Implementing Pomset Automata

Edgar Delaporte
(supervisor: Amazigh Amrane)

June 2024

In this document, we present an implementation of an extension of $L^\star$ on pomsets. We examine several data-structures for pomsets and their respective benefits in regards to this algorithm. We propose functional improvements of the original algorithm as well as a method allowing us to enforce its termination without using bimonoid isomorphism. We then use these results to discuss a similar extension of another active learning algorithm, L^λ, on pomsets.

Keywords
Automata Theory, Pomset, Active Learning, Algebra

Laboratoire de Recherche de l'EPITA
14-16, rue Voltaire
94270 Le Kremlin-Bicêtre CEDEX
France

Copying this document

Contents

Chapter 1

Introduction

Automata Theory is at the core of many theoretical results and applications such as the computation of the complexity of algorithms, lexical analysis for compilers and model checking. More generally, automata describe formal languages, allowing them to be used to analyse systems such as programming languages, proof systems, communication protocols or human languages. In most cases, automata model sequential processes. A state of an automaton then represents a state of the system which is modelled and transitions are used to go from state to state when an event occurs or an instruction is executed.

Automata learning algorithms are particular methods of machine learning in which the model can learn state machines by querying an information source holding the target machine. These methods are used to produce state machine models of hardware and software systems, thus facilitating the use of formal verification methods. The first active learning algorithm to be described was L^* [1] which constructs a minimal deterministic automata equivalent to a target state machine within a finite number of queries. It proceeds by searching iteratively all equivalence classes of the Myhill-Nerode relationship [2], with respects to prefix-suffix relations. It then constructs a deterministic finite automata (DFA) using the thus found partition of the language.

Pomsets (Partially ordered Multi-sets) are algebraic objects used to model concurrent or non-deterministic programs. They are an extension of words in which we discard the 'strongly connected' property of the ordering function, thus creating partially ordered sets. Automata recognising pomsets [3] [4] [5] are useful to study the behaviour of concurrent systems.

In the same manner that words can be recognised by monoids, pomsets can be recognised by bimonoids [6]. Bimonoids are a special case of tree automata in which transitions are of arity 0 or 2. TATA [7] has shown that the Myhill-Nerode theorem [2] can be generalised to tree automata and Lodaya and Weil have done the same for pomset automata [8]. This provides necessary and sufficient conditions for applying active learning methods on bimonoids.

An extension of L^* on pomsets have been described [6]. This algorithm expends the classical approach of active learning on monoids, by learning bimonoids recognising pomsets. It searches

for a partition of the target languages in regards to a subpomset-context relation and uses it to build a bimonoid in a similar way as the original algorithm. However, bimonoid isomorphism having never been described, equivalence queries do not seem to be computable given the current state of the art. This seems to be one of the reasons as to why there exists no actual implementation of said algorithm to the best of our knowledge.

In this document, we present an implementation in C++ code of this extension of L^* on pomsets, providing minimal data-structures for each step in order to guarantee minimal space and time complexity. We also examine the respective benefits of directed acyclic graphs (DAG) and abstract syntax trees (AST) in regards to the representation of pomsets and conclude that the time complexity saved by the recursive nature of AST is worth their greater space complexity. Lastly, we provide a naive method allowing us to perform equivalence queries on bimonoids without isomorphism in the case of finite language using a strategy similar to brute-force attack.

We then use these results to discuss an extension of another active learning algorithm on pomsets, L^λ [9]. We discuss how the improvement relatively to L^* given by this algorithm would still hold on pomset languages and provide some leads for proving such assertion, which will be the nature of our future work.

We then discuss probabilistic, algebraic and functional methods of deciding isomorphism of bimonoids and randomly generating said objects.

Chapter 2

Notations and Mathematical Definitions

2.1 Pomsets

Definition 1: A word is a set of letters of a certain alphabet Σ on which we consider a linear order λ. An order λ over a certain set is said to be linear iff it is reflexive, transitive, antisymetric and strongly connected. We define the "strongly connected" property of a set S as follow:

$$\forall (a, b) \in S^2, a \leq b \vee b \leq a$$

Instinctively, we can see words as a sequence of letters that we can always arrange from the first to the last.

$$p \text{ —— } o \text{ —— } u \text{ —— } l \text{ —— } e$$

Figure 2.1: A word over the alphabet Σ with $\lambda = \{(p, o), (o, u), (u, l), (l, e)\}$

.

We now want to extend the notion of words to sets for which some elements might not be comparable. Posets (**P**artially **O**rderded **Sets**) are thus a generalization of words in which we accept partial orders λ.

Definition 2: An order λ over a certain set is said partial iff it is reflexive, transitive and antisymetric.

We can note that the only condition which differentiates total from partial ordering is the absence of the "strongly connected" property, making total orders a particular case of partial or-

ders.

Definition 3: A labelled poset over some alphabet Σ is a tuple $(S, <, f)$ with S a set of elements of A, $<$ a partial order over S and f a labelling map that associates a label to any element of S.

Definition 4: A pomset [10] (**P**artially **O**rdered **M**ulti**Set**) over some alphabet Σ is an isomorphism class of labeled posets over Σ. Two labeled posets are part of the same isomorphism class if there exists a bijection between the two that preserves labelling and ordering. Within a pomset, two elements x and y that cannot be compared, are said to be in parallel and are denoted $x||y$.

The empty pomset ϵ is the only pomset for which the set of Letter S is the empty set.

We can graphically represent a pomset as a Hass Diagram as follow:

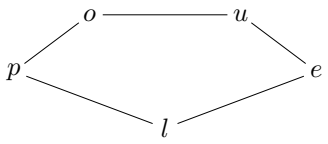

Figure 2.2: A pomset over S = $\{x_1, x_2, x_3, x_4, x_5\}$ with $\lambda = \{(x_1, x_2), (x_2, x_3), (x_3, x_5), (x_1, x_4), (x_4, x_5)\}$ and $f : S \longrightarrow \alpha$ a labelling map such as $f(x_1) = p, f(x_2) = o, f(x_3) = u, f(x_4) = l, f(x_5) = e$.

2.1.1 Rational operations

Let $(p_1, <_1, \lambda_1)$ and $(p_2, <_2, \lambda_2)$ be two disjoint pomsets over respectively Σ_1 and Σ_2, we define the following operations [11]:

- The parallel product of p_1 and p_1 is the pomset $p_1||p_1 = (p_1 \cup p_2, <_1 \cup <_2, \lambda_1 \cup \lambda_2)$ over $(\Sigma_1 \cup \Sigma_2)$, as illustrated in Figure 2.3.

- The sequential product of p_1 and p_2 is the pomset $p_1 \bullet p_2 = (p_1 \cup p_2, <_1 \cup <_2 \cup (p_1 \times p_2), \lambda_1 \cup \lambda_2)$ over $(\Sigma_1 \cup \Sigma_2)$, as illustrated in Figure 2.4.

- The substitution of a letter ξ by p_1 in p_2 is the new pomset equal to p_2 with all occurrences of the letter ξ replaced by copies of p_1. This is illustrated by Figure 2.5.

Let L be a sp-language of pomsets labelled by B, P a pomset labelled by A and $\xi \in A$. The substitution of ξ by L in P, denoted L o_ξ P, is the language L′ in the alphabet $(A/\{\xi\}) \cup B$ where every pomset is the result of the substitution of every element labelled by ξ in P by an element of L.

For example, let $B = \{a, b\}$, $A = B \cup \{\xi\}$, $P = a\xi(b||\xi)$ and $L = \{a||b, a\}$, then

$$P \; o_\xi \; L = \{a(a||b)(b||a||b), a(a||b)(b||a), aa(b||a||b), aa(b||a)\}$$

These operations can be repeated a certain number of times over an element e and are then called parallel iteration $e^{\oplus}$, sequential iteration e^{+} and iterated substitution $e^{*\xi}$.

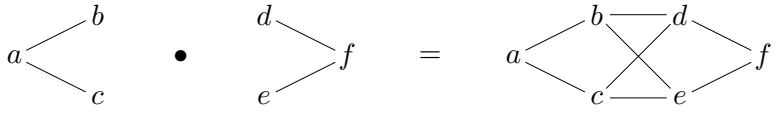

Figure 2.3: The sequential product of $p_1 = a(b||c)$ with $p_2 = (d||e)f$.

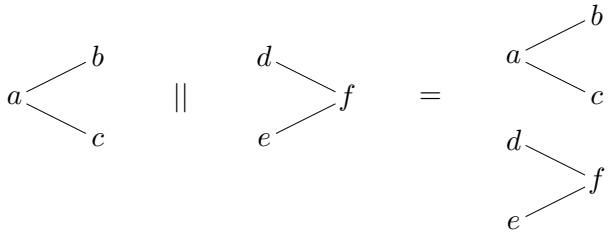

Figure 2.4: The parallel product of $p_1 = a(b||c)$ with $p_2 = (d||e)f$.

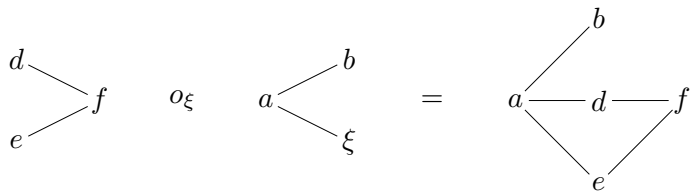

Figure 2.5: The substituion of ξ by $p_1 = (d||e)f$ in $p_2 = a(b||\xi)$.

2.1.2 SP-Pomsets

Definition 5: The set of series-parallel pomsets (or sp-pomsets for short) is the set of pomsets that are closed under parallel and sequential product. The empty pomset is series-parallel.

We denote $SP(\Sigma)$ the series-parallel algebra of Σ, the set of all sp-pomsets that can be constructed with letters of Σ.

We call the width of a pomset the length of the longest of its anti-chains. ϵ has a width of 0.

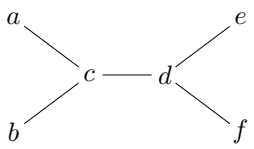

Figure 2.6: a sp-pomset

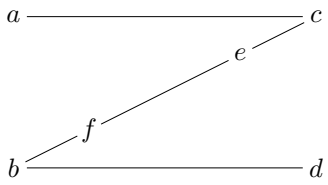

Figure 2.7: a non-sp-pomset

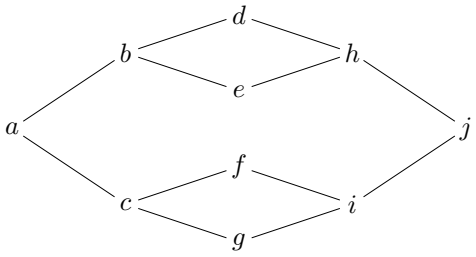

Figure 2.8: $p = a((b(d||e)h)||(c(f||g)i))j$, a sp-pomset of width 2.

Definition 6: A pomset of which the width is a finite number is said to have bounded-width.

Definition 7: A pomset language L over an alphabet Σ is a certain subset of $SP(\Sigma)$.

2.1.3 Classes of languages

Definition 8: In algebra on words, a language over Σ is a set of words constructed from letters of Σ. A language L is said regular iff it is accepted by a finite automaton. It is said rational iff all its words can be described using only concatenation, union and iteration.

Proposition 1: For any pomset language L, being regular is equivalent to being rational [12].

Definition 9: A pomset language is said to have bounded-width iff all of its elements have bounded-width.

A pomset language for which all elements are sp-pomsets is called a sp-language.

Definition 10: A sp-language is said to be series-rational if all its elements can be described using only concatenation, parallel product, sequential iteration and union.

Proposition 2: Every series-rational sp-languages has bounded-width [13].

We then can define general rational pomset languages [8] as subsets L of $SP(\Sigma)$ with the following properties:

- If p_1 and p_2 are elements of L, then so are $p_1 \bullet p_2$, $p_1||p_2$ and p_1^+.
- If p_1 and p_2 are elements of L and if $\xi \in L$, then $p_1 o_\xi p_2$ is so.
- If p is a rational expression and if $\xi \in L$, then $p^{*\xi}$ is so.

It is important to note that, while all series-rational sp-languages are rational, the converse does not hold. There are indeed rational languages that does not have bounded-width. For example, $a^{\oplus}$ is rational while not being bounded-width.

Choosing whether or not to consider these cases has a great influence on the complexity of the problems with which we will be confronted. We will therefore sometimes choose to consider only series-parallel sp-languages in order to limit ourselves to more reasonable cases in terms of algorithmic complexity.

2.2 Myhill-Nerode relationship

2.2.1 On words

Given two words (w_1, w_2) of a language L, we define a distinguishing extension as a word c such that :

$$w_1 \bullet c \in L \neq w_2 \bullet c \in L$$

We then define the Myhill-Nerode relationship $\sim_L$ as follows:

$$\forall (w_1, w_2) \in L^2, w_1 \sim_L w_2 \iff \nexists c \in \Sigma^*, w_1 \bullet c \in L \neq w_2 \bullet c \in L$$

The Myhill-Nerode theorem shows that this relationship partitions L into equivalence classes and that L is regular iff this partition is of finite order.

Proposition 3: A minimal DFA has as many states as the carnality of its partition in regards to the Myhill-Nerode relationship.

Such properties allows regular languages of infinite cardinality to be described with a finite number of elements by finding every equivalent class with respect to the Myhill-Nerode relationship. As we will see in the following sections, respecting this property is a sufficient condition for a language to be learned using active learning algorithms.

2.2.2 On pomsets

Similarly, we can define a Myhill-Nerode-like theorem on pomset languages. Given two pomsets (p_1, p_2) of a pomset language L, we a define a context as a pomset e containing exactly one occurrence of the letter ξ such that :

$$p_1 \ o_\xi \ c \in L \neq p_2 \ o_\xi \ c \in L$$

Previous works on Pomset Automata [3] have show that contexts of pomset languages share all the properties of distinguishing extensions of **2.2.1** and we therefore define a similar relationship:

$$\forall (p_1, p_2) \in L^2, p_1 \sim_L p_2 \iff \nexists c \in C, p_1 \ o_\xi \ c \in L \neq p_2 \ o_\xi \ c \in L$$

With C the set of all pomsets that can be constructed using letters of the alphabet and the letter ξ.

Such relation partition L into equivalence classes in a similar manner than on languages on words [3]. This is a sufficient condition that active learning algorithms on languages of words can be extended to languages of pomsets.

2.3 Pomset recognisers

2.3.1 Bimonoids

Definition 11: A bimonoid is a category defined as a tuple $(M, \bullet, ||, \mathbf{1})$ where M is a set called the carrier, $\bullet$ is a binary associative operation over M, $||$ is a binary associative commutative operation over M and $\mathbf{1}$ is a neutral operand for both $\bullet$ and $||$.

As monoids recognise words, bimonoids recognise pomsets [13] and can be manipulated in a similar manner. This is a convenient algebraic solution to study pomset languages without having to care about the algorithmic complexities induced by pomset automata. We will however need to operate on state machines in the following parts and therefore define Pomset recognisers to circumvent this issue.

2.3.2 Pomset recognisers

Definition 12: A pomset recogniser is a tuple $(M, \bullet, ||, \mathbf{1}, i, F)$ where $(M, \bullet, ||, \mathbf{1})$ is a bimonoid, i is a mapping function from the alphabet to M, and F is a subset of M of final states.

Pomset recognisers are therefore tree automata with M being the set of states, F the set of final states and ($\bullet$, $||$) the transition functions.

For example, given $M = \{q_a, q_b, q_1, q_\perp, \mathbf{1}\}$, i the function such that $i(a) = q_a$ and $i(b) = q_b$, $F = \{q_1\}$ and ($\bullet$, $||$) defined in Tables 2.1 and 2.2.

We obtain the pomset recogniser $P = (M, \bullet, ||, \mathbf{1}, i, F)$ that recognises the language $L = (a||b)^* = \{\epsilon, a||b, (a||b)(a||b), (a||b)(a||b)(a||b), ...\}$.

$\bullet$	q_a	q_b	q_1	$q_\perp$	**1**
q_a	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	q_a
q_b	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	q_b
q_1	$q_\perp$	$q_\perp$	q_1	$q_\perp$	q_1
$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$
1	q_a	q_b	q_1	$q_\perp$	**1**

Table 2.1: $\bullet operation$

$\mid\mid$	q_a	q_b	q_1	$q_\perp$	**1**
q_a	$q_\perp$	q_1	$q_\perp$	$q_\perp$	q_a
q_b	q_1	$q_\perp$	$q_\perp$	$q_\perp$	q_b
q_1	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	q_1
$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$	$q_\perp$
1	q_a	q_b	q_1	$q_\perp$	**1**

Table 2.2: $\mid\mid$ operation

2.4 L^*

2.4.1 General principle

L^* [1] was one of the first learning algorithm to be described. It creates a minimal DFA equivalent to a state machine held by a teacher by querying it for membership and equivalence.

Minimally adequate teacher

A teacher is an abstract object holding a state machine A and capable of answering queries about the later.

Definition 13: A teacher is said to be minimally adequate if it can answer at least the two following types of queries: **Membership queries**, consisting of answering *yes* or *no* as a whether the word w belongs to the language of A. **Equivalence queries**, consisting of determining whether a submitted DFA B is equivalent to A. In negative cases, the teacher must submit a counterexample, i.e. a word that distinguishes the languages of A and B.

Proposition 4: If a teacher is minimally adequate, L^* will be capable of learning its language within a finite number of queries [1].

Observation table

In order to classify the information given by the teacher, the learner maintains an observation table.

Definition 14: An observation table over an alphabet Σ is a tuple (S, E, T) with S a finite prefix-closed set of words over Σ, E a finite suffix-closed set of words over Σ and T a mapping function from $((S \cup S.\Sigma).E)$ to $\{0, 1\}$. A set of words X is said to be prefix-closed (resp. suffix-closed) iff all prefixes (resp. suffixes) of all elements of X are also in X. For any $s \in (S \cup S.\Sigma)$ we define $row(s)$ as the function $f : e \to T(s.e)$.

For example, with $S = E = \{\epsilon, a, b\}$ and T such that $T(x) = 1$ for $x \in \{\epsilon, a, aa, aaa\}$ and $T(x) = 0$ otherwise, we have the following observation table:

	ϵ	a	b
ϵ	1	1	0
a	1	1	0
b	0	0	0
aa	1	1	0
ab	0	0	0
ba	0	0	0
bb	0	0	0

Table 2.3: Observation table 1

By constructing S, the learner identifies the different equivalence classes of the target language in regards to the Myhill-Nerode relationship. E then represents the set of distinguishing extensions. In a similar manner, it identifies the transition functions by constructing T.

Definition 15: An observation table is said to be closed iff for every $t \in S.\Sigma$ there exists a $s \in S$ such that $row(t) = row(s)$. An observation table is said to be consistent iff

$$\forall (s_1, s_2) \in S^2, row(s_1) = row(s_2), \forall a \in \Sigma, row(s_1.a) = row(s_2.a)$$

Definition 16: Given a closed, consistent observation table (S, E, T) we define an acceptor $M((S, E, T))$ as a tuple (Q, q_0, F) with $Q = \{row(s), s \in S\}$ a set of states, q_0 the unique initial state and F the subset of Q of final states. $M((S, E, T))$ is a DFA by construction and has exactly $|S|$ states.

The algorithm starts with $S = E = \{\epsilon\}$. It will then loop while the observation table is non-closed or non-consistent and will try to remedy one of these properties with each iteration. Enforcing closure is done by adding elements to S until the condition is met. Enforcing consistency is done by adding elements to E that distinguishes elements that were indistinguishable beforehand. Once the table is closed and consistent, the learner will submit an acceptor A corresponding to the table as an hypothesis to the teacher. If the teacher finds no counterexample, then A is the minimal target DFA and the algorithm terminates. Otherwise, the learner will extract a new distinguishing element from the given counterexample and will continue checking for closure and consistency.

Proposition 5: For any minimally adequate teacher T holding a DFA A of n states and given m the maximum length of a counterexample handed by T, L^* will produce a minimal DFA B equivalent to A in polynomial time over n and m [1].

2.4.2 Extension on pomset recognisers

It is possible to extend L^* to learn pomset recognisers instead of DFA [6]. To avoid any ambiguity, we will refer to such extension as L_p^*.

Definition 17: The observation table for L_p^* over an alphabet Σ is defined analogously as to L^*'s. Given a placeholder letter ξ, the observation table is a tuple (S, E, T) with S a set of pomsets

known as *sub-pomsets*, E a set of contexts and T a mapping function from $((S \cup S.\Sigma \cup S||\Sigma)\ o_\xi\ E)$ to $\{0, 1\}$. The row function is defined similarly as in Definition 14.

For example, with $S = \{\epsilon, a, b\}$, $E = \{\xi, \xi.a, \xi||a, \xi||b\}$ and T such that $T(e) = 1$ iff $e \in (a^*)||b$ and $T(e) = 0$ otherwise, we have the following observation table:

	ξ	$\xi.a$	$\xi\|\|a$	$\xi\|\|b$
ϵ	0	0	0	0
a	0	0	0	1
b	0	0	1	0
aa	0	0	0	1
ab	0	0	0	0
ba	0	0	0	0
bb	0	0	0	0
a | |b	1	0	0	0

Table 2.4: Observation table 2

Closure and consistency are defined as in Definition 15, we however need two new properties for our observation table to produce correct outputs, associativity and sharpness.

Definition 18: An observation is said to be $\gamma - associative$ for $\gamma \in (\bullet, ||)$ iff

$$\forall (s_1, s_2, s_3, s_l, s_r) \in S^5, row(s_l) = row(s_1 \gamma s_2) \wedge row(s_r) = row(s_2 \gamma s_3) \implies row(s_l \gamma s_3) = row(s_1 \gamma s_2)$$

Definition 19: An observation table is said to be sharp iff

$$\forall (s_1, s_2) \in S^2, row(s_1) = row(s_2) \implies s_1 = s_2$$

Proposition 6: A sharp table is consistent [6].

Definition 20: A closed, associative, sharp observation table (S, E, T) induces an hypothesis $H = (Q, +, ||, 1, i, F)$ with:

- $Q = \{row(s), s \in S\}$
- $\forall (s_1, s_2) \in S, row(s_1) \bullet row(s_2) = row(s_1 \bullet s_2)$
- $\forall (s_1, s_2) \in S, row(s_1)||row(s_2) = row(s_1||s_2)$
- $1 = row(\epsilon)$
- $\forall a \in \alpha, i(a) = row(a)$
- $F = \{row(s), s \in S, row(s)(\xi) = 1\}$

An hypothesis defined this way is a pomset recogniser by construction. We have $|Q| = |S|$. We can observe that the constructed pomset recogniser will have as many states as the number of rows of the column. As shown in the previous section, there exists a bijection between

the states/rows and the equivalence classes of the Myhill-Nerode relationship of the target language.

Similarly to L^*, this algorithm starts with $S = \{\epsilon\}$ and $E = \{\xi\}$. It will then loop while the observation table is non-closed or non-associative and will try to remedy one of these properties with each iteration. Enforcing closure is done by adding elements to S until the condition is met. Enforcing $\gamma - associativity$ is done by adding new contexts to E that distinguishes elements that were indistinguishable beforehand. Once the table is closed and associative (which implies sharpness [6]), the learner will submit an hypothesis H corresponding to the table to the teacher. If the teacher finds no counterexample, then H is the minimal target pomset recogniser and the algorithm terminates. Otherwise, the learner will extract a new context from the given counterexample and will continue checking for closure and associativity.

Proposition 7: for any minimally adequate teacher T holing a pomset recogniser A of n states over an alphabet α of k letters and given m the maximum length of a counterexample given by T, L_p^* will produce a minimal pomset recogniser B equivalent to A in $O(n^3 + nm + kn)$ [6].

Chapter 3

Implementation

3.1 Pomsets

In order ensure the robustness and speed of our implementation, it is necessary to ensure the spatial minimality of the structures used and the temporal minimality of their algorithms. As we will see, these two characteristics don't always go hand in hand, so we will have to choose between several implementations of our objects, each presenting different advantages and disadvantages.

3.1.1 DAG

Directed acyclic graphs (DAG) are the smallest data-structure with which we can represent a pomset [14]. The DAG G of a pomset p of n elements is the transitive reduction graph of p and therefore has exactly n nodes.

A pomset p thus can be represented as a Hasse diagram (Q, λ, f) with Q a set of states, f an isomorphism between the elements of p and Q and λ a set of transition such that:

$$\forall (a, b) \in p, a > b \iff (f(a), f(b)) \in \lambda$$

See figure 3.1.

To implement such graph in C++, we define a class DAGNode that holds a Letter (unsigned integer) and a vector of pointers to its children. The Pomset class then have the vector of its DAGNodes as attribute.
This structure holds all the needed information of a pomset with a space complexity of $\frac{n^2}{4}+o(n^2)$ bits [14]. However, in addition to the list of node, we decide to keep two lists of references to the minimal and maximal elements in order to simplify basic operations. This representation is ideal for long pomsets (where the order is close to a chain). It is therefore worst for wide pomsets (where the order is close to an anti-chain). The main purpose of those two lists is to prevent the DAG to be run through entirely for each call to the sequential product. With such implementation, it is trivial to determine whether the pomset is empty and to retrieve the lists of maximal and minimal elements. Such object can be constructed from infix notation with a

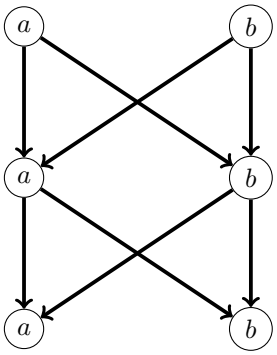

Figure 3.1: Directed Acyclic Graph modelling the pomset $p = (a||b) \bullet (a||b) \bullet (a||b)$.

simple Shunting-Yard algorithm [15].

However, such representation discards the reverse polish notation (RPN) of the pomsets which is needed in order to perform the membership query of a pomset to a bimonoid in the fastest possible way. Thus adding more operations to the query, in order to retrieve the RPN from the DAG. Performing substitution operations on DAGs is also highly non-trivial and would involve re-arranging the whole data-structure. As those are operations that will be done numerous times in our program, it seems interesting to look up for an implementation that would involve shorter queries and substitutions.

3.1.2 AST

Abstract syntax trees (AST) are data-structures that store a synthaxic expression directly in RPN. It is able to store a category of n elements in $2n - 1$ nodes [16]. In such implementation, internal nodes represent operations in $\{\bullet, ||\}$ and leaves represent the elements of the pomset. The RPN notation of the pomset can thus be retrieved with a simple post-order traversal.

See figure 3.2.

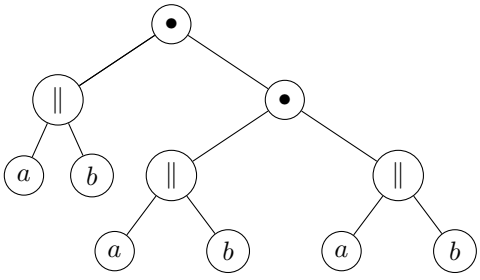

Figure 3.2: Abstract syntax tree modelling the pomset $p = (a||b) \bullet (a||b) \bullet (a||b)$.

To implement such graph in C++, we define a class BinTree that holds a data e (unsigned integer) and two pointers to its children. Such object has always 0 or 2 of its pointers non-null. In

the first case, the node is a leaf and e is a Letter. In the second, it is an internal node and e is either 0 ($\bullet$) or 1 ($||$).

Computing concatenation (resp. parallel product) on AST representing pomsets is trivial. Indeed, for two AST g_1 and g_2 it suffices to create a new AST g_3 with its root being a node with $e = \bullet$ (resp. $||$) and its children being g_1 and g_2. Therefore computing both operations in constant time regardless of the size of the operands.
In a similar manner, substitution $g_1 \ o_\xi \ g_2$ can be done by replacing the leaf.ves holding the letter ξ in g_1 by g_2, resolving in $O(n)$ in the worst case.

In order to always have two equivalent pomsets be the exact same C++ object we "canonise" our AST by recursively stacking all sequence of operations to the right using right bintree rotation. This allows us to compare two pomsets by literally comparing their AST.

As we will see in the following part, storing pomsets as AST is ideal for to query them to bimonoids. With regards to the minimal time complexity of the operations on pomset represented as AST and despite their greater space complexity as DAG, we will assume that we are using such data-structure for the following parts, as it seems to assure better performance.

3.2 Pomset recognisers

3.2.1 Data-structure

We can represent a bimonoid in C++ thanks to a class Bimonoid with a template T holding a vector M of T representing the carrier, two function pointers $concat$ and $parallelise$ of signature $(T, T) \to T$ representing the two composition laws, and a unit of type T.

We can then define the Pomset_rec class holding a Bimonoid with $T = unsigned\ int$, an alphabet Σ (vector of unsigned int), as subset F of M of final state and a function pointer of signature $unsigned\ int \to unsigned\ int$ capable of mapping Letters of the alphabet to elements of M.

We note that the complexity of such data-structure is dependant on the implementation of the internal function pointed to by the bimonoid and can thus vary by a great range. It seems coherent to affirm that the best implementation of these functions would be to read results in a previously filled array, thus implying two more "hidden" arrays of unsigned int of size n^2 essential to this structure.

3.2.2 Membership queries

In order to compute the membership query of a Pomset p represented as an AST to a Pomset_rec A, we need to parse the bintree using the internal functions of the bimonoid. This results to a single element of the bimonoid, as described in Algorithm 1. For a pomset of size n, this algorithm performs exactly $2n - 2$ recursive calls.

Then, we can compute the membership query of a pomset by checking if the element computed by Algorithm 1 is in the set of final element of the Pomset recogniser, as described in Algorithm

2.

Algorithm 1 process_ast(p)

```
if p == null then
    return A.unit
end if
if p is a leaf then
    return A.i(p.data)
else
    l = process_ast(p.left_child)
    r = process_ast(p.right_child)
    if data is • then
        return A.concat(l, r)
    else
        return A.parallelise(l, r)
    end if
end if
```

Algorithm 2 recognise(p)

```
if F.empty() then
    return false
end if
res = process_ast(p)
for state in F do
    if res = state then
        return true
    end if
end for
return false
```

3.3 Observation table

3.3.1 Data-structure

In order to perform the whole L_p^* algorithm of a pomset recogniser Q in C++, we define an Observation_table class. It holds a vector of sub-pomsets S, a vector of contexts E and an 2-dimension array t of booleans such that $t[x][y] = true \iff S[x]\ o_\xi\ E[y] \in L(Q)$.
We also maintain a table S^+ of pomsets in $(S.\Sigma) \cup (S||\Sigma)$ and extend t accordingly in order to not re-compute these values each time we check for closure or associativity.

Thus, for a pomset recogniser Q of n states over an alphabet Σ of k letters, our Observation_table will store $2n + 2nk$ pomsets and a table of $n^2 + 2n^2k$ booleans.

3.3.2 Constructing pomset recogniser

As we need to be able to extract a pomset recogniser from our table, we define a C++ class Hypothesis holding a Pomset_rec rec and a 2-dimension array of boolean t.
We are able to construct an Hypothesis object from an Observation_table as described in Definition 20.

The Hypothesis will duplicate the array t of the Observation_table and store it in order to define the internal functions of the bimonoid as described in Algorithm 3.

Algorithm 3 $concat(q_1, q_2)$

$p_1 = t.S[x]$ such that $t.row(x) = q_1$
$p_2 = t.S[y]$ such that $t.row(y) = q_2$
return $row(p_1 \bullet p_2)$

The parallel product is defined analogously.

The carrier of the bimonoid is defined as the vector of $row(s), s \in S$. The set of final states is defined as the the vector of $row(s), s \in S, t[row(s)][0] = 1$. The mapping function i is defined directly from the alphabet.

Once the Hypothesis is constructed, we can compare it to the teacher as an equivalence query. The teacher shall provide a boolean indicating the result of query as well as a counterexample pomset in the negative cases. This can be done on finite languages by testing all combination of Letters up to the maximal size of a Pomset of the language of the teacher as described in Algorithm 4.

Algorithm 4 $find\_counterexample(H, Q)$

$E = L(Q)$
for $e \in E$ **do**
  if $H.recognise(e)! = Q.recognise(e)$ **then**
    return e
  end if
end for
return null

This method of comparison is naive and does not allow to compare pomset recogniser representing languages of infinite cardinality. However bimonoid isomorphism having never been described, the state of the art does not seem to provide anything more effective. Better methods of comparing two pomset recognisers are discussed in **4.2**.

3.3.3 Checking and fixing closure

Algorithm 5 checks for closure and Algorithm 6 enforces closure on an Observation_table t.

We observe that making the table closed requires all the operations necessary for checking closure. We will therefore, in contrary to the pseudo-given provided by [6], never check for closure and always try to make the table closed at each loop of L_p^*, as this will lead to fewer operations.

Algorithm 5 $is\_closed(t)$

```
for e1 ∈ S+ do
    found = false
    r1 = row(e1)
    for e2 ∈ S do
        r2 = row(e2)
        if r1 == r2 then
            found = true
            break
        end if
    end for
    if !found then
        return false
    end if
end for
return true
```

Algorithm 6 $make\_closed(t)$

```
for e1 ∈ S+ do
    found = false
    r1 = row(e1)
    for e2 ∈ S do
        r2 = row(e2)
        if r1 == r2 then
            found = true
            break
        end if
    end for
    if found then
        S.add(e1)
        return
    end if
end for
```

3.3.4 Checking and fixing associativity

Algorithm 7 checks for γ-associativity and Algorithm 8 enforces γ-associativity on an Observation_table t learning a teacher T.

As for closure, we observe that making the table γ-associative requires all the operations necessary for checking γ-associativity. In a similar manner, we will never check for associativity and always will try to make the table associative directly.

Algorithm 7 $is\_associative(t, \gamma)$

for $s_1, s_2, s_3, s_r, s_l \in S$ **do**
  if $row(s_l)! = row(s_1\ \gamma\ s_2)$ *or* $row(s_r)! = row(s_2\ \gamma\ s_3)$ **then**
    continue
  end if
  $P_1 = s_l\ \gamma\ s_3$
  $P_2 = s_1\ \gamma\ s_r$
  if $row(P_1)! = row(P_2)$ **then**
    return false
  end if
end for
return true

Algorithm 8 $make\_associative(t, \gamma)$

for $s_1, s_2, s_3, s_r, s_l \in S$ **do**
  if $row(s_l)! = row(s_1\ \gamma\ s_2)$ *or* $row(s_r)! = row(s_2\ \gamma\ s_3)$ **then**
    continue
  end if
  $P_1 = s_l\ \gamma\ s_3$
  $P_2 = s_1\ \gamma\ s_r$
  for $i \in E$ **do**
    if $row(P_1)[i]! = row(P_2)[i]$ **then**
      $p_1, p_2, p_3, p_l, e = s_1, s_2, s_3, s_l, i$
      goto $exit\_loop$
    end if
  end for
end for
return
$exit\_loop$:
$B = (p_1\ \gamma\ p2)\ \gamma\ p_3$
if $row(p_l\ \gamma\ p3)[e]! = T.membership\_query(B)$ **then**
  $E.add(e\ o_\xi\ (\xi\ \gamma\ p_3))$
else
  $E.add(e\ o_\xi\ (s_1\ \gamma\ \xi))$
end if

3.4 L_p^*

3.4.1 Minimising counter-example

When an equivalence queries gives a negative response, the teacher hands a counter-example pomset to the table. This pomset is either one that belongs to the language of the teacher and not to the one of the table, or either conversely.

Which counter-example will be given is non-deterministic, there is thus no limit as to its size. We will then try to reduce this pomset to an equivalent minimal pomset, in order to ensure the spacial and temporal complexities of our program. This is done recursively by replacing sub-pomsets inside the counter-example by equivalent elements of S while the obtained pomset still is a counter-example. We implement this function $handle\_counter\_example$ in the exact same way as [6] and therefore do not provide pseudo-code.

3.4.2 The algorithm

Our implementation of L_p^* with a teacher Q and an Observation_table T is given in Algorithm 9.

Algorithm 9 $L_p^*(Q)$

$T.S = \{\epsilon\}$
$T.E = \{\xi\}$
while true **do**
  while $!T.closed$ *or* $!T.associative$ **do**
    $T.make\_closed(t)$
    $T.make\_associative(t, \bullet)$
    $T.make\_associative(t, ||)$
  end while
  $H = T.create\_hypothesis()$
  $c = find\_counter\_example(H, Q)$
  if $c == null$ **then**
    return H
  end if
  $c = T.handle\_counter\_example(c)$
  $T.E.add(c)$
end while

Chapter 4

Further discussions

4.1 Extending L^λ

L^λ, described in [9], is an extension of L^* that uses abstract data-structures instead of an observation table. It works by assigning a specific list of contexts to each prefix instead of computing each prefix/suffix relationship, as some of them are not relevant as to the expressiveness of the target language.

The algorithm maintains a prefix-closed list of words (called short-prefixes) S modelling the equivalent classes of the Myhill-Nerode relationship. It also maintains of partition B of $(S \cup S.\Sigma)$ called the pack of components. It maps to each $b \in B$ a list V_b of suffixes, used to distinguish classes of prefixes.

L^λ goes on by submitting equivalence queries to the teacher and treating counter-example. Each handed counter-example is expanded in order to find more components b of B. Components are then refined in order to find their appropriate distinguishing elements. This allows the algorithm to learn a DFA in finite time, with less membership queries and less total maintained symbols than L^* [9].

Our future work will mainly consist in finding an extension of L^λ to pomset recognisers. We believe that this can be done by replacing the list S of short-prefixes by a list of sub-pomsets and the pack of components maintaining suffixes by one maintaining contexts. The main problematic as to this extension is to find how to compute appropriate contexts for each component, this will probably need another solution that what is done in L_p^*, as we discard the observation table. The optimisation provided by the minimisation of counter-example handed by the teacher in L_p^* seems to be directly adaptable to this new algorithm. It is foreseeable that such an algorithm for pomset recognisers will retain the same complexity advantages as its version on DFA.

4.2 Comparing Bimonoids

As bimonoid isomorphism has never been described, and as our method described in Algorithm 4 is naive, we would like to find an optimal method for such operation, and thus present several leads to resolve this issue.

4.2.1 Probabilistic method

Probabilistic methods of computing isomorphism with random tests might be a solution for such problem. As described in [17], it is possible to generate random tests for finite state machine with geometric distribution in Σ^*. Using a large number of such tests sequentially would tend to increase confidence in our equivalence computation, making this method an interesting indicator. However, it's important to bear in mind that this method is inexact and thus will lead to false positives after a large number of attempts, which could undermine our work.

4.2.2 Algebraic method

As bimonoids are 2-categories [18], formal algebraic methods can be employed in order to compare them. [19] introduces 2-isomorphism, which is an extension of graph isomorphism on 2-categories graphs. This is done by mapping all circuit in the first operand's category graph to one of the other. This operation succeeds iff the two graphs are equivalent. Such method might be a coherent way of deciding equivalence between two bimonoids. However, as the graph isomorphism problem is np-complete, this method is very likely to imply great algorithmic complexity.

4.2.3 W-method

The w-method [17] is a test generation method that produces a test suite of polynomial size used to study the behaviour of a finite state machine. This is done by using certain properties of state covers and characterisation sets and by comparing the behaviour of each equivalence classes of states in both operands. This method has the advantage of being both exact and less resource-intensive than graph isomorphism. We thus believe that such solution should be considered first, both for simplicity and efficiency.

4.3 Random generation of pomset recognisers

As we would like to empirically test our active learning algorithms implementation, it would be handy to be able to randomly generate test cases. We believe that this can be done using random tree automata generation as described in [20]. This method allows generation of trim tree automata and would permit to generate random pomset recognisers if we limit the arity of the transition functions to 0 or 2.

However, as we need the composition laws of the bimonoid to be associative (and commutative for the parallel product), we need to enforce such properties on the transition functions generated by the algorithm. This seems to be doable using rewriting methods such as Knuth-Bendix completion [21]. Potential further work might be to prove that using rewriting methods on such generative algorithm does no undermine the "good" distribution of generated objects.

Chapter 5

Conclusion

We have presented an implementation of an extension of L^* to pomset recognisers. By doing so, we have compared different data-structure for pomsets and have concluded that abstract syntax tree are the best possible objects in this context, in regards to the temporal advantages they offer. We have also described algorithms computing membership query of a pomset to a pomset recogniser and a naive method of comparing two pomset recognisers. The obtained L_p^* written in C++ seems to efficiently replicate the theoretical behaviour of the algorithm while providing some functional improvements. Such algorithm can be used to learn and/or minimise a state machine recognising pomsets.

Our future work will consist in the theoretical and practical extension of L^λ on pomset recognisers, using the already accomplished work on L_p^*. Such extension would permit to reach new levels of performance when learning pomset languages.

Other leads for future work would be to formalise better methods of bimonoid isomorphism and pomset recogniser generation. Solving these problems would be of a great help in our work, as they would allow faster and safer ways of running our algorithms in large test benches, thus increasing our confidence as to the validity of our results.

Chapter 6

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. (pages 5, 13, and 14)

[2] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. (page 5)

[3] Kamal Lodaya and Pascal Weil. A kleene iteration for parallelism. In *Foundations of Software Technology and Theoretical Computer Science*, 1998. (pages 5 and 12)

[4] Tobias Kappé, Paul Brunet, Bas Luttik, Alexandra Silva, and Fabio Zanasi. On series-parallel pomset languages: Rationality, context-freeness and automata. *Journal of Logical and Algebraic Methods in Programming*, 103:130–153, 2019. (page 5)

[5] Amazigh Amrane, Hugo Bazille, Uli Fahrenberg, and Marie Fortin. Logic and languages of higher-dimensional automata, 2024. (page 5)

[6] Gerco van Heerdt, Tobias Kappé, Jurriaan Rot, and Alexandra Silva. *Learning Pomset Automata*, page 510–530. Springer International Publishing, 2021. (pages 5, 14, 15, 16, 22, and 24)

[7] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2008. (page 5)

[8] Kamal Lodaya and Pascal Weil. A kleene iteration for parallelism. In Vikraman Arvind and Ramaswamy Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science, 18th Conference, Chennai, India, December 17-19, 1998, Proceedings*, volume 1530 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 1998. (pages 5 and 11)

[9] Falk Howar and Bernhard Steffen. Active automata learning as black-box search and lazy partition refinement. In Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, editors, *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, volume 13560 of *Lecture Notes in Computer Science*, pages 321–338. Springer, 2022. (pages 6 and 25)

[10] Vaughan Pratt. Modelling concurrency with partial orders. *Internationaljournal of Parallel Programming*, 15, 04 2000. (page 8)

[11] Stephen L. Bloom and Zoltán Ésik. Free shuffle algebras in language varieties. *Theor. Comput. Sci.*, 1996. (page 8)

[12] Kamal Lodaya and Pascal Weil. Rationality in algebras with a series operation. *Inf. Comput.*, 171(2):269–293, 2001. (page 10)

[13] Kamal Lodaya and Pascal Weil. Series-parallel languages and the bounded-width property. *Theor. Comput. Sci.*, 2000. (pages 10 and 12)

[14] J. Ian Munro and Patrick K. Nicholson. Succinct posets. *CoRR*, abs/1204.1957, 2012. (page 17)

[15] Theodore Norvell. Parsing expressions by recursive descent. https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm, 1999. (page 18)

[16] Joel Jones. Abstract syntax tree implementation idioms. *Pattern Languages of Program Design*, 2003. Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003) http://hillside.net/plop/plop2003/papers.html. (page 18)

[17] Dan-Tuong Le. *Quantitative Analysis of Counterexample Generation for Automata Learning*. PhD thesis, Universitätsbibliothek der RWTH Aachen, 2020. (page 26)

[18] J.-M. Maranda. Formal categories. *Canadian Journal of Mathematics*, 17:758–801, 1965. (page 26)

[19] Hassler Whitney. *2-Isomorphic Graphs*, pages 125–134. Birkhäuser Boston, Boston, MA, 1992. (page 26)

[20] Thomas Hanneforth, Andreas Maletti, and Daniel Quernheim. Random generation of nondeterministic finite-state tree automata. *Electronic Proceedings in Theoretical Computer Science*, 134:11–16, November 2013. (page 26)

[21] D. E. Knuth and P. B. Bendix. *Simple Word Problems in Universal Algebras*, pages 342–376. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983. (page 26)