection with the accepting states set, by only checking the maximal subsets of states.

The algorithm proposed in [DWDHR06], doesn't build the deterministic automaton, but rather does it implicitly. It builds iteratively a set of subsets of states that have as initial targets the non-accepting states. At each iteration, it adds direct predecessors to the targets, until a fixed point is reached. Also, at each iteration, only maximal set of states, that is an antichain, based on the inclusion partial order are kept in memory for the next iteration. The final condition is to check if the initial state is found in the targets. It the initial state is included, it means that there exists a word such that it is not accepted by the automaton, it means that the automaton is not universal.

1.3 Objective

The objective of the final work is to provide an efficient implementation of different data structures that allow to compactly represent partially ordered sets, specifically antichains. The first step is to implement in Java, classes that will be provided to the `Owl` library [Sal16]. `Owl` is a LTL to deterministic automata translations tool-set written in `Java`. We present in more details the `Owl` library in section 4.2. A second step will be to implement antichain-based algorithms using the new antichains implementation and study the performance.

1.4 Related work

There are two interesting implementations that were found. The first one is `AaPAL` (Antichain and Pseudo-Antichain Library), a generic library that was implemented in the frame of Aaron Bohy's PhD thesis [Boh14b] to provide an antichain library. It is implemented in `C`. The other implementation of antichains have been done by De Causmaecker and De Wannemacker in [DCDW14]. The algorithms to find the ninth Dedekind number uses antichains and they needed to implement a representation of antichains. Their implementation is using `Java`. To improve efficiency and performances, Hoedt in [Hoe15a] has extended [DCDW14] antichains implementation by using bit sequence instead of tree reprensentation.

1.5 Structure of the preparatory work

This paper is the introduction to next year thesis, therefore the content concern only the preliminaries. The goal is to properly define the subject, existing implementations and the desired objectives. In section 2, we formally define antichains, and give examples of such data structures. In section 3, we summarize the work that has been done by others for antichains implementation. In section 4, we propose an overview of next year work and possibilities.

2 Data Structures

In this section, we will provide formal definitions of the data structures that we will implement. We recall the notion of binary relations and important propreties of such relations. We then define partially ordered set, totally order set and closed set. Finally we give a formal definition for antichains.

The definitions and examples for this section are based on [Boh14b] and [Maq11].

2.1 Binary relations

A binary relation for an arbitrary set S is a set of pair $R \subseteq S \times S$. There are five important properties: reflexitivity, transitivity, symmetry, antisymmetry and total.

A relation R on S is said to be:

- Reflexive: iff $\forall s \in S$ it holds that $(s, s) \in R$
- Transitive: iff $\forall s_1, s_2, s_3 \in S$, if $(s_1, s_2) \in R$ and $(s_2, s_3) \in R$ then it holds that $(s_1, s_3) \in R$
- Symmetric: iff $(s_1, s_2) \in R$ then $(s_2, s_1) \in R$.
- Antisymmetric: iff $(s_1, s_2) \in R$ and $(s_2, s_1) \in R$ then $s_1 = s_2$
- Total: iff $\forall s_1, s_2 \in S$ then $(s_1, s_2) \in R$ or $(s_2, s_1) \in R$

Orders A *partial order* is a binary relation that is *reflexive*, *transitive* and *antisymmetric*. We note a partial order relation by R. We note $s_1 R s_2$ to show the belonging of a binary relation to a partial order, which is equivalent to $(s_1, s_2) \in R$. A *total order* is a partial order that is *total*.

Example 2.1.1. For example, the comparison of natural numbers is a partial order. Let $\leq$ be a binary relation on $\mathbb{N}$ such that $\leq \subseteq \mathbb{N}^2$. The binary relation is defined following the usual semantic of the symbol, i.e. $n_1 \leq n_2$ if and only if n_1 is smaller or equal to n_2. Based on this, $\leq$ is a partial order. It is reflexive, transitive and antisymmetric. The binary relation $\leq$ on natural numbers is actually a total order since all the natural numbers can be compared against each other.

2.2 Partially ordered set

An arbitrary set S associated with a partial order $\preceq$ is called a *partially ordered set* or *poset*. It is denoted by the pair $\langle S, \preceq \rangle$.

Comparable Let $s_1, s_2 \in S$ and $\langle S, \preceq \rangle$ a poset. The two elementes s_1 and s_2 are said to be *comparable* if either $s_1 \preceq s_2$ or $s_2 \preceq s_1$. If neither of those two comparisons are correct, then s_1 and s_2 are said to be *incomparable*.

Bounds Let $\langle S, \preceq \rangle$ a partially ordered set. A *lower bound* of $P \subseteq S$ is an element $s \in S$ such that for all $p \in P$, it holds that $s \preceq p$. The *greatest lower bound* of elements of a set $P \subseteq S$ is an element $s \in S$ defineds as follow: for all $p \in P$ it holds that $s \preceq p$, and for all $s' \in S$ we have that if $s' \preceq p$ then $s' \preceq s$.

The greatest lower bound is unique. It means that if two elements $s_1, s_2 \in S$ are $\preceq$-incomparable and for a subset $P \subseteq S$, for all $p \in P$, if it holds that $s_1 \preceq p$ and $s_2 \preceq p$ and if all others elements $s' \in S$ with s' a lower-bound for P such that $s' \preceq s_1$ and $s' \preceq s_2$; the greatest lower bound is said to be undefined.

For a set of two elements $P = \{p_1, p_2\}$, we denote by $p_1 \sqcap p_2$ the greatest lower bound.

Lattices A *lower semilattice* is a poset $\langle S, \preceq \rangle$ where for all pair of elements $s_1, s_2 \in S$, we have that the greatest lower bound $s_1 \sqcap s_2$ exists.

Example 2.2.1. Let $\langle 2^{Q_A}, \subseteq \rangle$, a poset with Q_A a set of 3 elements. Figure 3 shows a graph with the incomparable elements of such poset.

2.3 Antichains

Closed sets A closed set is a set $L \subseteq S$ of a lower semilattice $\langle S, \preceq \rangle$ where $\forall \ell \in L$ we have that $\forall s \in S$ such that $s \preceq \ell$, then $s \in L$. Note that for two closed sets $L_1, L_2 \subseteq S$, we have that $L_1 \cup L_2$ and $L_1 \cap L_2$ are also closed sets, but $L_1 \setminus L_2$ does not result necessarily to a closed set.

Maximal/minimal elements We denote by $\lceil L \rceil$ the set of maximal elements of a closed set L which correspond to $\lceil L \rceil = \{\ell \in L | \forall \ell' \in L : \ell \preceq \ell' \Rightarrow \ell = \ell'\}$. Alternatively, to represent the set of minimal elements, the noation $\lfloor L \rfloor$ is used which has the following semantic $\lfloor L \rfloor = \{\ell \in L | \forall \ell' \in L : \ell' \preceq \ell \Rightarrow \ell = \ell'\}$.

Closure A *lower closure* of a set L on S noted $\downarrow\! L$ is the set of all elements of S that are *smaller or equal* to an element of L i.e. $\downarrow\! L = \{s \in S \mid \exists \ell \in L \cdot s \preceq \ell\}$. Note that for a closed set L we have that $\downarrow\! L = L$.

Antichain An *antichain* of a poset $\langle S, \preceq \rangle$ is a set $\alpha \subseteq S$ where all element of α are incomparable with respect to the partial order $\preceq$. Otherwise, if all elements are comparable the set is called a *chain*. Antichains allow to

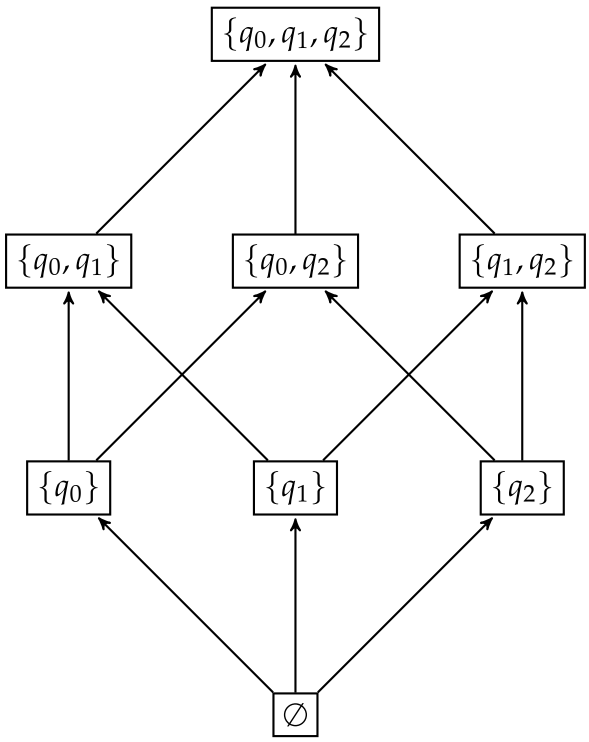

Figure 3: Example of antichains using the poset $\langle 2^{Q_A}, \subseteq \rangle$, with Q_A the set of states of an automaton A, which correspond to $Q_A = \{q_0, q_1, q_2\}$. Each directed edge of the graph correspond to a valid comparison using the set inclusion $\subseteq$. For example $\{q_0\} \rightarrow \{q_0, q_1\}$ corresponds to the inclusion $\{q_0\} \subseteq \{q_0, q_1\}$. Two elements of the graph with no connection, means that the elements are incomparable. For example, $\alpha = \{\{q_0\}, \{q_1\}, \{q_2\}\}$ is a set of incomparable elements and α is called an antichain.

represent closed set in a more compact way. For a closed set $L \subseteq S$ we can retrieve all elements of L by using the antichain $\alpha = \lceil L \rceil$. With respect to the definition of the lower closure we have that $\downarrow\alpha = L$.

2.4 Operations on antichains

This section list the classical propreties of antichains. All the examples are based on the poset defined in Example 2.2.1.

Proposition 2.4.1. Let $\alpha_1, \alpha_2 \subseteq S$ two antichains and $s \in S$:

- $s \in \downarrow\alpha_1$ iff $\exists a \in \alpha_1$ such that $s \preceq a$

  Example 2.4.1. Let $\alpha_1 = \{\{q_0, q_1\}\}$, $\{q_0\}$ belongs to the lower closure $\downarrow\alpha_1$ because, $\{q_0\} \subseteq \{q_0, q_1\}$.

- $\downarrow\alpha_1 \subseteq \downarrow\alpha_2$ iff $\forall a_1 \in \alpha_1, \exists a_2 \in \alpha_2$ such that $a_1 \preceq a_2$

  Example 2.4.2. Let $\alpha_1 = \{\{q_0\}, \{q_1\}\}$ and $\alpha_2 = \{\{q_0, q_1\}\}$, $\downarrow\alpha_1 \subseteq \downarrow\alpha_2$, since all elements of α_1 are included in the single element of α_2.

- $\downarrow\alpha_1 \cup \downarrow\alpha_2 = \downarrow\lceil \alpha_1 \cup \alpha_2 \rceil$

- $\downarrow\alpha_1 \cap \downarrow\alpha_2 = \downarrow\lceil \alpha_1 \sqcap \alpha_2 \rceil$ where $\alpha_1 \sqcap \alpha_2$ is defined as $\alpha_1 \sqcap \alpha_2 = \{a_1 \sqcap a_2 | a_1 \in \alpha_1, a_2 \in \alpha_2\}$, that is the greatest lower bound of elements between the two antichains.

3 Existing implementations

3.1 AaPAL

Bohy's **A**ntichain **a**nd **P**seudo-**A**ntichain **L**ibrary [Boh14a] is an open-source generic library for the manipulation of antichains and pseudo-antichains data structures, implemented in `C`. In this section we will mainly focus on the implementation of antichains.

Antichain representation An antichain is represented by a `struct`, containing as attributes the size of the antichain, and the incomparable elements of the antichains, as a list. The list is manipulated using the `GSList` object from the `glib` library. To allow modularity, the type of the elements is `void`.

Operations The operations implemented in `AaPAL` are the union, intersection and appartenance defined in Proposition 2.4.1. An interesting remark is that most of the complexity is given as a paramater to the functions. For example the function to compare two elements in an antichain is given as a parameter. It means that the complexity to define the domain of the antichain, must be implemented in the compare function. Same pattern goes for the intersection operation, the function to compute the intersection must be provided by the user. Also basic operations such as creating an antichain, adding an element to an antichain, checking emptiness or cloning an antichain are implemented. in `AaPAL`.

3.2 Antichains for Dedekind's problem

Dedekind's problem The Dedekind's problem correspond to a problem introduced by Richard Dedekind in the 19th century. He defined a rapidly growing sequence of natural numbers. The n_{th} Dedekind number is the count of antichains in the powerset of set with n elements.

De Causemaecker and De Wannemacker [DCDW14] improved and implemented a multithreaded algorithm to find the n^{th} Dedekind number that allow to compute the n^{th} Dedekind number from the powerset of a set with $(n-2)$-elements, that is re-use the number of antichains for $(n-2)$-set to compute the nth Dedekind number.

De Causemaecker and De Wannemacker only provide the executable of their algorithm in their paper for the Dedekind algorithm [DCDW14]. Hoedt proposed in [Hoe15a] an algorithm to find the ninth Dedekind number, which requires a representation of antichains. The representation used is an extension of the implementation proposed by De Causmaecker and De Wannemacker. The source of his implementation was found in is personal GitHub [Hoe15b]. We will therefore only focus on the implementation of Hoedt, which is an extension of the implementation proposed by De Causmaecker and De Wannemacker, but by representing an antichains using bitarray-like methods.

Antichains representation According to Hoedt in [Hoe15a], the first proposed representation in [DCDW14] was done by using a `TreeSet`. With the objective to improve the performance of the important operations, for example the intersection, Hoedt used another way to represent antichains, by using a bitarray-like representation. The goal of the bitarray representation is to represent set of natural numbers by a binary representation. For example the subset $\{1, 2\}$ would be represented by the binary number 11_2 and the subset $\{1, 3\}$ by 101_2 corresping respectively to 3_{10} and 5_{10}. Then an antichain is represented by enabling the bit of the corresponding number of an element of the antichain. Therefore, an antichain $\alpha = \{\{1, 2\}, \{1, 3\}\}$ is represented by the bitarray where the bit at indices 3 and 5 are enabled, that is 10100.

4 Next year overview

4.1 Requirements

Different requirements were defined regarding the desired implementation. The first requirement is that the library will have to be used in `Owl`, therefore the language of reference is `Java 10`. the choice of the language, even if highly encouraged by the library `Owl`, is also interesting due to features that are available in `Java`. One of which would interest us in our work, is the ability to profile our implementation. Profiling is a form of measurement of the time complexity or the memory usage of a program execution. There exists multiple tools to profile a `Java` program such as VisualVM, YourKit or JProfiler.

4.2 Owl library

Owl (Omega Word and automata Library) is a library and tool-set tailored for – but not limited to – semantic-based translations from LTL to deterministic automata.

`Owl` is a library developed at the *Technische Universität München* maintained by Salomon Sickert. The library can be used as a command line tools or from an online version available at *http://rabinizer-owl.pe.hu/*. It is implemented in `Java`.

It provides automata transformation and analysis tools as well as synthesis algorithms.

Symbolic antichain-based algorithms are missing.

This still need to be completed as I'm not sure yet what to talk about

4.3 Framework and implementation

The goal will be to try to find a good way to implement antichains representation and operations. One of the possible way to do so is to provide a framework that will allow to implement operations functions depending on the universe of the antichains. In `AaPAL` all the complexity is implemented in the `compare_elements` that must be given to the library functions. It that case, all the complexity must be implemented by the user to define the universe of the antichains. An example of specific implementation for natural numbers is the one proposed by Hoedt, presented in section 3.2, using bitarray. A first step is to proposed an framework for users to implement the specifics of the universe necessary for their algorithms. Next year thesis objective will be to propose a generic and abstract API, and implement specifics for known universe and test them on state-of-the-art algorithms, especially in automata theory.

Fill-in bib correctly!

References

[Boh14a] Aaron Bohy. Aapal website `http://lit2.ulb.ac.be/aapal`, 2014.

[Boh14b] Aaron Bohy. *Antichain based algorithms for the synthesis of reactive systems*. PhD thesis, Université de Mons, 2014.

[DCDW14] Patrick De Causmaecker and Stefan De Wannemacker. On the number of antichains of sets in a finite universe. *arXiv:1407.4288 [math]*, July 2014. arXiv: 1407.4288.

[DWDHR06] M. De Wulf, L. Doyen, T. A. Henzinger, and J. F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Computer Aided Verification*, volume 4144, pages 17–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[FJR09] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An Antichain Algorithm for LTL Realizability. In *Computer Aided Verification*, volume 5643, pages 263–277. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Hoe15a] Pieter-Jan Hoedt. Parallelizing with mpi in java to find the ninth dedekind number. 2015.

[Hoe15b] Pieter-Jan Hoedt. Hoedt's github repository `https://github.com/mrTsjolder/WV-Dedekind`, Last updated in 2015.

[Maq11] Nicolas Maquet. *New Algorithms and Data Structures for the Emptiness Problem of Alternating Automata*. PhD thesis, Université Libre de Bruxelles, 2011.

[Sal16] Sickert Salomon. Owl website `https://www7.in.tum.de/ sickert/projects/owl/`, 2016.