UNIVERSITÉ LIBRE DE BRUXELLES

DÉPARTMENT D'INFORMATIQUE

MEMO-F-403
PREPARATORY WORK FOR THE MASTER THESIS

Implementing data structures for Partially Ordered Set

Hakim BOULAHYA

Supervised by
Emmanuel FILIOT

May 17, 2018

Contents

Todo list

1 Introduction

In the paper, should I use We or I, or nothing ?

Data structures play an important role in algorithms complexity. With the objective to improve standard algorithms in automata theory, researchers at the ULB have implemented new algorithms to resolve important problems in the field. Those implementations uses antichains, which are data structures that allow to represent elements in a partially ordered set, in a more compact way. The goal of this preparatory work is to motivate the interest that are made in data structures for partially ordered set, especially antichains, and why we need efficient implementation. It also defines the desired objectives and propose a state of the art of the various existing implementations.

complexity vs efficiency ?

References maybe ?

1.1 Context and motivations

Theoritical context

Universality check with antichains 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 verifications-related problems can be reduced in polynomial time to it.

Source of this reduce statement ?

Let $A = \{Q_A, \Sigma, q_0, \delta, F_A\}$ be a finite automaton, we want to check whether or not 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 complementary 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 a set of states such that $S \subseteq 2^{Q_A}$ and $S \cap F_A = \emptyset$. The goal is to find a state $s \in S$, which when computing a word $w = w_0 w_1 ... w_n$, the automaton will ends in s. The idea is to first start with a set state with all non-accepting states, that is $T = Q_A \setminus F_A$. Then compute the sets that when reading a letter will lead to T, that is the function $Pre(S) = \{s \mid \exists s' \in S \cdot \exists \sigma \in \Sigma \ : \ s \xrightarrow{\sigma} s'\}$.

t prove that A is not universabl

The only way to compute the complementary of A is to first determinise it, which is hard. Let define A_d as the deterministic finite automaton (DFA) of A such that $L(A_d) = L(A)$. When A is transformed to A_d, the set of states of the DFA is the powerset of states, that is a state of the DFA A_d is represented by a subset of state of A. Now if $L(\bar{A})$ is the empty set, it means that there is no set of states that for which there exists a word such that, when computed by $\bar{A}$, this word is accepted. With the set $X = \{P, \subseteq Q, P' \subseteq Q, P' \cap \bar{F} = \emptyset | \exists w \in \Sigma^* s.t. P \xrightarrow{w} P'\}$, it provides a way to check the existence such set of states. The goal is to show that if there exists no set of states P' such that $P' \cap F = \emptyset$

i.e. $X = \emptyset$, then it is shown that $L(\bar{A}) = \emptyset$, therefore A is universal. The difficulty of the problem arise at the computation of X. Checking intersection between the accepting states set and all the subsets is costly. For a subset $P \subseteq Q$, for all subsets p such that $p \subseteq P$, it is known that, computing the word w starting from p or P, will lead to the same final state. Therefore it is more interesting to compute the maximal sets $\lceil X \rceil$, which correspond to an antichain of set of states that are $\subseteq$-incomparable. In section 2, we give formal definition of notions such as maximal sets or incomparable elements.

Costly vs expensive vs complexe vs hard etc..

Include Xi explanation

In [DWRLH06], the algorithm proposed follow an equivalent idea by using game theory. The universality problem is reduce to a two-player reachability game, which can be done in polynomial time. The objective of the game is for the protagonist to establish that the automaton A is not universal. To this end, the protagonist will provide a word, a letter at a time, and find a strategy that try to show that A ends in a rejecting state. The protagonist only has a strategy to win the game if and only if A is not universal.

Objective

this thesis is to provide an efficient implementation of different data structures that allow to compactly represent partially ordered sets, specifically antichains and pseudo-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`. A second step will be to implement antichain-based algorithms using the new antichains implementation and study the performance.

Include that UP is a problem important because verif-related problem reduce to it

Either put this here in the intro, or in the next year section

Related work `AaPAL` is a generic library is a that was implemented in the frame of Aaron Bohy's PhD thesis [Boh14b] to provide an antichain library. It is implemented in `C`. Another implementation of antichains have been done by De Causmaecker and De Wannemacker in [DCDW]. 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, Hoedts in [Hoe] has extended [DCDW] antichains implementation by using bit sequence instead of tree reprensentation.

How are antichains implemented in Acacia+ ? Is it AaPAL or another impl. ?

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

Discuss impl. specifics etc in chapter Implementation

What operations are implemented in those papers ?

What domain (of sets) is is used in

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 and pseudo-antichains.

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

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 $\langle S, \preceq \rangle$.

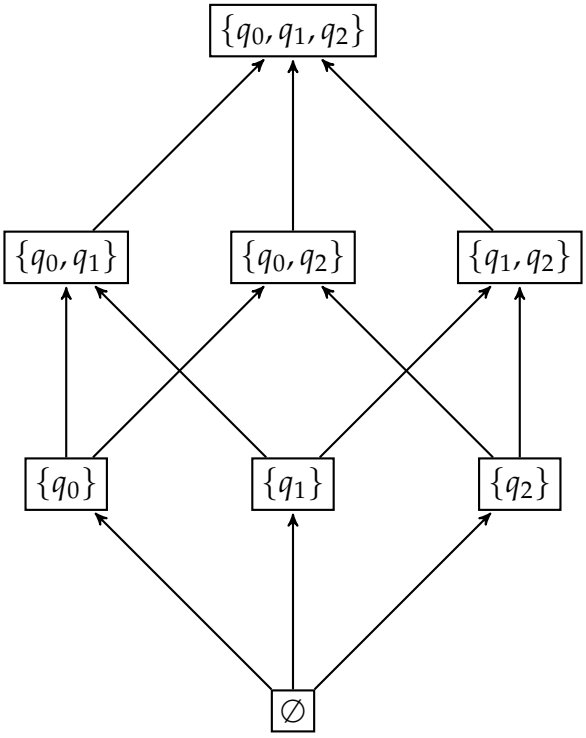

Figure 1: Example of antichains using the poset $\langle 2^{Q_A}, \subseteq \rangle$, With Q_A the set of state of the automaton A, which correspond to $Q_A = \{q_0, q_1, q_2\}$. Each directed edge of the graph correspond to a valid comparison using the set inclusion $\subseteq$. For example $\{q_0\} \rightarrow \{q_0, q_1\}$ 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.

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

Bounds Let $\langle S, \preceq \rangle$ a partially ordered set. We denote the *greatest lower bound* of the two elements $s_1, s_2 \in S$ by $s_1 \sqcap s_2 \in S$. The greatest lower bound is defined as follow: $s_1 \sqcap s_2 \preceq s_1$, $s_1 \sqcap s_2 \preceq s_2$ and for all $s' \in S$ we have that if $s' \preceq s_1$ and $s' \preceq s_2$ then $s' \preceq s_1 \sqcap s_2$.

Example 2.2.1. A more interesting example is the set inclusion comparison. Let A be automaton defined in Example **??**. Let $\langle 2^{Q_A}, \subseteq \rangle$, a poset. Figure 1 shows a graph with the incomparable elements of such poset.

Include definition of least upper bound

Still need to define the automaton, here or in another section.

Fix figure formulation as we havent

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.

2.3 Antichains

2.3.1 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 $\lfloor L \rfloor = \{\ell \in L | \forall \ell' \in L : \ell' \preceq \ell \Rightarrow \ell = \ell'\}$.

Meaning of — vs . vs : in set definition ?

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 $\preceq$. Antichains allow to 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

Cite original paper, FJR11 from Bohy' phd

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 $\preceq a$
- $\downarrow \alpha_1 \subseteq \downarrow \alpha_2$ iff $\forall a_1 \in \alpha_1, \exists a_2 \in \alpha_2$ such that $a_1 \preceq a_2$
- $\downarrow \alpha_1 \cup \downarrow \alpha_2 = \downarrow \lceil \alpha_1 \cup \alpha_2 \rceil$
- $\downarrow \alpha_1 \cap \downarrow \alpha_2 = \downarrow \lceil \alpha_1 \sqcap \alpha_2 \rceil$

Give complete definition for interesection

3 Implementation

Java already provide built-in implementation for `Set`.

Includes limitation of Java built-in and different possible solution for antichains found on stack overflow and others

In this thesis we are more interested in partially ordered sets as totally ordered sets are already implemented in`Java` built-in `SortedSet`.

3.1 Summary of objectives

The main focus of the thesis is to be able to provide an efficient implementation of antichains and pseudo-antichains in `Java`. The first step is to provide an interface for the different operations that can be applied to antichains. We then give a description of the implementation. Antichains provide a way to represent in a compact way partially ordered set that are closed. Pseudo-antichains are an extension of antichains and provide a compact way to represent partially ordered sets. Pseudo-antichains does not specifically require closed set.

Our goal is to find a way to not keep all the closure all element of the antichain in memory, but be able to retrieve those elements or check the belonging of a closure element from the incomparable elements of the antichain.

3.2 Existing implementation

3.2.1 AaPAL

Bohy's **A**ntichain **a**nd **P**seudo-**A**ntichain **L**ibrary [Boh14a] is an open-source `C` library for the manipulation of antichains and pseudo-antichains data structures.

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 described in this section. 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.

Operations such as creating an antichain, adding an element to an antichain, checking emptiness or cloning an antichain are implemented in `AaPAL`. In addition to that

create add elem clone
interesection Union
compare 2 antichains
containing element closure size

3.2.2 ABD

3.2.3 Hoedt's ABD modifications

Referring to [Hoe]

3.3 Difficulties

Is actually [Hoe] what we what to do, and if not, what will be the differences ?

One of the difficulties is to define the domain of the elements that the antichains should work on. In `AaPAL` the all complexity is implemented in the `compare_elements` that must be given to the library functions. It that case, all the complexity of the must be implemented by the user to define the domain of the antichains.

3.3.1 Tree vs Bitarray representations

What is the difference beetween domain, univers and set ?

3.4 Possible solutions

In the case of the domain specifications, a good approach could be to provide an interface/abstract class to the users, to let them provide their own implementation. In addition to that, usual domain such a natural numbers or others well known domain used, could be directly implemented in the library.

3.5 New implementation

4 Next year overview

—

4.1 Interface and implementation

The interface proposed in **??** gives a first overview of the desired implementation.

4.2 Possible extensions

As we mainly focus on efficiency, it could be interesting to use a `C` implementation such as `AaPAL`, and provide bindings to `Java`. We could try this method as an alternative to a pure `Java` implementation and compare performances.

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.

[DCDW] Patrick De Causemaecker and Stefan De Wannemacker. On the number of antichains of sets in a finite universe.

[DWRLH06] Martin De Wulf, Jean-François Raskin, Doyen Laurent, and Thomas A. Henzinger. Antichains: A new algorithm for checking universality of finite automata. 2006.

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

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