Morphisms and minimisation of weighted automata

Sylvain Lombardy, Jacques Sakarovitch

Morphisms and minimisation of weighted automata

Sylvain Lombardy *

Jacques Sakarovitch †

Abstract. This paper studies the algorithms for the minimisation of weighted automata. It starts with the definition of morphisms — which generalises and unifies the notion of *bisimulation* to the whole class of weighted automata — and the unicity of a minimal quotient for every automaton, obtained by partition refinement.

From a general scheme for the refinement of partitions, two strategies are considered for the computation of the minimal quotient: the *Domain Split* and the *Predecesor Class Split* algorithms. They correspond respectivly to the classical Moore and Hopcroft algorithms for the computation of the minimal quotient of deterministic Boolean automata.

We show that these two strategies yield algorithms with the same quadratic complexity and we study the cases when the second one can be improved in order to achieve a complexity similar to the one of Hopcroft algorithm.

1. Introduction

The main goal of this paper is to report on the design and analysis of algorithms for the computation of the minimal quotient of a weighted automaton.

The *existence* of a minimal deterministic finite automaton, canonically associated with every regular language is one of the basic and fundamental results in the theory of classical finite automata [1]. The problem of the *computation* of this minimal (deterministic) automaton has given rise to an extensive literature, due to the importance of the problem, both from a theoretical and practical point of view. The chapter *Minimisation of automata* of the recently published *Handbook of Automata Theory* [2] is devoted to this question by Jean Berstel and his colleagues [3]. It provides a rich and detailed account on the subject, together with an extensive list of references.

*LaBRI - UMR 5800 - Bordeaux INP - Bordeaux University - CNRS
†IRIF - CNRS/Paris University and Telecom Paris, IPP

In contrast, the problem of minimisation of nondeterministic Boolean, or of weighted, automata is much less documented. The chapter *Algorithms for weighted automata* in the *Handbook of Weighted Automata* [4] deals only with the case of so-called deterministic weighted automata on which the algorithms for deterministic Boolean automata can be generalised. The main reason is probably that the problem of finding a (Boolean) automaton equivalent to a given automaton and with a minimal number of states is untractable and known to be NP-hard and even PSPACE-complete [5]. Of course, the case of weighted automata, with arbitrary weight semiring, is at least as difficult.

There is another way to look at the minimisation of deterministic Boolean automata. The result, the minimal automaton, is obtained by merging states and in this way can be seen as the image of the original automaton by a map that preserves the structure of the computations, a map it is natural to call morphism. The other purpose of this paper, which indeed comes first, is then to set up the definition of *morphisms for arbitrary (finite) weighted automata*, and in particular for nondeterministic Boolean automata. The image of a morphism is a *quotient* and, as in the original case, an automaton has a *unique minimal quotient* (Theorem 3.12). The difference with the original case is that the minimal quotient is not canonically attached to the series or the language realised by the automaton but to the automaton itself.

To tell the truth, this point of view is not completely new. For transition systems for instance, which are automata without initial nor final states, it is common knowledge that the notion of *bisimulation* allows to merge states in order to get a system in which the computations are preserved and the *coarsest bisimulation* yields a minimal system. The notion of bisimulation has also been extended to some classes of weighted automata, for instance *probabilistic automata* [6], or automata with weights in a field or division ring [7]. It is also known that in all these cases the coarsest bisimulation is computed by iterative refinements of set partitions.

We show here that the classical algorithms of partition refinement that are used for deterministic Boolean automata (widely known as Moore and Hopcroft algorithms) may be analysed and abstracted in such a way they readily extend to weighted automata in full generality, without any assumption on the weight semiring nor on the automaton structure. This can be sketched as follows. In a partition refinement algorithm, and at a given step of the procedure, a partition $\mathcal{P}$ of the state set of an automaton, and a class C of $\mathcal{P}$ are considered. There are then two possible strategies for determining a refinement of $\mathcal{P}$. In the first one, the class C itself is split, by considering the labels of the *outgoing transitions* from the different states in C. This is an extension of the Moore algorithm and we call it the *Domain Split Algorithm*. In the second strategy, the class C determines the splitting of classes that contain the origins of the *transitions incoming* to the states in C. We call it the *Predecessor Class Split Algorithm* and it can be seen as inspired by the Hopcroft algorithm.

Although the two strategies yield distinct orderings of the splitting of classes, the two algorithms have many similarities that we describe in this paper. Not only do they have the same time complexity, in $\mathrm{O}(n\,m)$, where n is the number of states of the automaton and m the number of its transitions, but the criterium for distinguishing states — the splitting process — is based on the same state function that we call *signature*. And in both cases, achieving the above mentioned complexity implies that signatures are managed through the same efficient data structure that implements a *weak sort*.

Finally, our analysis allows to describe a condition — which we call *simplifiable signatures* — under which the Predecessor Class Split Algorithm can be tuned in such a way it achieves a better time in $\mathrm{O}(m \log n)$, the Graal set up by Hopcroft algorithm for deterministic Boolean automata (in

which $m = \alpha\, n$, where α is the size of the alphabet).

In conclusion, our study of the minimisation algorithms, with the identification of the concept of signature, allows to reverse the perspective. We have not transformed the known algorithms on deterministic Boolean automata by equipping them with supplementary features that allow to treat weighted automata, we have just expressed them in a way they can deal with all weighted automata and they apply then to deterministic Boolean automata just as they do with any other, not even in a simpler way.

The paper is organised as follows. In Section 2, we begin with the definitions and notation used for weighted automata. We add the definition of a special class of automata, a kind of normalised ones, which we call *augmented automata* and which will be useful for the writing of algorithms. At Section 3, we give two equivalent definitions for morphisms of weighted automata and show the existence of a unique minimal quotient.

In Section 4, we describe a general procedure for the partition refinement, the notion of signature, and both the Domain Split and Predecessor Class Split Algorithms. In the next Section 5, we describe the way to implement the weak sort, how it is used in the two algorithms, and we analyse their complexity, whereas we explain in Section 6 under which conditions and how it is possible to improve the complexity of the latter.

The algorithms and their efficient implementation described in this paper are accessible in the AWALI plateform recently made available to the public [8]. Some experiments presented at Section 7 show that the algorithms behave with their theoretical complexity. These experiments were presented at the CIAA conference in 2018 [9].

2. Weighted automata

In this paper, we deal with (finite) automata over a free monoid A^* with weight in a semiring $\mathbb{K}$, also called $\mathbb{K}$-automata. 'Classical' automata are the $\mathbb{B}$-automata where $\mathbb{B}$ is the Boolean semiring.

Indeed, all what follows apply as well to automata over a monoid M which is not necessarily free, for instance to *transducers* that are automata over the non free monoid $A^* \times B^*$. This generalisation holds as such automata are considered — as far as the constructions and results developped here are concerned — as automata over a free monoid C^*, where C is the set of labels on the transitions of the automaton. Let us note that there exists no theory of quotient that takes into account non trivial relations between labels. The only construction on automata that does (without destroying their structure) is the *circulation of labels* involved for instance in the *synchronisation* of transducers (*cf.* [10, 11]) or as a *preliminary* for the minimisation of sequential transducers (*cf.* [10, 12]).

2.1. Definition and notation

We essentially follow the definitions and notation of [10]. The model of weighted automaton used in this paper is more restricted though, for both theoretical and computational efficiency.

A $\mathbb{K}$-automaton $\mathcal{A}$ over A^* is a directed graph whose vertices are called *states* and whose edges, called *transitions*, are labelled by a pair made of a letter in A and a weight in $\mathbb{K}$, together with an

initial function and a *final function*, both from the set of states to $\mathbb{K}$. The states in the support of the initial function are usually called *initial states*, those in the support of the final function, *final states.* Figure 1 shows a $\mathbb{Z}$-automaton $\mathcal{A}_1$ over $\{a, b\}^*$ with the usual graphical conventions.

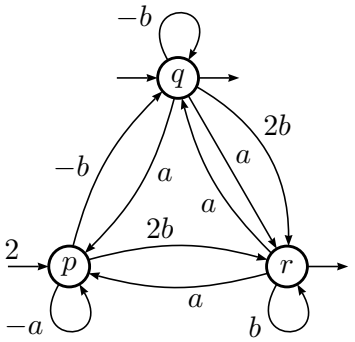

Figure 1. The $\mathbb{Z}$-automaton $\mathcal{A}_1$

Let $\mathcal{A}$ be a $\mathbb{K}$-automaton with set of states Q; we say that Q is the *dimension* of $\mathcal{A}$ and we denote $\mathcal{A}$ by a triple $\mathcal{A} = \langle I, E, T \rangle$ where I and T are vectors of dimension Q with entries in $\mathbb{K}$ that denote the initial function and the final functions and E is the adjacency matrix of $\mathcal{A}$, that is, a $Q \times Q$-matrix whose (p, q) entry is the sum of the weighted labels of the transitions from p to q. We also denote by E the map $E\colon Q \times A \times Q \to \mathbb{K}$ and $E(p, a, q)$ is the weight of the (unique) transition that goes from p to q and that is labelled by a, if it exists (otherwise its value is $0_{\mathbb{K}}$).

Example 2.1.

$$\mathcal{A}_1 = \left\langle \begin{pmatrix} 2 & 1 & 0 \end{pmatrix}, \begin{pmatrix} -a & -b & 2\,b \\ a & -b & a + 2\,b \\ a & a & b \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \right\rangle \qquad \text{and} \qquad E_1(q, b, r) = 2\,.$$

The *behaviour* of a $\mathbb{K}$-automaton $\mathcal{A} = \langle I, E, T \rangle$ is the *series* realised by $\mathcal{A}$ and is denoted by $|\mathcal{A}|$:

$$|\mathcal{A}| = I \cdot E^* \cdot T\,,$$

where $E^* = \sum_{n \in \mathbb{N}} E^n$ (the infinite sum does not bring any problem as our definition insures that the entries of E^n are homogenous polynomials of degree n). Two automata are said to be *equivalent* if their behaviours are equal. If $\mathbb{K} = \mathbb{B}$, $|\mathcal{A}|$ is the *language* accepted by $\mathcal{A}$.

The *future* of a state p of $\mathcal{A}$, denoted by $\mathsf{Fut}_{\mathcal{A}}(p)$, is the series realised by $\mathcal{A}$ when p is taken as the unique initial state, with initial value equal to $1_{\mathbb{K}}$. It holds

$$\mathsf{Fut}_{\mathcal{A}}(p) = I_p \cdot E^* \cdot T\,, \tag{1}$$

where I_p denotes the characteristic (row-) vector of p, that is, the entry of index p is the only non-zero entry and is equal to $1_{\mathbb{K}}$.

2.2. The augmented automaton

The algorithms we describe and study in the following sections classify states of an automaton according to (the labels of) the in- and out-going transitions and (the values of) the initial and final functions. In order to express in the same way the conditions stated on the transitions on one hand and on the initial and final functions on the other, and to compute uniformly with the former and the latter, it is convenient to transform the automaton and to put them in a special form which we call *augmented*.

A $\mathbb{K}$-automaton $\mathcal{A} = \langle\, I, E, T \,\rangle$ of dimension Q over A^* is transformed into its augmented version, denoted by $\widetilde{\mathcal{A}}$, in two steps. First, the alphabet A is supplemented with a new letter \$, which will be used as a left and right marker. We write $A_\$$ for $A_\$ = A \cup \{\$\}$, the *augmented alphabet*. Second, $\mathcal{A}$ is somehow *normalised* by the adjunction of two new states i and t to Q and of transitions that go from i to every initial state p with label \$ and with weight I_p and transitions that go from every final state q to t with label \$ and with weight T_q. As it will be useful to make its state set explicit, we denote $\widetilde{\mathcal{A}}$ by $\widetilde{\mathcal{A}} = \Big(\, Q, i, \widetilde{E}, t \,\Big)$ where $\widetilde{E}$ contains the complete description of $\widetilde{\mathcal{A}}$, as seen in (2).

$$\mathcal{A} = \langle\, I, E, T \,\rangle \qquad \text{and} \qquad \widetilde{\mathcal{A}} = \Big(\, Q, i, \widetilde{E}, t \,\Big) \quad \text{with} \quad \widetilde{E} = \begin{pmatrix} 0 & \boxed{I\$} & 0 \\ \boxed{0} & \boxed{E} & \boxed{T\$} \\ 0 & \boxed{0} & 0 \end{pmatrix} . \tag{2}$$

Finally, the only initial state of $\widetilde{\mathcal{A}}$ is i, with weight $1_{\mathbb{K}}$, and its only final state is t, also with weight $1_{\mathbb{K}}$. If w is in A^*, there is a 1-1 correspondence between the successful computations with label w in $\mathcal{A}$ and the computations with label $\$\, w \,\$$ in $\widetilde{\mathcal{A}}$, hence they are given the same weight by the two automata. Hence, $\big|\widetilde{\mathcal{A}}\big| = \$|\mathcal{A}|\$$.

Example 2.2. (continued)
The adjacency matrix $\widetilde{E}_1$ of the $\mathbb{Z}$-automaton $\widetilde{\mathcal{A}}_1$ is shown on Figure 2. There is no column i nor row t in the table for there is no transitions incoming to i nor transitions outgoing from t.

	p	q	r	t
i	$2\,\$$	$\$$	0	0
p	$-a$	$-b$	$2\,b$	0
q	a	$-b$	$a + 2\,b$	$\$$
r	a	a	b	$\$$

Figure 2. $\widetilde{E}_1$, the adjacency matrix of $\widetilde{\mathcal{A}}_1$

3. Morphisms of weighted automata and minimal quotients

The notion of morphism for deterministic Boolean automata does not raise difficulties and makes consensus. In contrast, the one of morphism for arbitrary Boolean automata and even more for (ar-

bitrary) weighted automata is far more problematic. In most cases, the concept is given other names, most often *simulation* or *bisimulation*, when not more complicated such as *weak bisimulation* or *pure epimorphism*, and definitions that may depend on the case considered and hide their generality.

3.1. Amalgamation matrices and morphisms

We start with the definition of *morphisms* — of *Out-morphisms* indeed as we shall see — by means of the notion of *conjugacy* of automata, as we already did in [10] or [13] and which is the most concise one. We then translate it into a more explicit definition via equivalence that is more suited for computations.

Definition 3.1. Let $\mathcal{A} = \langle\, I, E, T \,\rangle$ and $\mathcal{B} = \langle\, J, F, U \,\rangle$ be two $\mathbb{K}$-automata, of dimension Q and R respectively. We say that $\mathcal{A}$ *is conjugate to* $\mathcal{B}$ *by* X if there exists a $Q \times R$-matrix X with entries in $\mathbb{K}$ such that

$$I \cdot X = J\,, \quad E \cdot X = X \cdot F\,, \quad \text{and} \quad T = X \cdot U\,. \tag{3}$$

The matrix X is the *transfer matrix* of the conjugacy and we write $\mathcal{A} \xRightarrow{X} \mathcal{B}$.

If $\mathcal{A}$ is conjugate to $\mathcal{B}$, then, for every n, the sequence of equalities holds:

$$I \cdot E^n \cdot T = I \cdot E^n \cdot X \cdot U = I \cdot E^{n-1} \cdot X \cdot F \cdot U = \ldots = I \cdot X \cdot F^n \cdot U = J \cdot F^n \cdot U\,,$$

from which $I \cdot E^* \cdot T = J \cdot F^* \cdot U$ directly follows. And this is stated as the following.

Proposition 3.2. If $\mathcal{A}$ is conjugate to $\mathcal{B}$, then $\mathcal{A}$ and $\mathcal{B}$ are equivalent. □

The conjugacy relation is not an equivalence relation as it is reflexive and transitive but not symmetric.

A *surjective* map $\varphi\colon Q \to R$ is completely described by the $Q \times R$-matrix X_φ whose (q, r)-th entry is 1 if $\varphi(q) = r$, and 0 otherwise. Since φ is a map, every row of X_φ contains exactly one 1 and since φ is surjective, every column of X_φ contains at least one 1. Such a matrix is called an *amalgamation matrix* in the setting of symbolic dynamics ([14]). By convention, if we deal with $\mathbb{K}$-automata, an amalgamation matrix is silently assumed to be a $\mathbb{K}$-matrix, that is, the null entries are equal to $0_\mathbb{K}$ and the non-zero entries to $1_\mathbb{K}$.

Definition 3.3. Let $\mathcal{A} = \langle\, I, E, T \,\rangle$ and $\mathcal{B} = \langle\, J, F, U \,\rangle$ be two $\mathbb{K}$-automata of dimension Q and R respectively. A surjective map $\varphi\colon Q \to R$ is a *morphism* (from $\mathcal{A}$ onto $\mathcal{B}$) if $\mathcal{A}$ is conjugate to $\mathcal{B}$ by X_φ: $\mathcal{A} \xRightarrow{X_\varphi} \mathcal{B}$, that is,

$$I \cdot X_\varphi = J\,, \quad E \cdot X_\varphi = X_\varphi \cdot F\,, \quad \text{and} \quad T = X_\varphi \cdot U\,, \tag{4}$$

and we write $\varphi\colon \mathcal{A} \to \mathcal{B}$.

We also say that $\mathcal{B}$ is a *quotient* of $\mathcal{A}$, if there exists a morphism $\varphi\colon \mathcal{A} \to \mathcal{B}$.

It is straightforward that the composition of two morphisms is a morphism. And by Proposition 3.2, any quotient of $\mathcal{A}$ is equivalent to $\mathcal{A}$.

Definition 3.3 may be given a form which will probably be more eloquent to many than a mere matrix equation, and which makes clear that the automaton $\mathcal{B}$, and its state set R, are immaterial and that the conditions are upon $\mathcal{A}$ and the map equivalence of φ only. Above all, it will be the one used in the design of the algorithms to come. We describe it in the next subsection.

3.2. Morphisms and congruences

We start with some definitions and notation to deal with equivalences. An *equivalence* on a set Q is a *partition* of Q, that is, a set of non-empty disjoint subsets of Q, called *classes*, whose union is equal to Q. If $\mathcal{P}$ is an equivalence on Q, we denote by $\mathcal{P}$ both the *set* of classes in the partition as well as the *relation* on Q determined by the partition, that is, for every pair (p, q) of elements of Q, $p \,\mathcal{P}\, q$ if and only p and q belong to a same class C of $\mathcal{P}$.

Equivalences and surjective maps are indeed the same objects, seen from a slightly different perspective. An equivalence $\mathcal{P}$ on Q determines the surjective map that sends every q in Q onto its class C modulo $\mathcal{P}$ and conversely every surjective map φ determines the map equivalence. In both cases, they are represented by amalgation matrices.

Let us introduce a last definition. Let $\mathcal{P}$ be an equivalence on Q and $X_{\mathcal{P}}$ its amalgamation matrix. From $X_{\mathcal{P}}$ we construct a *selection matrix* $Y_{\mathcal{P}}$ by transposing $X_{\mathcal{P}}$ and by zeroing some of its non-zero entries in such a way that $Y_{\mathcal{P}}$ is row-monomial, with *exactly one* 1 per row. A matrix $Y_{\mathcal{P}}$ is not uniquely determined by $\mathcal{P}$ but also depends on the choice of the entry which is kept equal to 1, that is, of a 'representative' in each class of $\mathcal{P}$.

Definition 3.4. Let $\mathcal{A}$ be an automaton of dimension Q. We call *congruence* on Q the map equivalence of a morphism φ from $\mathcal{A}$ onto an automaton $\mathcal{B}$.

Proposition 3.5. Let $\mathcal{A} = \langle\, I, E, T \,\rangle$ be a $\mathbb{K}$-automaton over A^* of dimension Q. An equivalence $\mathcal{P}$ on Q is a *congruence* if and only if the following holds:

$$\forall p, q \in Q \qquad p \,\mathcal{P}\, q \quad \Longrightarrow \quad \forall a \in A\,,\ \forall C \in \mathcal{P} \quad \sum_{r \in C} E(p, a, r) = \sum_{r \in C} E(q, a, r)\,, \tag{5}$$

$$\forall p, q \in Q \qquad p \,\mathcal{P}\, q \quad \Longrightarrow \quad T_p = T_q\,. \tag{6}$$

Proof:
The condition is necessary. Let $\mathcal{B} = \langle\, J, F, U \,\rangle$ be a quotient of $\mathcal{A}$, $\varphi\colon \mathcal{A} \to \mathcal{B}$ a morphism, and $\mathcal{P}$ its map equivalence. The multiplication of E *on the right* by X_φ amounts to add together the *columns* of E whose indices are sent to a same element by φ, that is, indices which are in a same class of $\mathcal{P}$. The entries of the $Q \times \mathcal{P}$-matrix $G = E \cdot X_\varphi$ are:

$$\forall p \in Q\,,\ \forall C \in \mathcal{P} \qquad G_{p,C} = \sum_{r \in C} E_{p,r}\,. \tag{7}$$

Every $G_{p,C}$ is a linear combination of letters of A: $G_{p,C} = \sum_{a\in A} G(p,a,C)\,a$ and (7) may be rewritten as

$$\forall p \in Q\,,\ \forall C \in \mathcal{P}\,,\ \forall a \in A \qquad G(p,a,C) = \sum_{r\in C} E(p,a,r)\,. \tag{8}$$

On the other hand, the multiplication of F, *on the left*, by X_φ yields a matrix whose *rows* with indices in the same class modulo $\mathcal{P}$ are equal. Hence the equality $E\cdot X_\varphi = X_\varphi \cdot F$ implies (5). For the same reason, $T = X_\varphi \cdot U$ implies (6).

The condition is sufficient. Let $\mathcal{P}$ be an equivalence on Q such that (5) and (6) hold, $X_\mathcal{P}$ its amalgamation matrix, and $Y_\mathcal{P}$ a selection matrix.

The entries of T are indexed by Q, and $Y_\mathcal{P}\cdot T$ is a (column-) vector of dimension $\mathcal{P}$ obtained by picking one entry of T in each class C modulo $\mathcal{P}$. The vector $X_\mathcal{P}\cdot(Y_\mathcal{P}\cdot T)$, of dimension Q, is obtained by replicating, for every C in $\mathcal{P}$, the entry of $Y_\mathcal{P}\cdot T$ indexed by C for each p of Q in C. Since (6) expresses that all entries of T indexed by states in a same class are equal, $T = X_\mathcal{P}\cdot(Y_\mathcal{P}\cdot T)$ holds.

In the same way, the rows (of dimension $\mathcal{P}$) of $E\cdot X_\mathcal{P}$ are indexed by Q, and $Y_\mathcal{P}\cdot E\cdot X_\mathcal{P}$ is a $\mathcal{P}\times\mathcal{P}$-matrix obtained by picking one row of $E\cdot X_\mathcal{P}$ in each class C modulo $\mathcal{P}$. The matrix $X_\mathcal{P}\cdot(Y_\mathcal{P}\cdot E\cdot X_\mathcal{P})$, of dimension $Q\times\mathcal{P}$, is obtained by replicating, for every C in $\mathcal{P}$, the row of $Y_\mathcal{P}\cdot E\cdot X_\mathcal{P}$ indexed by C for each p of Q in C. Since (5) expresses that all rows of $E\cdot X_\mathcal{P}$ indexed by states in a same class are equal, $E\cdot X_\mathcal{P} = X_\mathcal{P}\cdot(Y_\mathcal{P}\cdot E\cdot X_\mathcal{P})$ holds.

Equations (5) and (6) express then that $\mathcal{A}$ is conjugate by $X_\mathcal{P}$ to the automaton

$$\langle\, I\cdot X_\mathcal{P},\ Y_\mathcal{P}\cdot E\cdot X_\mathcal{P},\ Y_\mathcal{P}\cdot T\,\rangle\ ,$$

hence the map $\varphi\colon Q \to \mathcal{P}$ is a morphism and $\mathcal{P}$, its map equivalence, is a congruence. □

Remark 3.6. As mentioned in the introduction, morphisms have a close relationship with *bisimulations*. One can even say it is the same notion expressed differently, for those weighted automata for which bisimulations have been defined: bisimulations for *transition systems* which are Boolean automata without initial and final states [15, 16], for *probabilistic systems* [17, 6], linear bisimulations when the weight semiring is a field [7], *etc.* and when it is freed from so-called *internal actions*.

Indeed an equivalence relationon the state set of an automaton $\mathcal{A}$ is a bisimulation relation if and only if it is a congruence. And a state of $\mathcal{A}$ is in bisimulation with its image in any quotient of $\mathcal{A}$.

Example 3.7. (continued)
Let $\mathcal{A}_1 = \langle\, I_1, E_1, T_1\,\rangle$ of dimension $Q_1 = \{p,q,r\}$:

$$\mathcal{A}_1 = \left\langle \begin{pmatrix} 2 & 1 & 0 \end{pmatrix}, \begin{pmatrix} -a & -b & 2\,b \\ a & -b & a+2\,b \\ a & a & b \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \right\rangle .$$

It is easily seen that if we add the columns q and r of E_1 we get a matrix whose rows q and r are equal; moreover $T_{1_q} = T_{1_r}$. Hence $\{\{p\},\{q,r\}\}$ is a congruence of Q_1.

Since it is characteristic, we could have taken just as well the statement of Proposition 3.5 as the definition of a congruence or a morphism. And all the more in this paper where it is the property which is uniformly called. But in view of further references to this paper, we prefer to put Definition 3.3 in front, for its mathematical efficiency (*e.g.* [18]).

3.3. Further properties of morphisms and congruences

Proposition 3.8. Let $\mathcal{A} = \langle I, E, T \rangle$ be a $\mathbb{K}$-automaton over A^* of dimension Q and $\mathcal{P}$ a congruence on Q. If two states p and q are equivalent modulo $\mathcal{P}$, then $\mathsf{Fut}_{\mathcal{A}}(p) = \mathsf{Fut}_{\mathcal{A}}(q)$.

Proof:
With the notation of Section 2, $\mathsf{Fut}_{\mathcal{A}}(p) = I_p \cdot E^* \cdot T$ and with the same computation as above, $\mathsf{Fut}_{\mathcal{A}}(p) = I_p \cdot X_{\mathcal{P}} \cdot F^* \cdot U$. Accordingly, $\mathsf{Fut}_{\mathcal{A}}(q) = I_q \cdot X_{\mathcal{P}} \cdot F^* \cdot U$. And if p and q are equivalent modulo $\mathcal{P}$, then $I_p \cdot X_{\mathcal{P}} = I_q \cdot X_{\mathcal{P}}$. □

Remark 3.9. Definition 3.3 gives a notion of morphism that is *directed* as the one of conjugacy is: E is multiplied *on the right* by X in (3). This is even more obvious with the statements of Proposition 3.5 or of Proposition 3.8.

Thus morphisms could, or should, be called *Out-morphisms* as it refers to *out-going* transitions. The same map φ would be an *In-morphism* if $\mathcal{B}$ is conjugate to $\mathcal{A}$ by X_φ^{t} (the transpose of X_φ).

In this work, we deal with Out-morphisms only, which we simply call morphisms. All statements can be dualised and transformed accordingly for In-morphisms.

Remark 3.10. *The case of Boolean automata.* Let $\mathcal{A} = \langle I, E, T \rangle$ and $\mathcal{B} = \langle J, F, U \rangle$ be two *Boolean* automata over A^* of dimension Q and R respectively and $\varphi\colon Q \to R$ a morphism from $\mathcal{A}$ to $\mathcal{B}$. In this special case, no weight is really involved: a transition exists or not, a state is initial or not, final or not. Moreover, I and J can also be seen as subsets of Q, J and U as subsets of R, E can also be seen as a subset of $Q \times A \times Q$, F as a subset of $R \times A \times R$. Definition 3.3 can then be rewritten in the following way: $I \cdot X_\varphi = J$ and $T = X_\varphi \cdot U$ translate into

$$\text{(i)} \qquad \varphi(I) = J \qquad\qquad \text{and} \qquad\qquad \text{(ii)} \qquad T = \varphi^{-1}(U)\,,$$

whereas $E \cdot X_\varphi = X_\varphi \cdot F$ yields

$$\text{(iii)} \quad \forall a \in A\,,\ \forall p, q \in Q \qquad (p, a, q) \in E \implies (\varphi(p), a, \varphi(q)) \in F \qquad \text{and}$$
$$\text{(iv)} \quad \forall a \in A\,,\ \forall p \in Q\,,\ \forall s \in R \qquad (\varphi(p), a, s) \in F \implies \exists q \in \varphi^{-1}(s) \quad (p, a, q) \in E\,.$$

Compared with previous terminology, morphisms of Boolean automata as we have just defined them are what we have called *locally surjective Out-morphisms* in [10] and subsequent works. In the case of *deterministic* Boolean automata, our definition coincide with the classical notion of morphisms of automata, when it is used.

Remark 3.11. *The case of augmented automata.* If $\varphi\colon \mathcal{A} \to \mathcal{B}$ is a morphism, then $\widetilde{\varphi}\colon \widetilde{\mathcal{A}} \to \widetilde{\mathcal{B}}$ is also a morphism, where $\widetilde{\mathcal{A}} = \left(Q, i, \widetilde{E}, t\right)$, $\widetilde{\mathcal{B}} = \left(R, j, \widetilde{F}, u\right)$, and $\widetilde{\varphi}(i) = j$, $\widetilde{\varphi}(t) = u$, and $\widetilde{\varphi}(q) = \varphi(q)$ for all q in Q. This amounts to say that $X_{\widetilde{\varphi}}$ has the form

$$X_{\widetilde{\varphi}} = \begin{pmatrix} 1 & \boxed{0} & 0 \\ \boxed{0} & \boxed{X_\varphi} & \boxed{0} \\ 0 & \boxed{0} & 1 \end{pmatrix}, \tag{9}$$

a matrix which we rather denote by $\widetilde{X_\varphi}$.

Conversely, if $\widetilde{\mathcal{A}} = \left(Q, i, \widetilde{E}, t\right)$ is the augmented automaton of $\mathcal{A} = \langle\, I, E, T \,\rangle$, the amalgamation matrix of *any morphism* of $\widetilde{\mathcal{A}}$ is of the form (9) and corresponds to a morphism of $\mathcal{A}$. In particular, if $\mathcal{A} = (\, Q, i, E, t\,)$ is an augmented $\mathbb{K}$-automaton, an equivalence $\mathcal{P}$ on Q is a *congruence on* $\mathcal{A}$ if and only if

$$\{i\} \in \mathcal{P}\,, \qquad \{t\} \in \mathcal{P}\,, \qquad \text{and} \tag{10}$$

$$\forall p, q \quad p\,\mathcal{P}\,q \quad \Longrightarrow \quad \forall a \in A_{\$}\,,\ \forall C \in \mathcal{P} \quad \sum_{r \in C} E(p, a, r) = \sum_{r \in C} E(q, a, r)\,. \tag{11}$$

The virtue of adding the marker \$ and considering augmented rather than arbitrary automata is that the sole equation (11) expresses conditions described by both equations (5) and (6). And it is the way that the property of being a congruence will be tested by the algorithms described in the sequel.

3.4. Minimal quotients

With Definition 3.3, we have defined the quotients of a $\mathbb{K}$-automaton $\mathcal{A}$. The following proposition states that every $\mathbb{K}$-automaton has a *minimal quotient*.

Theorem 3.12. Among all quotients of a $\mathbb{K}$-automaton $\mathcal{A}$, there exists one, unique up to isomorphism, which has a minimal number of states and which is a quotient of every quotient of $\mathcal{A}$.

Once again, it is convenient to express this property in terms of congruences, which focuses on the automaton itself rather than on its images. It starts with the definition of an order on equivalences, or partitions, on a set Q.

An equivalence $\mathcal{R}$ is *coarser* than an equivalence $\mathcal{P}$ if, for every C in $\mathcal{P}$, there exists D in $\mathcal{R}$ such that C is a subset of D. It follows that every class of $\mathcal{R}$ is the union of classes of $\mathcal{P}$. The equivalences on a set Q, ordered by this inclusion relation, form a lattice, with the identity — where every class is a singleton — at the top and the universal relation — with only one class that contains all elements of Q — at the bottom. The following is then a statement equivalent to Theorem 3.12.

Theorem 3.13. Every $\mathbb{K}$-automaton has a unique coarsest congruence.

Indeed, the equivalence map of the morphism onto the minimal quotient is the coarsest congruence and the coarsest congruence is the equivalence map of the morphism onto the minimal quotient.

Remark 3.14. It is important to stress once again that the minimal quotient of $\mathcal{A}$ is associated with $\mathcal{A}$, *and not* with the series realised by $\mathcal{A}$. *It is not* necessarily *the smallest* (in terms of number of states) $\mathbb{K}$-automaton that realises the series, *it is not* a canonical automaton associated with it.

For instance, the minimal quotient of a Boolean automaton is not necessary the smallest automaton for the accepted language. On the other hand, the minimal quotient of a deterministic Boolean automaton is the *minimal (deterministic) automaton* of the accepted language (which may well be larger than another Boolean automaton accepting the same language) that is, the notion we have defined for all (possibly weighted) automata coincides with the classical one in the case of deterministic Boolean automata.

Proof:
[Proof of Theorem 3.12]

Let $\mathcal{A} = \langle I, E, T \rangle$ be a $\mathbb{K}$-automaton of dimension Q. The *proof of the existence* of a coarsest congruence on $\mathcal{A}$ goes downward, so to speak, that is, we start from the identity on Q which is a congruence. Let $\varphi\colon Q \to R$ and $\psi\colon Q \to P$ be two morphisms. In order to prove the existence of a minimal quotient (or of a coarsest congruence) it suffices to verify that the lower bound of the map equivalences of φ and ψ is a congruence. Let $\varphi'\colon R \to S$ and $\psi'\colon P \to S$ such that $\omega = \varphi \vee \psi = \varphi' \circ \varphi = \psi' \circ \psi$. Hence

$$E \cdot X_\omega = E \cdot X_\varphi \cdot X_{\varphi'} = E \cdot X_\psi \cdot X_{\psi'} \,. \tag{12}$$

By definition, two states p and r of Q are equivalent modulo ω, which we write $p \,\omega\, r$, if and only if there exists a sequence $p = q_0, q_1, \ldots, q_{2n} = r$ of states of Q such that $q_{2i} \,\varphi\, q_{2i+1}$ and $q_{2i+} \,\psi\, q_{2i+2}$ for $0 \leqslant i \leqslant n-1$. Rows of indices q_{2i} and q_{2i+1} of $E \cdot X_\varphi$, and then of $E \cdot X_\varphi \cdot X_{\varphi'}$, are equal, rows of indices q_{2i+1} and q_{2i+2} of $E \cdot X_\psi$, and then of $E \cdot X_\psi \cdot X_{\psi'}$, are equal, hence rows of indices p and r of $E \cdot X_\omega$ are equal. For the same reason, $T_p = T_r$ and ω is a congruence. □

Remark 3.15. In the case of Boolean automata also, the characterisation of bisimulation follows from Remark 3.6.

Proposition 3.16. Two $\mathbb{K}$-automata are bisimilar if and only if their minimal quotients are isomorphic. □

The minimal quotient not only exists but is also effectively, and efficiently computable, as we see in the next sections.

4. The computation of the minimal quotient

To describe the algorithms computing the minimal quotient of a weighted automaton, it is convenient to consider the augmented automaton. From now on, every automaton is therefore augmented, the alphabet is $A_{\$}$, and the states which are different from the initial or final states are called *true states*.

4.1. Refinement algorithms and signatures

The principle of algorithms which are studied in this paper is partition refinement. The algorithms start with the coarsest equivalence (where all true states are gathered in one class), and split classes which are inconsistent with the constraints of a congruence. In order to split classes, we use a criterion on states of $\mathcal{A}$, which we call *signature*, and which, given a partition, tells if two states in a same class should be separated in a congruence of $\mathcal{A}$. This notion of signatures has been used in [19] for the minimisation of *incomplete* deterministic Boolean automata, the 'first' example for which the classical minimisation algorithms for complete deterministic Boolean automata have to be adapted.

The signature is first defined with respect to a class.

Definition 4.1. The *signature* of a state p of a $\mathbb{K}$-automaton $\mathcal{A} = \langle\, Q, i, E, t \,\rangle$ with respect to a subset D of Q is *the map* $\mathit{sig}\,[p, D]$ from $A_\$$ to $\mathbb{K}$, defined by:

$$\mathit{sig}\,[p, D](a) = \sum_{q \in D} E(p, a, q) \,.$$

For every state p and every subset D, the domain of $\mathit{sig}\,[p, D]$ is the set of labels of transitions from p to some state of D. In particular, if a letter a does not belong to the domain of $\mathit{sig}\,[p, D]$, then $\mathit{sig}\,[p, D](a) = 0$. When we want to explicitly describe $\mathit{sig}\,[p, D]$, we write it as a set of elements of the form $a \mapsto k$, where a is in the domain of $\mathit{sig}\,[p, D]$ and $\mathit{sig}\,[p, D](a) = k$.

Two states p and q cannot be equivalent with respect to some congruence $\mathcal{P}$ of Q if their signatures differ for some class D of Q. In order to compare states, it can be useful to consider the *global* signature which is the aggregation of signatures with respect to all classes of the partition. That is, for a given partition $\mathcal{P}$ and for every state p,

$$\mathit{GSig}[p] = \{(D, a) \mapsto k \mid D \text{ class of } \mathcal{P}, a \mapsto k \in \mathit{sig}\,[p, D]\} \,.$$

4.2. The Protoalgorithm

The computation of the coarsest congruence of an automaton $\mathcal{A} = \langle\, Q, i, E, t \,\rangle$ *goes upward*. It starts with $\mathcal{P}_0 = \big\{\{i\}, Q, \{t\}\big\}$, the coarsest possible equivalence. Every step of the algorithm splits some classes of the current partition, yielding an equivalence which is *thiner*, *higher* in the lattice of equivalences of $Q \cup \{i, t\}$.

It follows from Proposition 3.5 that a partition $\mathcal{P}$ is a congruence if and only if

$$\forall C \in \mathcal{P} \,,\ \forall p, q \in C \,,\ \mathit{GSig}[p] = \mathit{GSig}[q] \,.$$

Thus $\mathcal{P}$ is a congruence if and only if, for every pair (C, D) of classes of $\mathcal{P}$, all states p in C have the same signature with respect to D. A pair (C, D) for which this property is not satisfied is called a *splitting pair*. The equivalence on C induced by the signature with respect to D, called *the split of* C *by* D and denoted by $\mathit{split}[C, D]$, can be computed:

$$\forall p, q \in C \qquad \mathit{split}[C, D](p) = \mathit{split}[C, D](q) \quad \Longleftrightarrow \quad \mathit{sig}\,[p, D] = \mathit{sig}\,[q, D] \,.$$

The split of class C by class D of $\mathcal{P}$ leads to a new equivalence on Q: $\mathcal{P} \wedge \mathit{split}[C, D]$, and the protoalgorithm runs as follows:

$\mathcal{P} := \mathcal{P}_0$
while there exists a splitting pair (C, D) in $\mathcal{P}$
    $\mathcal{P} := \mathcal{P} \wedge \mathit{split}[C, D]$

When there are no more splitting pairs, the current equivalence is a congruence.

Proposition 4.2. At every iteration of the protoalgorithm, the equivalence $\mathcal{P}$ is coarser than or equal to the coarsest congruence.

Proof:
In the initial partition $\mathcal{P}_0$, all true states are in the same class, thus $\mathcal{P}_0$ is coarser than or equal to any congruence.

Let $\mathcal{P}$ be a partition computed at one iteration; we assume that $\mathcal{P}$ is coarser than or equal to the coarsest congruence $\mathcal{C}$, and we consider $\mathcal{P}'$ computed at the next iteration. There exists a splitting pair (C, D), such that $\mathcal{P}' = \mathcal{P} \wedge \mathit{split}[C, D]$. If $\mathcal{P}'$ is not coarser than or equal to $\mathcal{C}$, then there exists C_1 and C_2 in $\mathit{split}[C, D]$, p_1 in C_1 and p_2 in C_2 which belongs to the same class in $\mathcal{C}$. Since $\mathcal{P}$ is coarser than or equal to $\mathcal{C}$, D is the union of some classes (D_i) in $\mathcal{C}$; p_1 and p_2 are $\mathcal{C}$-equivalent, hence, for every i,

$$\mathit{sig}\,[p_1, D_i] = \mathit{sig}\,[p_2, D_i]\,. \tag{13}$$

Therefore, $\mathit{sig}\,[p_1, D] = \mathit{sig}\,[p_2, D]$, which is in contradiction with the fact that p_1 and p_2 are in different classes after the split of C with respect to D. Thus $\mathcal{P}'$ is coarser than or equal to $\mathcal{C}$ and, by induction, the proposition holds. □

Corollary 4.3. At the end of the protoalgorithm, the equivalence $\mathcal{P}$ is equal to the coarsest congruence. □

The procedure described by the protoalgorithm is not a true algorithm, in the sense that, in particular, it does not tell how to find a splitting pair, nor how to implement the function *split* to make it efficient. The main difference between the two algorithms described in the sequel is the selection of the splitting pairs.

- the first algorithm iterates over classes C and considers in the same iteration all the pairs (C, D) where D is any class which contains some *successors* of states of C. This leads to split $\mathcal{C}$ in classes which are consistent with every class D. We call it the *Domain Split Algorithm* (DSA for short).

- the second algorithm iterates over classes D and considers in the same iteration all the pairs (C, D) where C is any class which contains some *predecessors* of states of D. This leads to split several classes in such a way that all classes become consistent with the class D. We call it the Predecessor Class Split Algorithm (PCSA for short)

4.3. The Domain Split Algorithm

The Domain Split Algorithm is an extension of the classical Moore algorithm [20] for the minimisation of deterministic Boolean automata. At each iteration of this algorithm, a class C is processed. The *global signature* of every state of C with respect to the current partition is computed, and C is split accordingly. Compared to the protoalgorithm, the Domain Split Algorithm amounts therefore to consider at the same time all the pairs (C, D), where C is the current class, and D is any class of the current partition (actually, only classes which contain some successors of states of C are considered). Notice that it is mandatory to recompute signatures at each iteration, since they depend on the partition which is changing.

At the beginning of each iteration, the current class C is extracted from a queue. At the beginning of the algorithm, the partition $\mathcal{P}_0$ is $\{\{i\}, Q, \{t\}\}$, and Q is inserted in the queue. At the end of each iteration, the classes obtained from the split of C — or C itself if it has not been split — are inserted into the queue, except the singletons that cannot be split.

To insure termination, iterations are gathered to form rounds. A round ends when all classes that were inside the queue at the beginning of the round have been extracted. Hence the number of iterations in a round is equal to the number of classes which are in the queue at the beginning of the round. Notice that, for every state p, there is at most one class containing p that is processed during a round. If there is no split during a round, the algorithm has checked that the partition is a congruence and halts.

Otherwise, the partition is strictly refined and the number of classes strictly increases. At the beginning of the algorithm, there are three classes, and the maximal number of classes is $n+2$, where n is the number of true states of the automaton. Hence, including the last round, there are at most n rounds, and, for every state, at most n global signatures are computed.

Example 4.4. (continued)
On the automaton $\mathcal{A}_1$, the partition $\mathcal{P}_0$ is initialised with $D_1 = \{i\}$, $D_2 = \{p, q, r\}$, $D_3 = \{t\}$. Class D_2 is put into the queue (the other classes are singletons and cannot be split). For every state of D_2 and for every class D, one can compute the signature of this state with respect to D. For instance, the signature of p with respect to D_2 is

$$\begin{aligned} \mathit{sig}\,[p, D_2](a) &= E(p,a,p) + E(p,a,q) + E(p,a,r) = -1 + 0 + 0 = -1\,, \\ \mathit{sig}\,[p, D_2](b) &= E(p,b,p) + E(p,b,q) + E(p,b,r) = 0 + -1 + 2 = 1\,. \end{aligned}$$

From the signature with respect to each class, the global signature of the state can be computed.

The global signature (with respect to $\mathcal{P}_0$) of states in D_2 is:

$$\begin{aligned} \mathit{GSig}[p] &= \{(D_2, a) \mapsto -1, (D_2, b) \mapsto 1\}\,, \\ \mathit{GSig}[q] &= \{(D_2, a) \mapsto 2, (D_2, b) \mapsto 1, (D_3, \$) \mapsto 1\}\,, \\ \mathit{GSig}[r] &= \{(D_2, a) \mapsto 2, (D_2, b) \mapsto 1, (D_3, \$) \mapsto 1\}\,. \end{aligned}$$

States q and r share the same global signature which is different from the global signature of p. The class D_2 is then split into two classes $D_{21} = \{p\}$ and $D_{22} = \{q, r\}$ and the new partition is

$\mathcal{P}_1 = \{\{i\}, \{t\}, \{p\}, \{q, r\}\}$. The new round starts and the class D_{22} is put in the queue. The global signature of states (with respect to $\mathcal{P}_1$) in D_{22} is:

$$\mathit{GSig}[q] = \{(D_{21}, a) \mapsto 1, (D_{22}, a) \mapsto 1, (D_{22}, b) \mapsto 1, (D_3, \$) \mapsto 1\}$$
$$\mathit{GSig}[r] = \{(D_{21}, a) \mapsto 1, (D_{22}, a) \mapsto 1, (D_{22}, b) \mapsto 1, (D_3, \$) \mapsto 1\}$$

Both states have the same global signature, thus the class is not split. The round ends without any splitting, hence the current partition $\mathcal{P}_1$ is a congruence.

4.4. Predecessor Class Split Algorithm

The Predecessor Class Split Algorithm is *inspired* by the Hopcroft algorithm for the minimisation of deterministic Boolean automata. At each iteration of this algorithm, a class D is processed. For every state p which is a predecessor of some states of D, the signature of p with respect to D is computed, and C is split accordingly. Compared to the protoalgorithm, PCSA amounts therefore to consider at the same time all the pairs (C, D), where D is the current class, and C is any class of the current partition (actually, only classes which contain some predecessors of states of D are considered).

Like the DSA, the PCSA uses a queue to schedule the process of classes. At the beginning of the algorithm, every class of $\mathcal{P}_0$ is put into the queue, except $\{i\}$ since there is no predecessor of i. If a class C is split, every new class is put in the queue; if C was in the queue, it is replaced by its subclasses. The algorithm halts when the queue is empty. Actually, when the partition is a congruence, every class extracted from the queue does not induce any splitting, hence there is no more insertion of classes.

Apart from the initialisation of the queue, every class which is inserted in the queue comes from a split. Hence, for every state p, the number of times that a class containing p is extracted from the queue is at most equal to n, where n is the number of true states of the automaton. Notice that there is no notion of rounds in the PCSA.

Example 4.5. (continued)
On the automaton $\mathcal{A}_1$, the equivalence is initialised with $D_1 = \{i\}$, $D_2 = \{p, q, r\}$, $D_3 = \{t\}$. Classes D_2 and D_3 are put in the queue (i has no predecessor, thus D_1 can not split any class). Assume that D_2 is considered first. The signatures of predecessors of states in D_2 are:

$$\mathit{sig}\,[i, D_2] = \{\$ \mapsto 3\}\,, \qquad \mathit{sig}\,[p, D_2] = \{a \mapsto -1, b \mapsto 1\}\,,$$
$$\mathit{sig}\,[q, D_2] = \{a \mapsto 2, b \mapsto 1\}\,, \qquad \mathit{sig}\,[r, D_2] = \{a \mapsto 2, b \mapsto 1\}\,.$$

All states in the class $D_1 = \{i\}$ have the same signature (D_1 is a singleton!). In the class D_2, every state is met, but p has a signature which is different from the signature common to q and r. Hence D_2 is split into two classes $D_{21} = \{p\}$ and $D_{22} = \{q, r\}$ which are inserted in the queue.

The next class which is extracted from the queue is D_3, and the signatures of predecessors of states in D_3 are:

$$\mathit{sig}\,[q, D_3] = \{\$ \mapsto 1\}\,, \qquad \mathit{sig}\,[r, D_3] = \{\$ \mapsto 1\}\,.$$

Both states have the same signature and there is no other state in D_{22}, thus there is no split.

The class processed in the next iteration is D_{21}, and the signatures of predecessors of states in D_{21} are:

$$\begin{aligned} \mathit{sig}\,[i, D_{21}] &= \{\$ \mapsto 2\}\,, & \mathit{sig}\,[p, D_{21}] &= \{a \mapsto -1\}\,, \\ \mathit{sig}\,[q, D_{21}] &= \{a \mapsto 1\}\,, & \mathit{sig}\,[r, D_{21}] &= \{a \mapsto 1\}\,. \end{aligned}$$

States i and p are already in classes which are singletons; states q and r have the same signature and there is no other state in D_{22}, thus there is no split.

The next processed class is D_{22}, and the signatures of predecessors of states in D_{22} are:

$$\begin{aligned} \mathit{sig}\,[i, D_{22}] &= \{\$ \mapsto 1\}\,, & \mathit{sig}\,[p, D_{22}] &= \{b \mapsto 1\}\,, \\ \mathit{sig}\,[q, D_{22}] &= \{a \mapsto 1, b \mapsto 1\}\,, & \mathit{sig}\,[r, D_{22}] &= \{a \mapsto 1, b \mapsto 1\}\,. \end{aligned}$$

States i and p are already in classes which are singletons; states q and r have the same signature and there is no other state in D_{22}, thus there is no split.

The queue is empty; the algorithm halts and the current partition is a congruence.

5. Complexity of the refinement algorithms

The high-level description of the DSA and PCSA have shown that every state is processed at most n times, where n is the number of true states. Processing a state consists in running over its outgoing transitions (for DSA) or its incoming transitions (for PCSA) to compute signatures. Globally, every transition of the automaton is thus considered at most n times.

To be efficient, the time to compute the signatures must be linear in the *number of transitions* involved in the computation. The key point in the algorithms is the ability to compute signatures in such a way that the signatures of two states can be compared in linear time. Usually, to compare two lists efficiently, a preliminary step consists in sorting them. To reach the linear complexity, a true sort is not affordable. Hence, we present here a *weak sort* [21] which allows to compare signatures in linear time.

5.1. The weak sort

Let f be an evaluation function from a set X to a set Y. In our framework, the evaluation function is the signature. If Y is totally ordered, sorting a list of elements of X with respect to f consists in computing a permutation of the list such that the elements of the list are ordered with respect to the evaluation f. If such a list is sorted, the elements with the same evaluation are contiguous and it is then easy to gather them. Following the ideas given in [21], it is not necessary to fully sort the list, and we say that a *list* of elements of X is *weakly sorted* (with respect to f) if the elements with the same evaluation by f are *contiguous*. It is then easy to compute the map equivalence of f.

Both DSA and PCSA are based on a weak sort with respect to signatures. Since signatures are themselves lists, nested weak sorts are necessary to gather states with the same signatures such that equal signatures are represented by the same lists.

In a first step, and for every state p, transitions outgoing from p with the same label and with destinations in the same class must be gathered.

Moreover, for two states with the same signature, the list of pairs (label $\mapsto$ weight) that form the signature must appear in the same ordering, in such a way the equality test is insured to be efficient.

In a second step, the weak sort is used to gather states with the same signature and to form the new classes.

A *bucket sort algorithm* [22] realises a weak sort with linear complexity in the size of X. Assume first that arrays indexed by Y can be managed. Let T be such an array where elements are lists of values in X. The algorithm iterates over X and stores every x in X in the list $T[f(x)]$. Finally, by iterating over T, all the lists are concatenated in order to compute a weakly sorted list.

This naive description hides two issues. First, Y may be large, and the initialisation of every element of T to the empty list is linear in Y, which can be much larger than X. The second issue is related to the first one, there may be a few y in Y which are images of some x by f, and it is too expensive to iterate over all elements of Y to concatenate the lists.

The *hash maps* are a solution to the first issue (see [23, 22] for instance). They allow to avoid the blowing up of memory in the case where Y is huge. As the amortised access time is constant, it allows to implement the weak sort in linear time. To solve the second issue, the keys of the hash map can be linked in order to efficiently iterate over the elements of $f(X)$ (the data structure is then equivalent to the *linked hash maps* implemented for instance in Java [24]).

5.2. Application to the Domain Split Algorithm

The computation of the global signature $\mathit{GSig}[p]$ of the states of the current class C requires two steps.

First, an array indexed by $A_\$ \times \mathcal{P}$ stores lists of pairs in $Q \times \mathbb{K}$. For every state p in C, and for every transition $p \xrightarrow{a|k} q$, the pair (p, k) is inserted in a list $\mathit{meet}[a, D]$, where D is the class of q. This insertion is special in the case where $\mathit{meet}[a, D]$ is not empty and its last element is a pair (p', k') with $p' = p$: k' is then updated to $k' + k$; if $k' + k$ is zero, the pair is removed from the list. The second step builds the global signature itself. For every useful index (a, D), for every (p, k) in $\mathit{meet}[a, D]$, inserts $(a, D) \mapsto k$ into $\mathit{GSig}[p]$.

We see that the elements in $\mathit{GSig}[p]$ follow an ordering which is consistent with the ordering of iteration on *meet*. Two states with the same global signatures have therefore the same list.

5.3. Application to the Predecessor Class Split Algorithm

For PCSA, the computation of the signature is slightly easier. The first array is only indexed by $A_\$$. For each q in D, for every transition $p \xrightarrow{a|k} q$, (p, k) is inserted in a list $\mathit{meet}[a]$. Then, for every key a of *meet*, for every (p, k) in $\mathit{meet}[a]$, $a \mapsto k$ is inserted in $\mathit{sig}\,[p, D]$ with the same special insertion used in DSA. Notice that this special insertion occurs here in the second step whereas it is used in the first step in DSA.

5.4. Splitting of classes

In DSA the current class is split, the signature of every state of the class is considered, and it is not difficult to split the class in linear time with respect to its size.

In PCSA the current class D is the splitter; it may induce the splitting of several classes, and all the operations must be performed in a time which is linear with respect to the number of transitions incoming to states of D. In particular, the splitting of a class must be performed in a time which is linear with the number of the states which are predecessors of a state of D — that may be much smaller than the size of the class.

To this end, it is required that the deletion of any element of a class can be performed in constant time. Thus classes are implemented by double linked lists, and an array indexed by states gives, for each state, a pointer to the location of the state in its class.

5.5. Analysis of both algorithms

In DSA, as seen in Section 4.3, every state may be considered n times, where n is the number of states. The computation of the global signature of a state requires a number of operations which is linear with the number of transitions outgoing from this state. Every iteration requires a time which is linear with the number of transitions outgoing from the current class. Finally, the time complexity of DSA is in $\mathrm{O}(n(m+n))$, where n is the number of states and m is the number of transitions of the automaton, provided that each operation and the computation of a hash for the weights is in constant time.

In PCSA, every state may be considered n times. The computation of the signatures during an iteration is linear with the number of transitions incoming to the current class. Finally, the time complexity of PCSA is in $\mathrm{O}(n(m+n))$, under the same conditions as above.

Theorem 5.1. The Domain Split Algorithm and the Predecessor Class Split Algorithm compute the minimal quotient of a $\mathbb{K}$-automaton with n states and m transitions in time $\mathrm{O}(n(m+n))$.

In the case of a deterministic Boolean automaton, where $m = \alpha n$, with $\alpha = |A|$, we get back to the classical $\mathrm{O}(\alpha n^2)$ complexity of the Moore minimisation algorithm (*cf.* [1, 3] for instance). It may seem strange that PCSA which has been said to be inspired by Hopcroft's algorithm has the same complexity. This is explained in the next section.

6. Conditions for an $\mathrm{O}(m \log n)$ algorithm

Hopcroft's algorithm can be seen as an improvement of PCSA for complete Boolean deterministic automata. Its time complexity is $\mathrm{O}(\alpha n \log n)$ (*cf.* for instance [25, 26]), where α is the size of the alphabet; this algorithm has been extended to incomplete DFA [19, 27] with complexity $\mathrm{O}(m \log n)$.

The strategy: "All but the largest", introduced in [28], can be applied to improve PCSA in some cases that we now study.

At every step of PCSA, some classes C are evaluated (through signatures) with respect to the current splitter D. If the class D is split into several classes, $D_1, \ldots, D_k$, all these classes are processed as splitters in further iterations.

The idea of the "All but the largest" strategy is that it is useless to process the last of the subclasses of D because after the splits induced by D itself and the splits induced by all the other subclasses, this subclass does not induce any new split. If this is true, one can choose which subclass is not processed; in order to get a better complexity, the strategy commands to choose the larger one.

6.1. Simplifiable signatures

A sufficient condition to apply this strategy is that the signatures with respect to the last subclass can be deduced from the signatures with respect to D and the signatures with respect to the other subclasses.

The signatures are equipped with the pointwise addition: for every a in $A_\$$,

$$\forall a \in A_\$ \qquad \big(\mathit{sig}\,[p, D] + \mathit{sig}\,[p, D']\big)(a) = \mathit{sig}\,[p, D](a) + \mathit{sig}\,[p, D'](a)\,,$$

and if D is a subset of Q and ψ a partition of D, then it holds:

$$\mathit{sig}\,[p, D] = \sum_{D' \text{ class of } \psi} \mathit{sig}\,[p, D']\,.$$

Definition 6.1. An automaton has *simplifiable signatures* if, for every subset D of Q and every subset C of D, and for every pair of states p, q, it holds

$$\mathit{sig}\,[p, D] = \mathit{sig}\,[q, D] \text{ and } \mathit{sig}\,[p, C] = \mathit{sig}\,[q, C] \Longrightarrow \mathit{sig}\,[p, D \setminus C] = \mathit{sig}\,[q, D \setminus C].$$

A commutative monoid $(M, \oplus)$ is *cancellative* if for every a, b, and c in M, $a \oplus b = a \oplus c$ implies $b = c$. In particular, every group is cancellative, and if $\mathbb{K}$ is a ring, the additive monoid $(\mathbb{K}, +)$ is cancellative.

Lemma 6.2. Let $\mathbb{K}$ be a semiring such that the additive monoid $(\mathbb{K}, +)$ is cancellative. Then every $\mathbb{K}$-automaton has simplifiable signatures.

For other weight semirings, the simplifiability of signatures depends on the automaton. If $\mathcal{A}$ is a deterministic[1] $\mathbb{K}$-automaton, that is, if for every state p and every letter a, there is at most one transition outgoing from p with label a, then the signatures are simplifiable, independently of $\mathbb{K}$, since it holds:

$$\forall p \in Q\,,\ \forall a \in A_\$ \quad \mathit{sig}\,[p, D \setminus C](a) = \begin{cases} \mathit{sig}\,[p, D](a) & \text{if } \mathit{sig}\,[p, C](a) = 0_\mathbb{K}\,, \\ 0_\mathbb{K} & \text{otherwise}\,. \end{cases}$$

Typically, incomplete deterministic Boolean automata, as considered in [19], have simplifiable signatures whereas general Boolean automata have not.

[1] called *sequential* in [10].

Example 6.3. Let $\mathcal{A}_2$ be the nondeterministic Boolean automaton of Figure 3. It holds:

$$\begin{aligned} \mathit{sig}\,[p,\{r,s\}](a) &= 1 & \mathit{sig}\,[q,\{r,s\}](a) &= 1 \\ \mathit{sig}\,[p,\{s\}](a) &= 1 & \mathit{sig}\,[q,\{s\}](a) &= 1 \\ \mathit{sig}\,[p,\{r\}](a) &= 1 & \mathit{sig}\,[q,\{r\}](a) &= 0. \end{aligned}$$

The signature with respect to $\{r\}$ cannot be deduced from the signature with respect to $\{r,s\}$ and $\{s\}$. Hence, $\mathcal{A}_2$ has not simplifiable signatures.

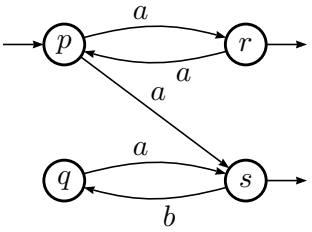

Figure 3. The nondeterministic automaton $\mathcal{A}_2$

6.2. The "All but the largest" strategy

If $\mathcal{A}$ is an automaton with simplifiable signatures, PCSA can be improved. We call this improved algorithm Fast Predecessor Class Split algorithm (FPCSA).

For every class C which is split with respect to a class D, if C is not already in the queue, all the classes in $\mathit{split}[C,D]$ except one of the largest are put in the queue. Notice that finding the largest class can be done in linear time with respect to the size of $\mathit{split}[C,D]$: the size of every subclass containing some predecessor of a state of D is computed at the same time as the signatures, the size of the class containing the other states is the difference of the size of C with the sum of the sizes of the other subclasses.

Actually, since C is not in the queue, the splitting of classes with respect to C has already been considered: for every D in the current partition, $\mathit{sig}\,[p,C]$ is the same for all p in D. Let C_1 be some subclass of C; if (D,C_1) is a splitting pair for some class D, then, since signatures are simplifiable, there exists some other class C_2 in ψ such that (D,C_2) is also a splitting pair.

Let $c(k)$ be the maximal number of times that a state p which belongs to a class D of size k that is removed from the queue will appear again in classes removed from the queue. A class D' containing p will be inserted only if D' results from a split of D and there exists another class D'' that results from the same split and whose size is at least as large as the size of D. Hence, the size of D' is at most $k/2$. Finally $c(k) \leqslant 1 + c(k/2)$, and, since a singleton class will not be split, $c(1) = 0$; therefore $c(k)$ is in $\mathrm{O}(\log k)$. Thus, the complexity of the algorithm is in $\mathrm{O}((m+n)\log n)$. This complexity meets the complexity of the Hopcroft algorithm [28] for the minimisation of complete deterministic automata which is in $\mathrm{O}(\alpha n \log n)$, where α is the size of the alphabet; in this case, $m = \alpha n$.

Theorem 6.4. If $\mathcal{A}$ is a $\mathbb{K}$-automaton with simplifiable signatures, FPCSA computes the minimal quotient of $\mathcal{A}$ in time $\mathrm{O}((m+n)\log n)$.

In the case of nondeterministic Boolean automata, as we have seen, the signatures are not simplifiable and the improvement from PCSA to FPCSA is therefore not warranted. Nevertheless, the Relation Coarsest Refinement algorithm described in [29] can be extended in order to compute the minimal quotient in time $\mathrm{O}(m \log n)$, as explained in [30]. For weighted nondeterministic automata over other semirings with no cancellative addition, like the $(\min, +)$-semiring, it is an open to know whether there exists an algorithm in time $\mathrm{O}(m \log n)$ for the computation of the minimal quotient.

7. Examples and benchmarks

DSA, PCSA and FPCSA are implemented in the AWALI library [8]. We present here a few benchmarks to compare their respective performances and to check that their execution time is consistent with their asserted complexity. Benchmarks have been run on an iMac Intel Core i5 3,4GHz, compiled with Clang 9.0.0.

First, we study a family of automata which is an adaptation of a family used in [31] to show that the Hopcroft algorithm requires $\Theta(n \log n)$ operations.

Let φ be the morphism defined on $\{a, b\}^*$ by $\varphi(a) = a\,b$ and $\varphi(b) = a$; for instance $\varphi(a\,b\,a\,a\,b) = a\,b\,a\,a\,b\,a\,b\,a$. The k-th Fibonacci word is $w_k = \varphi^k(a)$; its length is equal to the k-th Fibonacci number F_k, hence it is in $\Theta\Big(\big(\frac{1+\sqrt{5}}{2}\big)^k\Big)$. Notice that for every $k \geqslant 2$, $w_k = w_{k-1}.w_{k-2}$. Let $\mathcal{F}_k$ be the automaton with one initial state and a simple circuit around this initial state with label w_k (all states are final).

We observe on the benchmarks of Table 1 that the running time of the DSA is quadratic, while the running time of both PCSA and FPCSA algorithms are in $\Theta(k\,F_k)$ (*i.e.* $\Theta(F_k \log F_k)$, where F_k is the number of states).

	k	14	17	20	23	26	30
	F_k	987	4181	17711	75025	317811	2178309
DSA	t (s)	0.42	7.37	139		-	
	$10^{-7}t/F_k^2$	4.3	4.2	4.4			
PCSA	t (s)	0.010	0.045	0.257	1.36	73	257
	$10^{-7}t/k\,F_k$	7.2	6.3	7.3	7.6	6.7	7.5
FPCSA	t (s)	0.006	0.025	0.140	0.70	41	139
	$10^{-7}t/k\,F_k$	4.2	3.5	3.9	3.8	3.5	3.7

Table 1. Minimisation of $\mathcal{F}_k$

The second family is an example where PCSA and FPCSA have not the same complexity. Notice that these automata are acyclic and there may exist faster algorithms (*cf.* [3] for specific algorithms for acyclic Boolean deterministic automata), but this is out of the scope of this paper.

The n-th "Railroad" automaton has $2n$ states numbered from 1 to $2n$, and for every p in $[1; n-1]$, there are transitions from states $2p-1$ and $2p$ to states $2p+1$ and $2p$, as described by Figure 4. The

state 1 is initial and both $2n-1$ and $2n$ are final.

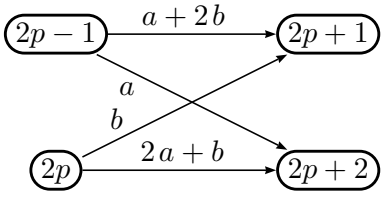

Figure 4. The transitions of the Railroad $\mathbb{Z}$-automaton

The benchmarks of the minimisation of "Railroad" automata are shown on Table 2. On these automata, if a temporary class contains all states between 1 and $2k$, it is split into one class $[1; 2k-2]$ and one class $\{2k-1, 2k\}$: the size of the classes lowers slowly. On these examples, DSA and PCSA are therefore quadratic.

In FPCSA, when this splitting occur, the largest class ($[1; 2k-2]$) is not put in the queue for further splittings; therefore, except at the first round, all splitters are pairs of states and the algorithm is linear.

	n	2^{10}	2^{12}	2^{13}	2^{14}	2^{15}	2^{22}
DSA	t (s)	3.29	53.2	214		-	
	$10^{-6}t/n^2$	3.1	3.2	3.2			
PCSA	t (s)	0.31	4.92	20.5	86.1	346	-
	$10^{-7}t/n^2$	3.0	2.9	3.1	3.2	3.2	
FPCSA	t (s)	0.008	0.030	0.061	0.12	0.24	30.8
	$10^{-6}t/n$	7.8	7.3	7.4	7.3	7.3	7.3

Table 2. Minimisation of Railroad(n)

References

[1] Hopcroft JE, Motwani R, Ullman JD. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 2006. 3rd edition.

[2] Pin JÉ (ed.). Handbook of Automata Theory. European Mathematical Society, 2021.

[3] Berstel J, Boasson L, Carton O, Fagnot I. Minimisation of Automata. In: Pin JÉ (ed.), Handbook of Automata Theory, Vol. I, pp. 337–373. European Mathematical Society, 2021.

[4] Droste M, Kuich W, Vogler H (eds.). Handbook of Weighted Automata. Springer, 2009.

[5] Garey MR, Johnson DS. Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences). W. H. Freeman, 1979.

[6] Baier C, Hermanns H. Weak Bisimulation for Fully Probabilistic Processes. In: Grumberg O (ed.), CAV 1997, volume 1254 of *Lect. Notes in Comput. Sci.* Springer, 1997 pp. 119–130.

[7] Boreale M. Weighted Bisimulation in Linear Algebraic Form. In: Bravetti M, Zavattaro G (eds.), CONCUR 2009, volume 5710 of *Lect. Notes in Comput. Sci.* Springer, 2009 pp. 163–177.

[8] AWALI: Another Weighted Automata LIbrary. `vaucanson-project.org/Awali`.

[9] Lombardy S, Sakarovitch J. Two Routes to Automata Minimization and the Ways to Reach It Efficiently. In: Câmpeanu C (ed.), CIAA 2018, volume 10977 of *Lect. Notes in Comput. Sci.* Springer, 2018 pp. 248–260.

[10] Sakarovitch J. Elements of Automata Theory. Cambridge University Press, 2009.

[11] Frougny C, Sakarovitch J. Synchronized relations of finite and infinite words. *Theoretical Computer Science*, 1993. **108**:45–82.

[12] Reutenauer C. Subsequential functions: characterizations, minimization, examples. In: IMYCS'90, number 464 in Lect. Notes in Comput. Sci. 1990 pp. 62–79.

[13] Béal MP, Lombardy S, Sakarovitch J. Conjugacy and Equivalence of Weighted Automata and Functional Transducers. In: Grigoriev D (ed.), CSR 2006, number 3967 in Lect. Notes in Comput. Sci. 2006 pp. 58–69.

[14] Lind D, Marcus B. An Introduction to Symbolic Dynamics and Coding. Cambridge University Press, 1995.

[15] Milner R. A Complete Axiomatisation for Observational Congruence of Finite-State Behaviors. *Inf. Comput.*, 1989. **81**(2):227–247.

[16] Benson DB, Ben-Shachar O. Bisimulation of Automata. *Inf. Comput.*, 1988. **79**(1):60–83.

[17] Larsen KG, Skou A. Bisimulation through Probabilistic Testing. *Inf. Comput.*, 1991. **94**(1):1–28.

[18] Lombardy S, Sakarovitch J. Derived terms without derivation. *J. Comput. Sci. and Cybernetics*, 2021. **37**(3):201–221. In arXiv: arxiv.org/abs/2110.09181.

[19] Béal MP, Crochemore M. Minimizing incomplete automata. In: Finite-State Methods and Natural Language Processing FSMNLP'08. 2008 pp. 9–16.

[20] Moore EF. Gedanken-Experiments on Sequential Machines. In: Shannon C, McCarthy J (eds.), Automata Studies, pp. 129–153. Princeton University Press, 1956.

[21] Paige R. Efficient Translation of External Input in a Dynamically Typed Language. In: Technology and Foundations - Information Processing IFIP'94, volume A-51 of *IFIP Transactions*. North-Holland, 1994 pp. 603–608.

[22] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms, 3rd Edition. MIT Press, 2009. ISBN 978-0-262-03384-8. URL `http://mitpress.mit.edu/books/introduction-algorithms`.

[23] Knuth DE. The art of computer programming, , Volume III, Sorting and Searching, 2nd Edition. Addison-Wesley, 1998. ISBN 0201896850. URL `https://www.worldcat.org/oclc/312994415`.

[24] Java™ Platform. Standard Edition 17 API Specification. Class `LinkedHashMap<K,V>`, 2021. Available at: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/LinkedHashMap.html.

[25] Gries D. Describing an algorithm by Hopcroft. *Acta Informatica*, 1973. **2**:97–109.

[26] Berstel J, Carton O. On the Complexity of Hopcroft's State Minimization Algorithm. In: Implementation and Application of Automata CIAA'04, volume 3317 of *Lect. Notes in Comput. Sci.* Springer, 2004 pp. 35–44.

[27] Valmari A, Lehtinen P. Efficient Minimization of DFAs with Partial Transition. In: STACS'08, volume 1 of *LIPIcs*. Schloss Dagstuhl, 2008 pp. 645–656.

[28] Hopcroft J. An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. In: Kohavi Z, Paz A (eds.), Theory of Machines and Computations, pp. 189–196. Academic Press, New York, 1971.

[29] Paige R, Tarjan RE. Three Partition Refinement Algorithms. *SIAM J. Comput.*, 1987. **16**(6):973–989.

[30] Fernandez J. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Sci. Comput. Program.*, 1989. **13**(1):219–236. doi:10.1016/0167-6423(90)90071-K. URL `https://doi.org/10.1016/0167-6423(90)90071-K`.

[31] Castiglione G, Restivo A, Sciortino M. On extremal cases of Hopcroft's algorithm. *Theor. Comput. Sci.*, 2010. **411**(38-39):3414–3422.