Theoretical Computer Science 1004 (2024) 114621

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Incremental NFA minimization ☆

Christian Bianchini [a], Alberto Policriti [a], Brian Riccardi [a,b,*], Riccardo Romanello [a]

[a] Department of Mathematics, Computer Science, and Physics - University of Udine, Via Le Scienze, 206, Udine, 33100, Italy
[b] Department of Informatics, Systems and Communication - University of Milano-Bicocca, Viale Sarca, 336, Milan, 20126, Italy

ARTICLE INFO

Keywords:
Automata
Bisimulation
Minimization
Incremental

ABSTRACT

We tackle the (classic) problem of minimizing (non)deterministic finite automata. The algorithm we put forward has the peculiarity of being incremental, *i.e.*, the minimization proceeds by successive iterations, each producing a partially minimized automaton language-equivalent to the input one.

Our algorithm builds upon Almeida et al. from 2014, fixing a minor mistake and generalizing it to the nondeterministic case. It relies on a colouring procedure of a graph associated to the automaton, keeping track of partial information. After dealing with the deterministic case, we extend this idea to the bisimulation-minimization of nondeterministic automata.

The algorithms for both the deterministic and the nondeterministic cases run in time $\mathcal{O}(nm)$ for an automaton with n states and m transitions. The complexity for the deterministic case matches the complexity claimed by Almeida et al.. The nondeterministic case improves the fastest known incremental algorithm for this problem.

We conclude introducing and using a notion of *signature* of a state, whose aim is to exploit pre-computed information potentially available, to speed-up the process. A signature is used to produce an initial partition of the automaton's states and can be easily integrated in both the incremental and the non-incremental algorithm.

1. Introduction

Finite state automata are fundamental objects in Theoretical Computer Science and find their application in Text Processing, Compilers Design, Artificial Intelligence and many other areas. The minimization of an automaton is the process of constructing a new (language-equivalent) automaton which is minimal in the number of states. This problem can be traced back to the '50s by the work of Moore [1]. A fundamental result in Automata Theory is the Myhill-Nerode Theorem [2], establishing that, in the deterministic case, this minimal automaton is in fact *the minimum* (up to isomorphism). In the wider setting of nondeterministic automata there is no analog result and finding any state-minimal automaton is PSPACE-complete [3]. For this reason, a practical alternative is the minimization with respect to bisimulation. *Bisimilarity* is indeed a valid choice since in the deterministic case two states are bisimilar if and only if they are Myhill-Nerode equivalent.

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.

* Corresponding author.
E-mail addresses: bianchini.christian@spes.uniud.it (C. Bianchini), alberto.policriti@uniud.it (A. Policriti), brian.riccardi@unimib.it, brian.riccardi@uniud.it (B. Riccardi), riccardo.romanello@uniud.it (R. Romanello).

https://doi.org/10.1016/j.tcs.2024.114621
Received 15 January 2023; Received in revised form 23 February 2024; Accepted 3 May 2024

Available online 9 May 2024

Thus, the problem of minimizing automata reduces to the problem of computing bisimilarity between states, which in turn is equivalent to determining the coarsest partition of a set stable with respect to some binary relation [4]. The two main paradigms to compute the aforementioned partition are *top down* and *bottom up*.

Top down algorithms (also known as *partition refinement*) start with the partition that separates states between final and non final and subsequently *refine* the partition until it is stable. By a careful choice of which block to split at each refining step, Paige and Tarjan [4] devised an algorithm that computes the maximum bisimulation equivalence in time $\mathcal{O}(m \log n)$, where n is the number of states and m is the number of transitions of the automaton. The iconic Hopcroft's Algorithm [5] (which Paige and Tarjan's solution is based on) deals with the special case of deterministic automata. Furthermore, it has recently been proved in [6] that, when restricting the attention to top-down algorithms, bisimilarity computation requires $\Omega(m \log n)$ worst-case time. In this sense, the algorithm proposed in [4] is *optimal*.

In contrast to the previous approach, bottom up algorithms start with the finest partition—the one where each state constitutes a singleton—and proceed by subsequently *merging* two blocks found to be equivalent. For this reason, the technique is also known in the literature as *partition-aggregation*. The main advantage of this paradigm is that the algorithm is *incremental*, that is, it proceeds in subsequent stages where at the end of each merging step the resulting automaton is language-equivalent to the input one. In this way, the minimization process can be stopped at any time and can be resumed later.

The first algorithm of this kind is due to Watson [7]. After a series of improvements Watson and Daciuk [8] reduced the running time to $\mathcal{O}(n^2|\Sigma|\alpha(n))$ for deterministic automata with n states, alphabet Σ and where $\alpha(n)$ is related to the *inverse* of Ackermann's function [9], which can be treated as a constant for any reasonable value of n[1]. The main idea is to propagate the definition of bisimilarity: if states p and q are equivalent, then also their (unique) transitions by the same character must lead to equivalent states. This is done by a recursive function EQUIV which resembles an equivalence algorithm by Hopcroft and Karp [10]. A subsequent work by Almeida et al. [11] aimed at simplifying the algorithm by Watson and Daciuk maintaining its running time. Unfortunately, there is a small mistake in their version of EQUIV which leads to a $\Omega(n^3)$ algorithm in the worst case.

Above algorithms are focused on the minimization of deterministic automata. The nondeterministic case was tackled by Björklund and Cleophas [12] adapting ideas from Watson and Daciuk. They devised an incremental algorithm for computing bisimilarity in time $\mathcal{O}(n^2r^2|\Sigma|)$, where r is the degree of nondeterminism of the automaton, that is, the maximum number of equally-labelled transitions exiting from any state. Their solution associates a propositional formula to the automaton (capturing bisimilarity between pairs of states) and subsequently builds (incrementally) a maximal model that satisfies it. As a side-effect, since in the deterministic case $r = 1$, the algorithm by Björklund and Cleophas solves the minimization of deterministic automata by partition aggregation in $\mathcal{O}(n^2|\Sigma|)$ time (as promised in [11]).

In this work, which is an extension of [13], we present our correction to the algorithm by Almeida et al. providing a simplified version of the one by Watson and Daciuk while maintaining the quadratic running time. The solution is based on the concept of *associated graph* whose purpose is twofold: 1) to distill the behaviour of the aforementioned algorithms by interpreting them as graph colourings, and 2) to design our own incremental procedure. Having established the connection between minimization and graph colouring, it is natural to generalize the algorithm for bisimilarity-minimization on nondeterministic automata. Furthermore, the proposed solution improves by a factor of r the running time of Björklund and Cleophas [14].

We conclude introducing and using a notion of *signature* of a state, whose aim is to exploit pre-computed information potentially hidden in the automaton, to speed-up the process. A signature is used to induce an initial partition of the automaton's states and can be easily integrated in both the incremental and the non-incremental algorithm as a preprocessing step.

The paper is organized as follows: in the next section we give some basic notions about partitions, relations, and automata. In Section 3 we briefly describe the algorithm by Almeida et al. and we point out the mistake. Our procedure for the deterministic case is introduced in Section 4. Continuing, in Section 5 we lift the algorithm to the nondeterministic case. Finally, the notion of signature of a state is presented and used in Section 6.

2. Preliminaries

2.1. Relations and partitions

A *binary relation* over a set A is a subset $\rho \subseteq A \times A$. Its size will be denoted by $|\rho|$. We say that $a, b \in A$ are in relation when $(a, b) \in \rho$ and we denote this by writing $a\rho b$. The *identity relation* is $\iota_A = \{(a, a) : a \in A\}$. The relation ρ over A can be interpreted as a directed graph $G_{A,\rho} = (A, \rho)$.

An *equivalence relation* (or equivalence) is a relation which is reflexive, symmetric, and transitive. Given $a \in A$, the *equivalence class* of a is the set $[a] = \{b : a\rho b\}$. The *quotient set* $A/\rho = \{[a] : a \in A\}$ forms a *partition* of A. Notice that if ρ is an equivalence, the connected components of $G_{A,\rho}$ are the equivalence classes.

Given a binary relation ρ over A, we define its *equivalence closure* ρ^e as the smallest equivalence relation (*w.r.t.* set inclusion) that contains ρ.

[1] It holds $\alpha(n) \leq 5$ for $n \leq 2^{2^{16}}$.

2.2. Languages and automata

An *alphabet* is a finite and non-empty set Σ of *symbols*. A *string* is a finite sequence $w = w_1 \dots w_n$ of symbols. Σ^* is the set of all finite length strings of symbols in Σ, and we call a subset $L \subseteq \Sigma^*$ a *language*. The *empty* string is denoted by ϵ and has length $|\epsilon| = 0$.

A *nondeterministic finite state automaton* (NFA) is a tuple $\mathcal{N} = \langle Q, \Sigma, I, \delta, F \rangle$ where Q is a non-empty finite set of *states*, Σ is the alphabet, $I \subseteq Q$ is the set of *initial states*, $\delta : Q \times \Sigma \to 2^Q$ is the *transition function* and $F \subseteq Q$ is the set of *final* states. The *degree of nondeterminism* is $r = \max\{|\delta(q,x)| \ : \ q \in Q, x \in \Sigma\}$. We say that $\mathcal{N}$ is *complete* if $|\delta(q,x)| \geq 1$ for every state and symbol. In what follows we will assume complete automata: this is not a loss of generality since it is always possible to complete an automaton by adding *one* state and at most $|\Sigma|(n+1)$ transitions. Furthermore, pseudocodes from sections 4 and 5 greatly benefit from this completeness assumption in terms of simplicity, but it is not difficult to adapt our ideas for non-complete automata. As usual, the transition function can be recursively extended to strings, *i.e.* $\delta^* : Q \times \Sigma^* \to 2^Q$, still denoted by δ.

We say that state $q \in Q$ *accepts* a string $w \in \Sigma^*$ if $\delta(q,w) \cap F \neq \emptyset$. The set of strings accepted by q is denoted by $L(q)$. The *language accepted* by automaton $\mathcal{N}$ is $L(\mathcal{N}) = \bigcup_{q \in I} L(q)$. A *minimal* automaton accepting L has the minimum number of states amongst all automata accepting L.

A *deterministic finite state automaton* (DFA) is a NFA $\mathcal{D}$ with the added conditions that I is a singleton (whose unique element is denoted by q_0), and for each state q and each symbol x it holds $|\delta(q,x)| = 1$. In this case we will treat δ as a function $Q \times \Sigma \to Q$.

Since we are interested in the language accepted by a given automaton, we would like to get rid of *redundant* states, that is, we want to equate states accepting the same language. Formally:

Definition 1. Let $\mathcal{N} = \langle Q, \Sigma, \delta, I, F \rangle$ be a NFA. We define relation $\sim_{\mathcal{N}} \subseteq Q \times Q$ as:

$$p \sim_{\mathcal{N}} q \overset{\text{def}}{\iff} L(p) = L(q).$$

We say that p and q are *equivalent* (resp. *distinguishable*) whenever $p \sim_{\mathcal{N}} q$ (resp. $p \nsim_{\mathcal{N}} q$). In the special case of $p \in F$ and $q \notin F$ (or viceversa) we say that pair (p,q) is *trivially distinguishable*. When the automaton is known by the context, we will write the relation symbol $\sim$ without subscript.

Observation 1. *In case of a DFA $\mathcal{D}$, we may unroll the definition of $\sim$ in the following recursive way (which will be useful later):*

$$\begin{aligned} p \sim q &\iff (\forall w \in \Sigma^*)(\delta(p,w) \in F \leftrightarrow \delta(q,w) \in F) \\ &\iff (p \in F \leftrightarrow q \in F) \wedge (\forall x \in \Sigma)(\delta(p,x) \sim \delta(q,x)). \end{aligned}$$

Given a NFA $\mathcal{N} = \langle Q, \Sigma, I, \delta, F \rangle$, and an equivalence ρ over Q, the *quotient* of $\mathcal{N}$ by ρ is defined as $\mathcal{N}/\rho = \langle Q/\rho, \Sigma, I_\rho, \delta_\rho, F/\rho \rangle$, where $I_\rho = \{[q] \mid q \in I\}$, and $\delta_\rho([q],x) = \{[q'] \mid q' \in \delta(q,x)\}$. The *Myhill-Nerode Theorem* [2] establishes that, for any DFA $\mathcal{D}$, the quotient automaton $\mathcal{D}/\sim$ is well defined and is the unique (up to isomorphism) minimal automaton recognizing $L(\mathcal{D})$. In the case of NFAs there is no analog result and finding any such minimal automaton is PSPACE-complete [3].

2.3. Bisimulation and bisimilarity

Definition 2. Let $\mathcal{N} = \langle Q, \Sigma, \delta, I, F \rangle$ be a NFA. A *bisimulation* is a binary relation $B \subseteq Q \times Q$ such that, for every pair $(p,q) \in B$:

B1. $p \in F \leftrightarrow q \in F$,
B2. $(\forall x \in \Sigma)(\forall p' \in \delta(p,x))(\exists q' \in \delta(q,x))(\langle p', q' \rangle \in B)$,
B3. $(\forall x \in \Sigma)(\forall q' \in \delta(q,x))(\exists p' \in \delta(p,x))(\langle p', q' \rangle \in B)$.

Two states p and q are said *bisimilar* if there exists a bisimulation which contains (p,q). The set of all bisimulations over the states of $\mathcal{N}$ is denoted by $\mathfrak{B}_{\mathcal{N}}$.

If B_1 and B_2 are two bisimulations over some automaton $\mathcal{N}$, it is not difficult to prove that also $B_1 \cup B_2$ is a bisimulation over $\mathcal{N}$—essentially, the union does not invalidate the three conditions of Definition 2 which were previously verified by pairs of B_1 and pairs of B_2. Furthermore, notice that in case $\mathcal{N}$ is a DFA the relation of bisimilarity of Definition 2 is the same as the recursive reformulation of $\sim$ given in Observation 1. The aforementioned properties, whose proof can be found in the literature (cf. [15, Chapter 1], and [16, Proposition 3]), are summarized in Lemma 1.

Lemma 1. *Let $\mathcal{N}$ be a NFA. Then, the following hold:*

1. *$\mathfrak{B}_{\mathcal{N}}$ is closed under union, and admits a* unique *largest (with respect to set-union) bisimulation $\mathcal{B}_{\mathcal{N}}$,*
2. *$\mathcal{B}_{\mathcal{N}}$ is an equivalence relation that relates all and only bisimilar states, and*
3. *$\mathcal{B}_{\mathcal{N}} \subseteq \sim_{\mathcal{N}}$, and if $\mathcal{N}$ is deterministic, then $\mathcal{B}_{\mathcal{N}} = \sim_{\mathcal{N}}$.*

In particular, point (3) of Lemma 1 justifies the use of $\mathcal{B}_{\mathcal{N}}$ as an approximation of $\sim_{\mathcal{N}}$ for nondeterministic automata. As usual, we will omit the subscript when the automaton is known by the context.

2.4. Partition aggregation

Given an automaton $\mathcal{N}$, our goal is to compute the bisimilarity relation $\mathcal{B}$ over its set of states, so that the resulting quotient $\mathcal{N}/\mathcal{B}$ can be returned as the *minimized* version of $\mathcal{N}$. The *partition-aggregation* strategy will compute an ascending chain $\imath \subseteq B_1 \subseteq \ldots \subseteq B_n = \mathcal{B}$ of bisimulation-equivalences.

A partition-aggregation algorithm proceeds by a sequence of merging steps where at each step i the bisimulation B_i is computed. Since each B_i is a bisimulation, the minimization process can be stopped at any step obtaining a language-equivalent automaton with no more states than the input one [7,8,11,14]; the minimization process can be resumed later from this intermediate automaton. In this sense, the algorithm is *incremental*. This property is not shared with top down algorithms that proceed by *partition-refinement*—such as Hopcroft's Algorithm and its many successors—where only the final result is a bisimulation.

3. The algorithm proposed by Almeida et al.

This section is devoted to a brief description of the algorithm proposed by Almeida et al. It uses the *union-find* [17,9] data structure to manage the partition of states, so that finding and merging classes with the FIND and UNION primitives can be done in $\mathcal{O}(\alpha(n))$.

Algorithm 1 Aggregation-based minimization by Almeida et al.

```
 1: function MinimizeAlmeida(Q, Σ, δ, F)
 2:     for all q ∈ Q do
 3:         Make(q)
 4:     E ← (F × F^c) ∪ (F^c × F)
 5:
 6:     for all ⟨p, q⟩ ∈ Q × Q do
 7:         fp ← Find(p)
 8:         fq ← Find(q)
 9:         if fp ≠ fq ∧ ⟨p, q⟩ ∉ E then
10:             E ← ∅
11:             H ← ∅
12:             if Equiv(p, q) then
13:                 for all ⟨p', q'⟩ ∈ E do
14:                     Union(p', q')
15:             else
16:                 E ← E ∪ H
17:
18:     return {[p] | p ∈ Q}
19: end function

20: function Equiv(p, q)
21:     if ⟨p, q⟩ ∈ E then
22:         return ⊥
23:     if ⟨p, q⟩ ∈ H then
24:         return ⊤
25:
26:     H ← H ∪ {⟨p, q⟩, ⟨q, p⟩}
27:     for all x ∈ Σ do
28:         ⟨p', q'⟩ ← ⟨Find(δ(p, x)), Find(δ(q, x))⟩
29:         if p' ≠ q' ∧ ⟨p', q'⟩ ∉ E then
30:             E ← E ∪ {⟨p', q'⟩, ⟨q', p'⟩}
31:             if ¬Equiv(p', q') then
32:                 return ⊥
33:
34:     H ← H \ {⟨p, q⟩, ⟨q, p⟩}
35:     E ← E ∪ {⟨p, q⟩, ⟨q, p⟩}
36:     return ⊤
37: end function
```

Pairs of states are recursively considered until their equivalence is estabilished. Intermediate results are cached, so that queries on pairs of states already found to be (non-)equivalent can be answered in constant time.

At lines 2–3, the identity relation is constructed and pairs of trivially distinguishable states are added to the *memoization* table $\overline{E}$. In the main loop at lines 6–16, we iterate over all pairs of states to check for equivalence. If a pair is either on the same class—*i.e.* is a pair of states already found to be equivalent—or is in the memoization table—*i.e.* is a pair of distinguishable states—the minimization continues to the next iteration. Otherwise, two empty collections E and H are prepared, respectively the set of *potentially equivalent* pairs of states and the *history* pairs. E and H are considered global variables and can be accessed from EQUIV. The recursive function EQUIV is responsible for checking if states p, q are equivalent and, if so, pairs in E are merged. Otherwise, all visited pairs are set to be distinguishable and this information is used to update $\overline{E}$. At the end the partition in equivalence classes is returned.

The underlying idea of EQUIV(p, q) is to recursively check the transitions from p and q on all symbols (see Observation 1). If two states are found to be cached as distinguishable, the recursion stops returning $\bot$. If they are found to be in visit, it is useless to continue the visit and nothing can be said (*i.e.* EQUIV returns $\top$ postponing the decision to the upper-level of the recursion). These preliminary checks are at lines 21–24. Next, each x-transition is checked recursively in the loop at 27–32, stopping when the states are found to be distinguishable. At the end, if p and q are not found to be distinguishable, the pair is removed from the history H, added to E, and $\top$ is returned.

Detailed proof of the algorithm's correctness can be found in [11].

On the complexity analysis, the authors claim that the algorithm terminates in time $\mathcal{O}(n^2|\Sigma|\alpha(n))$. This comes from the assumption that each pair visited during the recursion of EQUIV will be skipped on the subsequent iterations of MINIMIZEALMEIDA (*cf.* [11, Lemma 4.9]). This assumption is wrong and a family of counterexamples can be constructed such that MINIMIZEALMEIDA terminates in time $\Omega(n^3|\Sigma|\alpha(n))$.

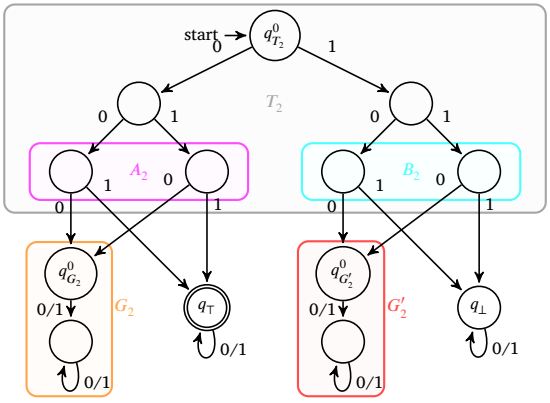

Fig. 1. An example of automaton $\mathcal{E}_2$. In this case, G_2 and G'_2 are 2-states linear components.

3.1. The counterexample

In this section we briefly describe an infinite family of automata to *fool* Algorithm 1 into running in cubic time. More precisely, we give a schema to build, for every integer $n \geq 1$, an automaton $\mathcal{E}_n$ of size $6n+1$ for which there exist n^2 pairs of states $\langle p, q\rangle$ such that EQUIV(p, q) runs in $\Omega(n|\Sigma|)$ time.

Automaton $\mathcal{E}_n$ results from the composition of five building blocks, which are DFAs over the alphabet $\Sigma = \{0, 1\}$:

- G_n, any automaton with n states whose initial state is $q^0_{G_n}$,
- G'_n, a copy of G_n whose initial state is $q^0_{G'_n}$,
- $W_\top$, whose unique (and final) state is $q_\top$,
- $W_\bot$, whose unique (and non-final) state $q_\bot$,
- T_n, shaped as a *complete* binary tree of $4n-1$ states. The $2n$ *leaves* are partitioned into two sets of n states each, respectively A_n and B_n, which are either all final or all non-final.

Now, let $\mathcal{E}_n = \langle Q, \Sigma, \delta, q^0_{T_n}, F\rangle$ be the DFA obtained combining all automata described above, where:

- Q is the union of all the sets of states,
- F is the union of all the sets of final states,
- The initial state of T_n becomes the initial state of $\mathcal{E}_n$,
- δ is the union of all transition functions with the following adjustments:
  - $\delta(q_A, 0) = q^0_{G_n}$ and $\delta(q_A, 1) = q_\top$, for each $q_A \in A_n$,
  - $\delta(q_B, 0) = q^0_{G'_n}$ and $\delta(q_B, 1) = q_\bot$, for each $q_B \in B_n$.

Then, automaton $\mathcal{E}_n$ is called a *counterexample of size n* (in Fig. 1 is represented the case for $n = 2$).

To show how such automaton fools Algorithm 1, consider the execution of EQUIV(q_A, q_B) for $q_A \in A_n$ and $q_B \in B_n$. Since the states are not trivially distinguishable, the procedure will first take the 0-transition leading to a complete visit of both G_n (from q_A) and G'_n (from q_B), where all visited pair of states will be found equivalent. After that, the procedure will traverse the 1-transition reaching pair $\langle q_\top, q_\bot\rangle$, which is trivially distinguishable, leading to the conclusion that also $\langle q_A, q_B\rangle$ is distinguishable. The information about the equivalence between G_n and G'_n (stored in E by the algorithm) is lost and both sub-automata will be visited again when starting a visit from another pair.

Thus, the execution of EQUIV(q_A, q_B) runs in $\Theta(n|\Sigma|)$ steps and this will happen for each of the n^2 pairs in $A \times B$, for a total of $\Theta(n^3|\Sigma|)$ steps.

4. Deterministic case

In this section we present our idea to correct the previously presented algorithm, discussing its correctness and complexity. The main point is to run the recursive check on a richer data structure, the *associated graph* introduced below, whereby the running time of the overall algorithm is going to be $\mathcal{O}(n^2|\Sigma|)$.

As already mentioned in the Preliminaries, the completeness assumption for automata will ease the pseudocode and the explanation of Algorithm 2. Nevertheless, our ideas can be easily adapted for the general case of non-complete automata while maintaining the promised complexity.

4.1. The associated graph

The reason why Algorithm 1 is not quadratic on some automata is the fact that whenever a pair of distinguishable states is found the recursion stops and loses reusable information gathered on elements of E. A graph associated to the automaton clarifies how pairs of states evolve when they are found to either be equivalent or distinguishable.

Definition 3. Let $D = \langle Q, \Sigma, \delta, q_0, F\rangle$ be a DFA. We define its *associated graph* $\mathcal{G} = (V, A)$ as follows:

$$V = Q \times Q,$$

$$A = \{\langle p, q\rangle \to \langle \delta(p, x), \delta(q, x)\rangle \mid p, q \in Q, x \in \Sigma\}.$$

Vertex $\langle p, q\rangle$ is said to be *equivalent/distinguishable* whenever p and q are. Furthermore, it is said to be *trivially distinguishable* in case exactly one of the two is final.

By colouring $\mathcal{G}$ with distinguishable vertices in black and equivalent vertices in white, the problem of computing $\sim$ can be seen as the problem of correctly colouring the associated graph.

The algorithm by Almeida et al. can be described as follows. It starts by colouring *trivially* distinguishable and equivalent vertices in black and white, respectively, and in grey the remaining vertices. At each iteration of the main loop, it considers a grey vertex $\langle p, q\rangle$ and starts a visit of $\mathcal{G}$ from it. If the visit reaches a black vertex $\langle p', q'\rangle$, the recursion stops and all vertices in the path from $\langle p, q\rangle$ to $\langle p', q'\rangle$ (saved in H) are coloured in black. Otherwise, if all paths lead either to white or grey vertices all visited vertices are coloured white. The main issue with Algorithm 1 is that when a black vertex is encountered all information of vertices in $E \setminus H$ gets lost.

4.2. The algorithm for deterministic automata

We present the minimization algorithm based on the observations above.

Algorithm 2 Proposed algorithm for deterministic automata.

```
 1: function MinimizeDfa(Q, Σ, δ, F)
 2:     for all ⟨p,q⟩ ∈ Q × Q do
 3:         if p, q are triv. distinguishable then
 4:             Color(⟨p,q⟩) ← Black
 5:         else if p = q then
 6:             Color(⟨p,q⟩) ← White
 7:         else
 8:             Color(⟨p,q⟩) ← Grey
 9:     for all ⟨p,q⟩ ∈ Q × Q do
10:         if Color(⟨p,q⟩) = Grey then
11:             H ← EmptyGraph
12:             eq ← Equiv(⟨p,q⟩)
13:             if ¬eq then
14:                 H ← Reverse(H)
15:                 Visit(H, h)
16:     W ← {⟨p,q⟩ | Color(⟨p,q⟩) = White}
17:     return Q/W^e
18: end function

19: function Equiv(⟨p,q⟩)                          ▷ H and h global
20:     if ⟨p,q⟩ ∈ H then
21:         return ⊤
22:     else if Color(⟨p,q⟩) = Black then
23:         h ← ⟨p,q⟩
24:         return ⊥
25:     else if Color(⟨p,q⟩) = White then
26:         return ⊤
27:     else                                       ▷ here ⟨p,q⟩ is Grey and fresh
28:         Color(⟨p,q⟩) ← White
29:
30:         for all x ∈ Σ in lex. order do
31:             ⟨p_x, q_x⟩ ← ⟨δ(p,x), δ(q,x)⟩
32:             AddArc(H, ⟨p,q⟩, ⟨p_x, q_x⟩)
33:             eq ← Equiv(⟨p_x, q_x⟩)
34:             if ¬eq then return ⊥
35:
36:         return ⊤
37: end function
```

We assume that primitive Color has access to a data-structure to get/set the colour of a vertex in constant time (*e.g.*, an array or a succint bitvector). Furthermore, we assume that such primitive gets/sets symmetrically a pair, *i.e.*, when $\textsc{Color}(p, q)$ is used both $\langle p, q\rangle$ and $\langle q, p\rangle$ are considered.

The general structure of MinimizeDfa is the same as MinimizeAlmeida rewritten in terms of colourings. The only difference is that instead of maintaining two sets E and H we maintain the global variable $\mathcal{H}$ which represents the visited portion of $\mathcal{G}$. The idea is that when $\textsc{Equiv}(\langle p, q\rangle)$ returns to the main loop, after line 15, vertices in $\mathcal{H}$ will be correctly coloured, either in White or Black. In Algorithm 1 this information was lost while in Algorithm 2 $\mathcal{H}$ is used to determine extra black vertices. Helper procedures Reverse and Visit perform, respectively, arc-reverse of a graph and the Black-colouring of $\mathcal{H}$ starting from the source vertex h. After the loop, we gather all White pairs in $\mathcal{W}$ and we return the quotient $Q/\mathcal{W}^e$—remember that the equivalence classes can be computed as the connected components of $G_{Q,\mathcal{W}}$ (cf. Section 2.1).

Let us analyze the version of $\textsc{Equiv}(\langle p, q\rangle)$ in Algorithm 2. At lines 20–27 some base cases are checked. In particular, if $\langle p, q\rangle$ is Black, then it is stored in the global variable h and $\bot$ is returned. Otherwise, if $\langle p, q\rangle$ is White we return $\top$ to continue the downstream inspection. Finally, in case $\langle p, q\rangle$ is Grey, it is coloured White and at lines 30–34 the for loop tries to continue the recursive visit by reading each symbol *in lexicographic* order. Before each recursive call $\mathcal{H}$ is updated by adding arc $\langle p, q\rangle \to \langle p_x, q_x\rangle$—we assume vertex $\langle p_x, q_x\rangle$ is added, if not already present.

Below we outline the main arguments for complexity and correctness.

4.3. Complexity analysis

First, notice that $|\mathcal{G}| = |V| + |A| = n^2 + nm$. It is clear that, summing over all the iterations of the main loop, line 9, the associated graph is visited at most thrice: during the "forward" recursion pass and, optionally, during REVERSE and VISIT. In fact, every vertex starts GREY and becomes WHITE during the forward pass of an EQUIV-call. Possibly, if the call returns $\perp$, some WHITE vertices become BLACK.

Finally, the connected components of graph $G_{Q,\mathcal{W}}$ can be computed in time $\mathcal{O}(n^2)$.

In total, we have the following:

Theorem 1. *Given a complete DFA with n states and $m = n|\Sigma|$ transitions,* MINIMIZEDFA *terminates in time $\mathcal{O}(nm)$.*

4.4. Correctness

To prove correctness we will show the following invariant at line 15: all vertices in $\mathcal{H}$ are correctly coloured either BLACK or WHITE—Lemma 2 below.

We start by showing some properties of the colouring performed by EQUIV and VISIT. First of all, notice that, since we are in the deterministic case, for every $u = \langle p, q\rangle \in V$ and $w \in \Sigma^*$ there is a unique path in $\mathcal{G}$ starting from u and spelling w: denote by $\delta(u, w) = \langle \delta(p, w), \delta(q, w)\rangle$ the last vertex of this path. In the following, we recall that a path in a graph is said to be *simple* when it has no repeating vertices.

Definition 4. Let D be a DFA, and $\mathcal{G} = (V, A)$ its associated graph. Let $u \in V$, and $w \in \Sigma^*$. We say that w for u is:

1. *simple* if the path $u \rightsquigarrow \delta(u, w)$ in $\mathcal{G}$ is simple, and
2. *avalanche* if it is simple and vertex $\delta(u, w)$ is BLACK.

If there exists w avalanche for u, denote by $\mathrm{av}(u)$ the lexicographically smallest such w and name $u \rightsquigarrow \delta(u, \mathrm{av}(u))$ the *avalanche path of u*.

If EQUIV(u_0) is called in the main loop, line 12, the visit checks all *simple* words for u_0 in lexicographic order, either until all words are explored or—if it exists—until $\mathrm{av}(u_0)$ is found. Any vertex that can reach the avalanche path should be coloured in BLACK.

Proposition 1. *Consider $\mathcal{H}$ upon return of* EQUIV(u_0) *at line 12. If there exists $u \in \mathcal{H}$ distinguishable, then:*

1. $\mathrm{av}(u_0)$ *exists, and*
2. *If all* WHITE *vertices in $\mathcal{G} \setminus \mathcal{H}$ are equivalent, then:*

$$(\forall u \in \mathcal{H})\left(u \text{ distinguishable} \implies u \overset{\mathcal{H}}{\rightsquigarrow} \delta(u_0, \mathrm{av}(u_0))\right).$$

Proof. Suppose $u \in \mathcal{H}$ is distinguishable:

1. First of all, we prove that there exists $w \in \Sigma^*$ such that $\delta(u_0, w)$ is BLACK. Since $u \in \mathcal{H}$, there exists w_0 such that $u = \delta(u_0, w_0)$. Since u is distinguishable, there exists w_1 such that $\delta(u, w_1)$ is *trivially* distinguishable (*i.e.* a final/non-final pair, which is BLACK from the start). Thus, $w = w_0 w_1$ and $\delta(u_0, w_0 w_1)$ is BLACK. Since $\mathrm{av}(u_0)$ is the lexicographically smallest such w, we proved point (1).
2. We want to prove that, under suitable conditions, every distinguishable node $u \in \mathcal{H}$ can reach the avalanche path of u_0 remaining inside $\mathcal{H}$.
   Let $h = \delta(u_0, \mathrm{av}(u_0))$, $u \in \mathcal{H}$ distinguishable, and π be the $\mathcal{G}$-path leading u to some *trivially* distinguishable v. We claim π must cross the avalanche path of u_0 (call it α). Suppose not, for the sake of contradiction. Since h is the unique BLACK vertex in $\mathcal{H}$, $v \notin \mathcal{H}$. Hence, π must traverse some arc $u' \to u''$ with $u' \in \mathcal{H}$ and $u'' \notin \mathcal{H}$. By assumption on π and construction of EQUIV it follows that $u' \notin \alpha$ and u'' is WHITE. Since π leads u'' to v it follows that u'' is distinguishable contradicting the hypothesis on WHITE vertices in $\mathcal{G} \setminus \mathcal{H}$. □

Now we prove that the colouring performed in the main loop is correct, *i.e.*, if it colours a vertex in WHITE (resp. in BLACK) the corresponding states are equivalent (resp. distiguishable).

Lemma 2. *The following hold at the end of each iteration of loop 9–15:*

D1. $\{\langle p, q\rangle \mid \mathrm{COLOR}(\langle p, q\rangle) = \mathrm{BLACK}\} \cap \sim = \varnothing$,
D2. $\{\langle p, q\rangle \mid \mathrm{COLOR}(\langle p, q\rangle) = \mathrm{WHITE}\} \subseteq \sim$.

Proof. Before entering the loop both properties hold by initialization.

D1. Assume (**D1**) and (**D2**) hold before EQUIV(u_0). It is sufficient to prove that at the end of the iteration for every $\langle p, q\rangle \in \mathcal{H}$ we have that $\langle p, q\rangle$ is BLACK if and only if $\langle p, q\rangle$ is distinguishable.
($\rightarrow$) If $u = \langle p, q\rangle \in \mathcal{H}$ is BLACK, then it must have been coloured by VISIT. Therefore, $eq = \bot$ and before REVERSE there was $u \rightsquigarrow h$ in $\mathcal{H}$. Since h was BLACK before the EQUIV-call, by (**D1**) it follows h distinguishable. Thus, $\langle p, q\rangle$ is distinguishable.
($\leftarrow$) If $u = \langle p, q\rangle \in \mathcal{H}$ is distinguishable, then by Proposition 1 it follows that after REVERSE and VISIT pair $\langle p, q\rangle$ has been coloured in BLACK.
D2. It follows from (**D1**) and the fact that all vertices in $\mathcal{H}$ are either BLACK or WHITE. □

We summarize the results of this section as follows:

Theorem 2. *Let $\mathcal{D} = \langle Q, \Sigma, \delta, q_0, F\rangle$ be a DFA, and $\mathcal{W}$ be the set of* WHITE *pairs after any iteration of loop 9–15 of* MINIMIZEDFA(Q, Σ, δ, F). *Then, the following hold:*

1. *Incrementality: $\mathcal{D}/\mathcal{W}^e$ is a partially minimized automaton equivalent to $\mathcal{D}$, and*
2. *Correctness: if $\mathcal{W}$ is from the last iteration, then $\mathcal{D}/\mathcal{W}^e$ is the minimum automaton accepting $L(\mathcal{D})$.*

Proof. Incrementality follows from Lemma 2. Correctness follows from both Lemma 2 and the fact that, after all iterations, every vertex of $\mathcal{G}$ has been (correctly) coloured either in BLACK or WHITE. □

5. Nondeterministic case

Before diving into the algorithm, please remember (as for Section 4) that the completeness assumption for automata is just for ease of explanation.

Algorithm 2 is not directly applicable to the nondeterministic case, the reason being that reaching a pair of non-bisimilar states—i.e. sufficient condition to colour in BLACK a node by Algorithm 2—is not a sufficient condition now to declare a pair of states distinguishable.

To tackle this issue we first turn the associated graph into a bipartite graph. In the definition below, for each state p we introduce the *shadow* state $\overline{p}$ as a distinct copy of the *real* p.

Definition 5. Let $\mathcal{N}$ be a complete NFA. The *associated graph* $\mathcal{G}(\mathcal{N})$ is a bipartite directed graph with vertices $V_0 \cup V_1$ and arcs $A_0 \cup A_1$, defined as:

$$
\begin{aligned}
V_0 &= Q \times Q,\\
V_1 &= \left\{ \left\langle p, \overline{q}, x \right\rangle, \left\langle \overline{p}, q, x \right\rangle \mid p, q \in Q,\ x \in \Sigma \right\},\\
A_0 &= \left\{ \langle p, q\rangle \to \left\langle p', \overline{q}, x \right\rangle, \langle p, q\rangle \to \left\langle \overline{p}, q', x \right\rangle \mid p' \in \delta(p, x),\ q' \in \delta(q, x) \right\},\\
A_1 &= \left\{ \left\langle p', \overline{q}, x \right\rangle \to \left\langle p', q' \right\rangle, \left\langle \overline{p}, q', x \right\rangle \to \left\langle p', q' \right\rangle \mid p' \in \delta(p, x),\ q' \in \delta(q, x) \right\}.
\end{aligned}
$$

Vertex $\langle p, q\rangle$ in the "left" V_0 is called *equivalent* (resp. *distinguishable*) whenever states p and q are bisimilar (resp. non-bisimilar). Furthermore, it is called *trivially distinguishable* when exactly one of the two is final.

The bisimilarity between p and q (Definition 2) can be checked in two steps: 0) choose $x \in \Sigma$ and $p' \in \delta(p, x)$, and 1) respond with suitable $q' \in \delta(q, x)$. The idea is to mimic step 0) traversing arcs of A_0 and step 1) traversing arcs of A_1. The triplet $\left\langle p', \overline{q}, x \right\rangle \in V_1$ indicates that we have chosen symbol x, state $p' \in \delta(p, x)$, and we are expecting to respond with some $q' \in \delta(q, x)$ ($\overline{q}$ provides the information on the state that must respond). A_1 arcs do something similar.

Then, we have to adjust the BLACK colouring of vertices in $\mathcal{G}$ to conform to nondeterminism. Observe that $u \in V_0$ needs only one BLACK child to be coloured in BLACK, while it needs all children WHITE to be coloured in WHITE. Dually, $u \in V_1$ behaves the same but with reversed colours. A check will be performed using the variable DOUBTS(u) which, roughly speaking, counts how many BLACK neighbours we need to find to mark u as BLACK.

5.1. The algorithm for nondeterministic automata

We present Algorithms 3 and 4 for the nondeterministic case, whose essential ingredients are those of Algorithm 2. Details are left to the reader.

First of all, notice that we are actually dealing with *four* colours: $\bot$ (never been explored), GREY (in visit), BLACK (distinguishable) and WHITE (equivalent). The procedure MINIMIZENFA is structurally the same as MINIMIZEDFA, the difference being the usage and maintenance of $\mathcal{H}$.

Algorithm 3 Proposed algorithm for nondeterministic automata, adapted from DFA case.

```
 1: function MinimizeNfa(Q, Σ, δ, F)
 2:     for all ⟨p, q⟩ ∈ Q × Q do
 3:         if p, q are triv. distinguishable then
 4:             Color(⟨p, q⟩) ← Black
 5:         else if p = q then
 6:             Color(⟨p, q⟩) ← White
 7:         else
 8:             Color(⟨p, q⟩) ← ⊥
 9:     for all u ∈ Q × Q do
10:         H ← EmptyGraph
11:         EquivLeft(u)
12:         for all v ∈ V0 ∩ H do
13:             if Color(v) ≠ Black then
14:                                          ▷ v is either Grey or White
15:                 Color(v) ← White
16:     W ← {u ∈ V0 | Color(u) = White}
17:     return Q/W^e
18: end function

19: procedure Relax(v)                       ▷ Used in Algorithm 4
20:     for u ∈ Adj(H, v) do                 ▷ H is global
21:         Doubts(u) ← Doubts(u) − 1
22:         if Doubts(u) = 0 then
23:             Color(u) ← Black
24:             Relax(u)
25: end procedure
```

Algorithm 4 Procedures to colour the associated graph

```
 1: function EquivLeft(u)                    ▷ H is global
 2:     if Color(u) ≠ ⊥ then
 3:         return Color(u)
 4:
 5:     Color(u) ← Grey
 6:     Doubts(u) ← 0
 7:
 8:     for v ∈ Adj(G, u) ∧ Color(u) ≠ Black do
 9:         col ← EquivRight(v)
10:         if col = Black then
11:             Color(u) ← Black
12:             Doubts(u) ← 0
13:         else if col = Grey then
14:             AddArc(H, v, u)
15:             Doubts(u) ← 1
16:
17:     if Color(u) = Grey then
18:         if Doubts(u) = 0 then
19:             Color(u) ← White
20:
21:     if Color(u) = Black then
22:         Relax(u)
23:     return Color(u)
24: end function

25: function EquivRight(u)                   ▷ H is global
26:     if Color(u) ≠ ⊥ then
27:         return Color(u)
28:
29:     Color(u) ← Grey
30:     Doubts(u) ← 0
31:
32:     for v ∈ Adj(G, u) ∧ Color(u) ≠ White do
33:         col ← EquivLeft(v)
34:         if col = White then
35:             Color(u) ← White
36:             Doubts(u) ← 0
37:         else if col = Grey then
38:             AddArc(H, v, u)
39:             Doubts(u) ← Doubts(u) + 1
40:
41:     if Color(u) = Grey then
42:         if Doubts(u) = 0 then
43:             Color(u) ← Black
44:
45:     if Color(u) = Black then
46:         Relax(u)
47:     return Color(u)
48: end function
```

Function Equiv of Algorithm 2 is now split into two separate (and mutually recursive) functions, EquivLeft and EquivRight, which test the equivalence of pairs of states respectively in V_0 and V_1.

Function EquivLeft takes as input the current vertex $u \in V_0$. If u has already been encountered we return its colour. Otherwise, it is coloured in Grey with zero Doubts. At lines 8–15 each successor v of u is recursively visited. Since $u \in V_0$, if v is recursively found Black, then u can be safely marked Black. Otherwise, there is not enough information to safely assign a Black/White colour to u. In particular, if v is Grey we add arc $v \to u$ to $\mathcal{H}$ (notice that it is reversed *w.r.t.* the transition) and we set Doubts(u) to 1—Blackness of u depends on the (possible) future Blackness of one of its neighbours v.

After the loop, at lines 17– 19, if u is still Grey we consider two cases: if Doubts(u) = 0, then each of its successors has the same colour (in this case White) which can be safely assigned to u.

In case u was coloured Black (lines 21–23) this information is propagated (Relaxed) to its neighbours in $\mathcal{H}$. Notice that in this case we explicitly define the procedure Relax (pseudocode of Algorithm 3): in Algorithm 2 the corresponding procedure Visit's purpose, was to colour in Black all vertices reachable from some distinguishable vertex. In Algorithm 3 we must consider the doubts of each vertex v by colouring in Black only non-doubtful vertices.

EquivRight is similar to EquivLeft, the key difference being the update of Doubts(u) (line 39): vertices from V_1 need *only one* White neighbour to prove their Whiteness, while *each* of their neighbours must be Black to prove their Blackness.

As an extra (implementation) detail, notice that EquivLeft and EquivRight can be combined into a single Equiv(u, s) function whose second parameter, $s \in \{0, 1\}$, manages the *side* of $\mathcal{G}$ we are visiting (either left or right).

5.2. Complexity analysis

First, we bound the size of the associated graph.

Lemma 3. *Given a complete NFA with n states and $m \geq n|\Sigma|$ transitions, its associated graph has size $|\mathcal{G}| \leq 7nm$.*

Proof. The associated graph has size $|\mathcal{G}| = |V_0| + |V_1| + |A_0| + |A_1|$. Clearly, $|V_0| = n^2$ and $|V_1| = 2n^2|\Sigma|$. Arcs are more delicate:

$$\begin{aligned} |A_0| &= \sum_{p\in Q}\sum_{q\in Q}\sum_{x\in\Sigma}\Big(|\delta(p,x)| + |\delta(q,x)|\Big) \\ &= n\sum_{p\in Q}\sum_{x\in\Sigma}|\delta(p,x)| + n\sum_{q\in Q}\sum_{x\in\Sigma}|\delta(q,x)| \\ &= 2nm \end{aligned}$$

It is not difficult to see that $|A_1| = |A_0|$. The claim follows since $n|\Sigma| \leq m$ (the automaton is complete). □

It is evident that EquivLeft and EquivRight combined perform the equivalent of a visit of $\mathcal{G}$, and Relax visits every arc of $\mathcal{H}$ at most once. Hence, their cost over all the execution of Algorithm 3 is bounded by the size of $\mathcal{G}$.

Finally, the computation of $Q/\mathcal{W}^e$ has time complexity $\mathcal{O}(n^2)$ as in the deterministic case.

Theorem 3. *Given a complete NFA with n states and $m \geq n|\Sigma|$ transitions,* MinimizeNfa *terminates in time $\mathcal{O}(nm) \subseteq \mathcal{O}(n^2|\Sigma|r)$.*

5.3. Correctness

First, we prove that from a White vertex on the left we reach in two hops, again, a White vertex.

Proposition 2. *Let $p \neq q$, and $u = \langle p, q\rangle \in V_0$ be* White. *Then, for every arc $u \to v \in A_0$, vertex $v \in V_1$ is either* Grey *or* White.

Proof. Since $p \neq q$, there are only two places at which $u = \langle p, q\rangle$ changed from Grey to White:

Line 15 of Algorithm 3.
In this case, upon termination of EquivLeft(u), u is Grey and Doubts(u) > 0 (line 17 of Algorithm 4). Inside the loop of its EquivLeft-call every neighbour was recursively found to be either Grey or White—or u would have been coloured in Black. Finally, it cannot be the case that some Grey neighbour v became Black afterward, since Relax(v) would have coloured u in Black by setting Doubts(u) $= 0$.

Line 19 of Algorithm 4.
In this case, inside the loop of EquivLeft(u) none of u's neighbours were found to be either Black or Grey. Thus, they must all be White.

Therefore, every neighbour of u is either Grey or White. □

Proposition 3. *At line 15 of Algorithm 3 (end of the main loop), for every $v \in V_1$, if v is either* Grey *or* White, *then there exists $v \to u' \in A_1$ such that $u' \in V_0$ is* White.

Proof. If v is White, then it must have been coloured at line 35 of Algorithm 4 after finding a White neighbour. If v is Grey, upon termination of EquivLeft at line 11 (Algorithm 4) some of its neighbours must have been found to be Grey. Since every Grey left node is coloured in White before the end of the iteration, it follows again that v has some White neighbour. □

As in the DFA case, the colouring performed by loop 9–15 is correct.

Lemma 4. *The following hold at the end of each iteration of loop 9–15 of Algorithm 3:*

N1. $\mathcal{R}_W = \{\langle p, q\rangle \mid \text{Color}(\langle p, q\rangle) = \text{White}\} \subseteq \mathcal{B}$,
N2. $\mathcal{R}_B = \{\langle p, q\rangle \mid \text{Color}(\langle p, q\rangle) = \text{Black}\} \cap \mathcal{B} = \emptyset$.

Proof.

N1. By Lemma 1 it is sufficient to prove that $\mathcal{R}_W$ is a bisimulation.
First, notice that pairs violating (**B1**) are coloured in Black from the start. Consider $\langle p, q\rangle \in \mathcal{R}_W$. If $p = q$, (**B2**) and (**B3**) trivially hold. Otherwise, let $x \in \Sigma$ and $p' \in \delta(p, x)$. From Proposition 2 it follows that $v = \langle p', \overline{q}, x\rangle$ is either Grey or White. From Proposition 3 it follows that some neighbour u' of v is in $\mathcal{R}_W$. By Definition 5 we have $u' = \langle p', q'\rangle$ for some $q' \in \delta(q, x)$. Thus, (**B2**) holds for $\langle p, q\rangle$. The very same argument can be used to prove that (**B3**) holds for $\langle p, q\rangle$. Hence, $\mathcal{R}_W$ is a bisimulation.
N2. The result follows from (**N1**) and the fact that vertices in $V_0 \cap \mathcal{H}$ are either Black or White. □

At the exit of the loop, all vertices have been coloured. As for the deterministic case, we link the behaviour of MinimizeNfa to the correctness of $\mathcal{R}_W$:

Theorem 4. *Let $\mathcal{N} = \langle Q, \Sigma, \delta, I, F\rangle$ be a NFA, $\mathcal{G} = (V_0 \cup V_1, A_0 \cup A_1)$ be its associated graph, and $\mathcal{R}_W \subseteq V_0$ be the set of* White *pairs after any iteration of loop 9–15 of* MinimizeNfa(Q, Σ, δ, F)*. Then, the following hold:*

1. *Incrementality: $\mathcal{N}/\mathcal{R}^e_W$ is a partially minimized automaton equivalent to $\mathcal{N}$, and*
2. *Correctness: if $\mathcal{R}_W$ is from the last iteration, then $\mathcal{N}/\mathcal{R}^e_W$ is the bisimulation-minimum automaton accepting $L(\mathcal{N})$—equivalently, $\mathcal{R}_W = \mathcal{B}$.*

Proof. Let $\mathcal{R}_B$ (resp. $\mathcal{R}_W$) be the set of pairs from V_0 which have been coloured in Black (resp. White). Incrementality follows from (**N1**) of Lemma 4 ($\mathcal{R}_W \subseteq \mathcal{B}$ is a bisimulation).

Correctness follows from both Lemma 4 and the fact that, upon termination, $\mathcal{R}_B, \mathcal{R}_W$ is a partition of V_0. Therefore:

$$\begin{aligned}\mathcal{B} &= V_0 \cap \mathcal{B}\\ &= (\mathcal{R}_B \cup \mathcal{R}_W) \cap \mathcal{B}\\ &= (\mathcal{R}_B \cap \mathcal{B}) \cup (\mathcal{R}_W \cap \mathcal{B})\\ &= \varnothing \cup (\mathcal{R}_W \cap \mathcal{B})\\ &\subseteq \mathcal{R}_W,\end{aligned}$$

and we conclude $\mathcal{R}_W = \mathcal{B}$. □

6. Signatures of states

In this section we introduce the *signature of a state* with the idea of reducing the number of pairs to be checked by the minimization algorithm.

Definition 6. Let $\mathcal{N} = \langle Q, \Sigma, \delta, I, F\rangle$ be a NFA, $\mathcal{L}$ be a (possibly infinite) set of *labels*, $B \subseteq Q \times Q$ be any bisimulation equivalence, and $\sigma : Q \to \mathcal{L}$. We say that σ is a *signature* if:

$$(\forall p, q \in Q)(pBq \to \sigma(p) = \sigma(q)).$$

Furthermore, we call $\sigma(p)$ the *signature of* p.

Ideally, we would like a signature with the property that two states are equivalent if and only if they have identical signatures. Since this would be too much to ask—at that point we would have $\mathcal{B}$—we are happy with less informative signatures. More precisely, we assume to be able to spend a reasonable amount of time (*e.g.* linear in the size of the automaton) to pre-compute a signature σ and consider as Black (see Algorithms 2 and 3) all pairs of states with different signatures: this will not change the asymptotic complexity of the algorithm in the worst case, but could considerably improve the complexity for many automata.

The next lemma, whose proof can be easily derived from Definition 6, formalizes the above observation.

Lemma 5. *Let σ be a signature and consider the relation $=_\sigma$ defined as:*

$$p =_\sigma q \overset{\text{def}}{\iff} \sigma(p) = \sigma(q)$$

Then, $=_\sigma$ is an equivalence relation and $\mathcal{B} \subseteq =_\sigma$.

Thus, instead of blindly call Equiv on every pair of states, the function will be called on states equivalent by signature. All other pairs will be considered Black by default.

6.1. A simple signature

We now focus our attention on defining a simple signature computable in linear time.

Definition 7. Let $p \in Q$ be a state. We define $l^*(p)$ as the minimum length of a string accepted by p and $L^*(p)$ as the set of such minimum-length strings:

$$\begin{aligned}l^*(p) &= \min\{|w| \mid w \in L(p)\}\\ L^*(p) &= \left\{w \in L(p) \mid |w| = l^*(p)\right\}\end{aligned}$$

We put $l^*(p) = \infty$ and $L^*(p) = \emptyset$ in case $L(p) = \emptyset$. Moreover, we define the application $\sigma_l : Q \to (\Sigma^* \cup \{\infty\})$ as:

$$\sigma_l(p) = \begin{cases} \infty & \text{if } L(p) = \emptyset, \\ \min_{\prec} L^*(p) & \text{otherwise} \end{cases}$$

Lemma 6. *The application σ_l of Definition 7 is well defined and it is a signature.*

Proof. Well-definedness follows from the fact that a non-empty set of strings admits a unique lexicographically minimum element. Consider now two bisimilar states p and q. Then, $L(p) = L(q)$ and it follows $\sigma_l(p) = \sigma_l(q)$. □

Minimum-length strings accepted by p can be expressed in terms of minimum-length strings accepted by states reachable from p.

Lemma 7. *Let $p \in Q$ be a state. Then, exactly one of the following is true:*

1. $l^*(p) = 0$ *and* $L^*(p) = \{\epsilon\}$, *or*
2. $l^*(p) = \infty$ *and* $L^*(p) = \emptyset$, *or*
3. $0 < l^*(p) < \infty$ *and*
   $L^*(p) = \left\{ x \cdot w \mid x \in \Sigma,\ p' \in \delta(p, x),\ l^*(p) = l^*(p') + 1,\ w \in L^*(p') \right\}$.

Proof. Cases 1 and 2 are obvious. Consider $p \in Q$ with $0 < l^*(p) < \infty$.

First, notice that the ($\supseteq$)-inclusion is obvious. We prove the other inclusion. By Definition 7, $L^*(p)$ is non-empty and all its strings are of length $l^*(p) \geq 1$. Consider any such string $x \cdot w \in L^*(p)$. Since xw is accepted by p, there has to be a path in the automaton that starts in p, spells xw and reaches a final state. Consider any such path and let $p' \in \delta(p, x)$ be the state next to p in that path. Obviously, w is accepted by p' and so it has to be $l^*(p') \leq |w|$. Furthermore, there cannot exists any shorter string w' accepted by p', or a shorter string $(x \cdot w')$ would also be accepted by p. Thus, $l^*(p') = |w| = l^*(p) - 1$. □

Corollary 1. *Let $p \in Q$ be a non-final state with $L^*(p) \neq \emptyset$. Then, there exist $x \in \Sigma$ and $p' \in \delta(p, x)$ such that $\sigma_l(p) = x \cdot \sigma_l(p')$.*

Proof. State p must be of type 3 of Lemma 7, thus $\sigma_l(p) = x \cdot w$ for some $w \in L^*(p')$ and $p' \in \delta(p, x)$. Since xw is lexicographically minimum for p, w must be lexicographically minimum for p'. Therefore, $\sigma_l(p') = w$. □

Corollary 1 constitutes a recursive rewriting of signature σ_l.

6.2. The algorithm for the simple signature

We devise Algorithm 5 to compute $Q/=_{\sigma_l}$ in time proportional to the size of the automaton $\mathcal{A}$. For clarity of exposition, we will consider the case of a deterministic automaton with a single final state: the algorithm can be easily generalized for the case of a NFA with multiple final states.

Function ComputeSigmaLPartition is constituted by four main operations. First, the automaton is Reversed, *i.e.* transitions $\delta(p, x) = q$ are transformed into transitions $\tau(q, x) = p$. Afterwards, at line 7, this visit computes an array $dist$ which contains the values of l^*. Next, at line 8, ComputeSuccessors returns the array $succ$ that contains, for each state p, the first symbol of $\sigma_l(p)$ (with the special case for $\sigma_l(f) = \epsilon$). Finally, ComputeBuckets partitions the states into the σ_l-classes.

Let us move on ComputeSuccessors. In virtue of Corollary 1, it is sufficient to find the lexicographically minimum $x \in \Sigma$ that leads p of distance $dist[p] \neq \infty$ to some p' of smaller distance. The final state f is covered by initialization of $succ$ (lines 34–35). The for-loop 37–44 takes care of non-final states at finite distance.

The function Sig(p) returns a pair $\langle x, i \rangle$ that encodes $\sigma_l(p)$ in the following sense: x is the first symbol of $\sigma_l(p)$ and i is the index of the bucket associated with $\sigma_l(\delta(p, x))$.

Let us describe ComputeBuckets, which is responsible of returning the σ_l-partition. The partition is maintained in the list $Buckets$ initially containing the class of the final state f. The array B stores the indices of the bucket of each state. The index of the new bucket to be constructed is stored in variable $next$. The variable d stores next distance to be inspected. If two states p and q have distance $dist[p] \neq dist[q]$, then they must belong to different buckets. For this reason, states are first sorted by distance (line 17) and intervals of states at the same distance d are further inspected. Lines 20–21 are responsible of finding the interval $Q[i \dots j]$ to inspect. $Q[i \dots j]$ is then sorted (according to Sig-value) at line 22 and is given as input to SplitInterval. After the loop, the bucket of the remaining states—those with infinite distance—is added to the partition, which is then returned.

The function SplitInterval takes as input a sorted interval of states and partitions them according to their Sig-value. First, a fresh bucket b is created. b will contain all states with Sig-value sig—notice that states with the same value are stored consecutively in the array. Then, each state Q_k in the interval is considered one at a time and its Sig-value is stored in $curr$. If $curr = sig$, then Q_k is added to bucket b. Otherwise, the new sig is saved and Q_k constitutes the start of a new bucket. The old b is added to the list of buckets, it is re-initialized with the singleton $\{Q_k\}$ and, since a new bucket is being created, $next$ is incremented. In any case, the bucket-index of Q_k is updated at line 60. At the exit of loop 51–60, the last bucket is added and the value of $next$ for the subsequent iterations is returned.

Algorithm 5 Given a DFA $\mathcal{A}$ with states Q, alphabet Σ, transitions δ and single final state f, returns $Q/=_{\sigma_l}$.

```
 1: Q, Σ, δ, f, dist, succ, Buckets, B
 2:                                                  ▷ Global variables
 3:
 4: function ComputeSigmaLPartition(A)
 5:     Q, Σ, δ, f ← A
 6:     τ ← Reverse(δ)
 7:     dist ← BFS(Q, τ, f)
 8:     succ ← ComputeSuccessors()
 9:     return ComputeBuckets()
10: end function
11:
12: function ComputeBuckets()
13:     Buckets ← { {f} }
14:     B ← Array(|Q|)
15:     B[f] ← 1
16:     i, next, d ← (2, 2, 1)
17:     Q ← LinearSortByDistance(Q)
18:     while i ≤ |Q| ∧ d < ∞ do
19:         j ← i
20:         while j + 1 ≤ |Q| ∧ dist[Q_{j+1}] = d do
21:             j ← j + 1
22:         Q[i ... j] ← LinearSortBySig(Q, i, j)
23:         next ← SplitInterval(next, i, j)
24:         i, d ← (j + 1, d + 1)
25:     Buckets ← Buckets ∪ {q ∈ Q | dist[q] = ∞}
26:     return Buckets
27: end function
28:
29: function Sig(p)
30:     p' ← δ(p, succ[p])
31:     return ⟨succ[p], B[p']⟩
32: end function
33: function ComputeSuccessors()
34:     succ ← Array(|Q|, ∞)
35:     succ[f] ← ε
36:     Q' ← Q \ ({f} ∪ {q ∈ Q | dist[q] = ∞})
37:     for all p ∈ Q' do
38:         d ← dist[p]
39:         found ← ⊥
40:         for all x ∈ Σ in lex order ∧ ¬found do
41:             p' ← δ(p, x)
42:             if dist[p'] + 1 = d then
43:                 succ[p] ← x
44:                 found ← ⊤
45:     return succ
46: end function
47:
48: function SplitInterval(next, i, j)
49:     b ← ∅
50:     sig ← Sig(Q_i)
51:     for k = i to j do
52:         curr ← Sig(Q_k)
53:         if curr = sig then
54:             b ← b ∪ {Q_k}
55:         else
56:             sig ← curr
57:             Buckets ← Buckets ∪ {b}
58:             b ← {Q_k}
59:             next ← next + 1
60:         B[Q_k] ← next
61:     Buckets ← Buckets ∪ {b}
62:     return next + 1
63: end function
```

6.3. Complexity analysis

Remember that we are dealing with a complete deterministic automaton with n states and $m = n|\Sigma|$ transitions.

The time complexity of ComputeSigmaLPartition is the sum of the time complexities of Reverse, BFS, ComputeSuccessors and ComputeBuckets.

Reverse and BFS are standard algorithms that run in time $\mathcal{O}(n+m)$.

ComputeSuccessors takes time $\mathcal{O}(n+m)$. The initialization of $succ$ and Q' has cost $\mathcal{O}(n)$. Afterwards, the cost of each iteration of loop 37–44 is bounded by the size of the alphabet. Summing over all iterations, the running time is bounded by $\mathcal{O}(m)$.

SplitInterval has time complexity linear in the size $j - i + 1$ of the interval is called on: b and $Buckets$ can be implemented as linked-lists, thus, each insertion can be done in constant time. Therefore, each iteration of loop 51–60 has cost $\mathcal{O}(1)$.

ComputeBuckets takes time $\mathcal{O}(n)$. The sort at line 17—which can be implemented as a radix-sort—runs in time $O(n)$. Afterwards, the loop 20–21 acts on disjoint intervals, thus its cost is the sum of the costs of each interval. Fix an interval $Q[i \dots j]$. Sorting at line 22—again, radix-sort—costs $\mathcal{O}(j - i + 1)$, the same as SplitInterval$(next, i, j)$. Thus, summing over all intervals we obtain $\mathcal{O}(n)$.

Concluding, we have the following:

Theorem 5. ComputeSigmaLPartition *terminates in time* $\Theta(n|\Sigma|)$.

6.4. Correctness

Reverse, BFS, LinearSortByDistance and LinearSortBySig may be assumed to be correct because they are minor variations of well-known algorithms.

For what concerns ComputeSuccessors, loop 37–44 operates over non-final states p with $L^*(p) \neq \emptyset$. Thus, Corollary 1 ensures that the inner **if**-guard is satisfied for the correct x.

In what follows, we say that *array B is correct up to d*, for $d \geq 1$ if:

$$(\forall q_1, q_2 \in \{q \in Q \mid dist[q] < d\})(B[q_1] = B[q_2] \leftrightarrow q_1 =_{\sigma_l} q_2).$$

Now we prove that function Sig(p) correctly encodes the signature $\sigma_l(p)$. This is the essence of the correctness of both SplitInterval and ComputeBuckets.

Lemma 8. *Let $p \in Q$ such that $1 \leq dist[p] < \infty$. If array B is correct up to $dist[p]$, then:*

$$\text{Sig}(p) = \langle x, B[p'] \rangle \iff \sigma_l(p) = x \cdot \sigma_l(p').$$

Proof. By definition of SIG(p), it holds $p' = \delta(p, succ[p])$—which exists by hypothesis about $dist[p]$. The statement follows from $succ$ correctness and from Corollary 1. □

Corollary 2. *Let $p, q \in Q$ with $dist[p] = dist[q]$ and suppose p, q and B satisfy the hypotheses of Lemma 8. Then:*

$$\text{SIG}(p) = \text{SIG}(q) \iff p =_{\sigma_l} q.$$

Proof. Let $\text{SIG}(p) = \langle x, B[p'] \rangle$, and $\text{SIG}(q) = \langle y, B[q'] \rangle$. Then:

$$\begin{aligned}
\text{SIG}(p) = \text{SIG}(q) &\iff x = y \wedge B[p'] = B[q'] && \text{(Equality of pairs)} \\
&\iff x = y \wedge \sigma_l(p') = \sigma_l(q') && (B \text{ is correct}) \\
&\iff x \cdot \sigma_l(p') = y \cdot \sigma_l(q') \\
&\iff \sigma_l(p) = \sigma_l(q). && \text{(Lemma 8)}
\end{aligned}$$ □

Having proved the correctness of function SIG, the correctness of SPLITINTERVAL follows immediately. Assume B to be correct up to $1 \leq d < \infty$ and let i, j such that interval $Q[i \dots j]$ contains all and only states at distance d ordered by SIG-value. SPLITINTERVAL($next, i, j$) puts in the same bucket states that agree on SIG-value: the correctness follows from Corollary 2. Moreover, after loop 51–60, B is clearly correct up to $d + 1$.

Finally, we consider COMPUTEBUCKETS. The core is the correctness of loop 18–24.

Lemma 9. *After each iteration of loop 18–24, B is correct up to d.*

Proof. We prove the statement by induction on $d \geq 1$.

Base: Before entering the loop, line 15 ensures that B is correct up to $1 = d$.

Step: Consider $d > 1$ and assume B correct up to d from the previous iteration. After loop 20–21, $Q[i \dots j]$ clearly contains all and only states at distance d (they are sorted at line 17). Line 22 correctly sorts the interval *w.r.t.* SIG-value. By the argument above, SPLITINTERVAL sets B correct up to $d + 1$. Finally, the assignment at line 24 increments d. Thus, B is now correct up to d. □

Lemma 9 ensures that states at distance $d < \infty$ are correctly put into buckets. Finally, line 25 adds the bucket of infinite-distance states which are all σ_l-equivalent (see Definition 7).

Therefore, we have the following:

Theorem 6. COMPUTESIGMALPARTITION *is correct.*

7. Conclusions

Bisimilarity is a fundamental (equivalence) relation among the states of finite automata, finding applications and variants in a number of different areas. Algorithms for computing bisimilarity are a *classic* and can be subdivided in two categories: top-down and a bottom-up. The former (partition refinement) approach starts with a coarse partition and refines it until the result is produced, while the latter (partition aggregation) starts from a singleton-classes equivalence relation and merges classes until possible.

Although algorithms belonging to the bottom-up category are, to the best of our knowledge, still currently asymptotically slower than their alternative ones, aggregation based techniques enjoy the property of being *incremental*: automata resulting at intermediate stages of the computation are *partially minimized yet language-equivalent to the input one*.

Moreover, partition aggregation algorithms, even though less celebrated than partition refinement ones, introduced by Hopcroft and generalized by Paige and Tarjan, are interesting (at least) for two reasons. The first is theoretical: if two methods compute the same relation (just one from “above” and the other from “below”), why is there a complexity gap? Is there some (hidden) *cost* involved in maintaining incrementality? The second is practical: some applicative contexts can greatly benefit from having a partially minimized *equivalent* automaton, especially when, as alternative, long sequences of refinement steps are involved.

In this work, while fixing a minor mistake in the algorithm by Almeida et al., we reduced bisimilarity computation to a colouring problem on an associated graph. We then extended the algorithm to nondeterministic case, obtaining a complexity improvement on the best known bound for this case. Furthermore, we introduced the notion of signature of a state with the aim of improving both top-down and bottom-up algorithms.

As future works, we would like to experimentally study the proposed algorithms and to investigate on different signatures in both refinement-based and aggregation-based techniques. As a further line of research, it will be interesting to study the effect of applying the technique introduced here to the *colour refinement algorithm* (a.k.a. Weisfeiler-Leman-1 algorithm, see [18]), currently implemented using an algorithm by Cardon and Crochemore (see [19]) belonging to the top-down/partition-refinement category.

CRediT authorship contribution statement

Christian Bianchini: Conceptualization, Formal analysis, Writing – original draft, Writing – review & editing. **Alberto Policriti:** Conceptualization, Formal analysis, Supervision, Writing – original draft, Writing – review & editing. **Brian Riccardi:** Conceptualization, Formal analysis, Writing – original draft, Writing – review & editing. **Riccardo Romanello:** Conceptualization, Formal analysis, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] E.F. Moore, Gedanken-experiments on sequential machines, in: C.E. Shannon, J. McCarthy (Eds.), in: Automata Studies. (AM-34), vol. 34, Princeton University Press, Princeton, 1956, pp. 129–154.
[2] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to automata theory, languages, and computation, ACM SIGACT News 32 (1) (2001) 60–65.
[3] A.R. Meyer, L.J. Stockmeyer, The equivalence problem for regular expressions with squaring requires exponential space, in: 13th Annual Symposium on Switching and Automata Theory, College Park, Maryland, USA, October 25-27, 1972, IEEE Computer Society, 1972, pp. 125–129.
[4] R. Paige, R.E. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16 (6) (1987) 973–989, https://doi.org/10.1137/0216062.
[5] J. Hopcroft, An n log n algorithm for minimizing states in a finite automaton, in: Z. Kohavi, A. Paz (Eds.), Theory of Machines and Computations, Academic Press, 1971, pp. 189–196.
[6] J.F. Groote, J. Martens, E.P. de Vink, Lowerbounds for bisimulation by partition refinement, Log. Methods Comput. Sci. 19 (2) (2023), https://doi.org/10.46298/LMCS-19(2:10)2023.
[7] B. Watson, Taxonomies and toolkits of regular language algorithms, Ph. D. thesis, Mathematics and Computer Science, 1995.
[8] B.W. Watson, J. Daciuk, An efficient incremental DFA minimization algorithm, Nat. Lang. Eng. 9 (1) (2003) 49–64, https://doi.org/10.1017/S1351324903003127.
[9] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM 22 (1975) 215–225, https://doi.org/10.1145/321879.321884.
[10] J.E. Hopcroft, R.M. Karp, A Linear Algorithm for Testing Equivalence of Finite Automata, Computer Science Technical Reports, vol. 114, Cornell University, 1971, https://hdl.handle.net/1813/5958.
[11] M. Almeida, N. Moreira, R. Reis, Incremental dfa minimisation, RAIRO Theor. Inform. Appl. 48 (2) (2014) 173–186, https://doi.org/10.1051/ita/2013045.
[12] J. Björklund, L. Cleophas, Minimization of finite state automata through partition aggregation, in: F. Drewes, C. Martín-Vide, B. Truthe (Eds.), Language and Automata Theory and Applications - 11th International Conference, LATA, in: Lecture Notes in Computer Science, vol. 10168, Springer International Publishing, 2017, pp. 223–235.
[13] C. Bianchini, A. Policriti, B. Riccardi, R. Romanello, Incremental NFA minimization, in: U.D. Lago, D. Gorla (Eds.), Proceedings of the 23rd Italian Conference on Theoretical Computer Science, ICTCS 2022, Rome, Italy, September 7–9, 2022, in: CEUR Workshop Proceedings, vol. 3284, CEUR-WS.org, 2022, pp. 161–173, https://ceur-ws.org/Vol-3284/9606.pdf.
[14] J. Björklund, L. Cleophas, Aggregation-based minimization of finite state automata, Acta Inform. 58 (3) (2021) 177–194, https://doi.org/10.1007/S00236-019-00363-5.
[15] B. Jacobs, J. Rutten, An introduction to (co)algebra and (co)induction, in: D. Sangiorgi, J.J.M.M. Rutten (Eds.), Advanced Topics in Bisimulation and Coinduction, in: Cambridge Tracts in Theoretical Computer Science, vol. 52, Cambridge University Press, 2012, pp. 38–99.
[16] P.C. Kanellakis, S.A. Smolka, Ccs expressions, finite state processes, and three problems of equivalence, in: Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, 1983, pp. 228–240.
[17] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third edition, The MIT Press, 2009.
[18] M. Grohe, K. Kersting, M. Mladenov, P. Schweitzer, Color refinement and its applications, in: An Introduction to Lifted Probabilistic Inference, The MIT Press, 2021.
[19] A. Cardon, M. Crochemore, Partitioning a graph in $\mathcal{O}(|A|\log_2|V|)$, Theor. Comput. Sci. 19 (1) (1982) 85–98, https://doi.org/10.1016/0304-3975(82)90016-0.