Hopcroft and Karp's algorithm for Non-deterministic Finite Automata

Filippo Bonchi, Damien Pous

Hopcroft and Karp's algorithm for Non-deterministic Finite Automata

Filippo Bonchi* and Damien Pous†

November 2011

Abstract

An algorithm is given for determining if two non-deterministic finite automata are language equivalent. We exploit up-to techniques to improve the standard algorithm by Hopcroft and Karp for deterministic finite automata [5], so as to avoid computing the whole deterministic automata. Although the proposed algorithm remains exponential in worst case (the problem is PSPACE-complete), experimental results show that it can be much faster than the standard algorithm: only a very small portion of the determinized automata have to be explored in practice.

Keywords

Language Equivalence, Non-deterministic Finite Automata, Bisimulation, Coinduction, Up-to techniques, Congruence.

1 Introduction

Checking language equivalence of finite automata is a classical problem in computer science, that finds applications in many fields ranging from compilers construction to model checking.

Equivalence of deterministic finite automata (DFA) can be checked either via minimization [4, 2] or through Hopcroft and Karp's algorithm [5], which exploits an instance of what is nowadays called a *coinduction proof principle* [7, 12, 10, 1]: two states recognise the same language if and only if there exists a *bisimulation* relating them. In order to check the equivalence of two given states, Hopcroft and Karp's algorithm creates a relation containing them and tries to build a bisimulation, by adding pairs of states to this relation: if it succeeds then the two states are equivalent, otherwise they are different.

On the one hand, minimization has the advantage of checking the equivalence of all the states at once (while Hopcroft and Karp only of a certain pair of the states). On the other hand, minimization has the disadvantage of

*ENS Lyon, Université de Lyon, LIP (UMR 5668)
†CNRS, Université de Grenoble, LIG (UMR 5217)

needing the whole automata from the beginning, while Hopcroft and Karp's algorithm can be executed "on the fly" on lazy DFA, which are constructed on demand. This difference is fundamental for our work: when starting from non-deterministic finite automata (NFA), the powerset construction used to get deterministic automata induces an exponential factor.

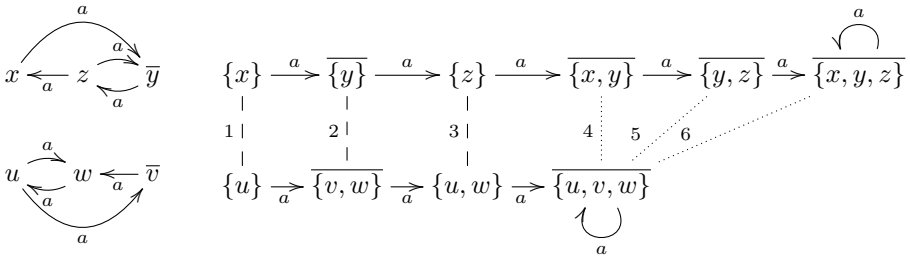
Indeed, the algorithm we introduce in this work for checking equivalence of NFA "usually" does not build the whole deterministic automaton, but just a small part of it. We write "usually" because in few bad cases, the algorithm still needs exponentially many states of the DFA (otherwise we would have solved in polynomial time the problem of language equivalence, which is PSPACE-complete [6]).

Our algorithm is grounded on a simple observation on determinized NFA: for all sets X and Y of states of the original NFA, the union (written $+$) of the language recognised by X (written $[\![X]\!]$) with the language recognised by Y ($[\![Y]\!]$) is equal to the language recognised by the union of X and Y ($[\![X + Y]\!]$). In symbols:

$$[\![X + Y]\!] = [\![X]\!] + [\![Y]\!]$$

This fact allows us to introduce a sound and complete proof technique for language equivalence, namely *bisimulation up to context*, that exploits both *induction* (on the operator $+$) and *coinduction*: if a bisimulation R equates both the (sets of) states X_1, Y_1 and X_2, Y_2, then $[\![X_1]\!] = [\![Y_1]\!]$ and $[\![X_2]\!] = [\![Y_2]\!]$ and, by the above observation, we can immediately conclude that also $X_1 + X_2$ and $Y_1 + Y_2$ are language equivalent. Intuitively, bisimulations up to context are bisimulations which *do not need to relate* $X_1 + X_2$ and $Y_1 + Y_2$ when X_1 (resp. X_2) and Y_1 (resp. Y_2) are already related.

To better illustrate this idea, consider the following example, where we check the equivalence of the states x and u from the NFA depicted below on the left-hand side. (Final states are overlined, labelled edges represent transitions.)



The determinized automaton is depicted on the right-hand side. Each state is a set of states of the NFA, final states are overlined: they contain at least one final state of the NFA. The numbered lines show a relation which is a bisimulation containing x and u. Actually, this is the relation that is built by the standard Hopcroft and Karp's algorithm (the numbers express the order in which each pair is added).

The dashed lines (numbered by 1, 2 and 3) form a smaller relation which is not a bisimulation, but a bisimulation up to context: the equivalence of states

$\{x, y\}$ and $\{u, v, w\}$ could be immediately deduced from the fact that $\{x\}$ is related to $\{u\}$ and $\{y\}$ to $\{y, w\}$, without the need of further exploring the determinized automaton.

Bisimulations up-to, and in particular bisimulations up to context, have been introduced in the context of concurrency theory [7, 8, 11] as a proof technique for bisimilarity of CCS or π-calculus processes. As far as we know, they have never been employed for proving language equivalence of non deterministic automata.

Among these techniques one should also mention *bisimulation up to equivalence*, which, as we show in this paper, is implicitly used in the original Hopcroft and Karp's algorithm. This technique can be briefly explained by noting that not all bisimulations are equivalence relations, and thus, it might be the case that a bisimulation relates (for instance) X and Y, Y and Z but not X and Z. However, since $[\![X]\!] = [\![Y]\!]$ and $[\![Y]\!] = [\![Z]\!]$, we can immediately conclude that X and Z recognise the same language. Analogously to bisimulations up to context, a bisimulation up to equivalence *does not need to relate* X and Z when X and Z are already related to some Y (more generally, when X and Z belong to the equivalence closure of the relation).

The techniques of up-to equivalence and up-to context can be combined resulting in a powerful proof technique which we call *bisimulation up to congruence*. Our algorithm is in fact just an extension of Hopcroft and Karp's algorithm that attempts to build a bisimulation up to congruence instead of a bisimulation up to equivalence.

An important consequence, when using the up to congruence technique, is that we do not need to build the whole deterministic automata, but just those states that are needed for the bisimulation up-to. For instance, in the above NFA, the algorithm stops after equating z and $u + v$ and does not build the remaining four states of the DFA. Despite their use of the up to equivalence technique, this is not the case with Hopcroft and Karp's algorithm, where all accessible subsets of the deterministic automata have to be visited at least once.

Summarising, the contributions of this work are

(1) the observation that Hopcroft and Karp implicitly use bisimulations up to equivalence for DFA (Section 2),

(2) a new sound and complete proof technique for proving language equivalence of NFA (Section 3.1), and

(3) an efficient algorithm for checking language equivalence of NFA (Sections 3.2, 3.3, and 3.4).

Outline. The remaining of the paper is structured as follows. We recall Hopcroft and Karp's algorithm for DFA in Section 2, showing how it can be interpreted in terms of bisimulation up to equivalence. We then describe our algorithm in Section 3, based on bisimulations up to congruence. We discuss the complexity of this algorithm in Section 4; in particular, we provide good

and bad cases, as well as some experimental data showing that the introduced optimisation can be very useful in practice.

Notation. We denote sets by capital letters $X, Y, S, T \ldots$ and functions by lower case letters $f, g, \ldots$ Given sets X and Y, $X \times Y$ is the Cartesian product of X and Y, $X \uplus Y$ is the disjoint union and X^Y is the set of functions $f \colon Y \to X$. Finite iterations of a function $f \colon X \to X$ are denoted by f^n (formally, $f^0(x) = x$, $f^{n+1}(x) = f(f^{n+1}(x))$). The collection of subsets of X is denoted by $\mathcal{P}(X)$. The (omega) iteration of a function $f \colon \mathcal{P}(X) \to \mathcal{P}(X)$ on a powerset is denoted by f^ω (formally, $f^\omega(Y) = \bigcup_{n \geq 0} f^n(Y)$). For a set of letters A, A^* denotes the set of all finite words over A; ϵ the empty word; and $w_1 \cdot w_2$ (and $w_1 w_2$) the concatenation of words $w_1, w_2 \in A^*$. We use 2 to denote the set $\{0, 1\}$ and 2^{A^*} to denote the set of all formal languages over A.

2 Hopcroft and Karp's algorithm for DFA

In this section, we introduce (1) a notion of bisimulation for proving language equivalence of deterministic finite automata, (2) a naive algorithm that automatically checks language equivalence of DFA by mean of bisimulations (Section 2.2) and (3) the standard Hopcroft and and Karp's algorithm (Section 2.3). Moreover, we observe that this algorithm exploits the proof technique of bisimulation up to equivalence (Section 2.4).

A deterministic finite automaton (DFA) over the input alphabet A is a triple (S, o, t), where S is a finite set of states, $o \colon S \to 2$ is the output function, which determines if a state $x \in S$ is final ($o(x) = 1$) or not ($o(x) = 0$), and $t \colon S \to S^A$ is the transition function which returns for each state x and input letter $a \in A$ the next state. For $a \in A$, we will write $x \xrightarrow{a} x'$ to mean that $t(x)(a) = x'$. For $w \in A^*$, we will write $x \xrightarrow{w} x'$ for the least relation such that (1) $x \xrightarrow{\epsilon} x$ and (2) $x \xrightarrow{aw'} x'$ iff $x \xrightarrow{a} x''$ and $x'' \xrightarrow{w'} x'$.

From any DFA, there exists a unique function $[\![-]\!] \colon S \to 2^{A^*}$ mapping states to formal languages, defined as follows for all $x \in S$:

$$[\![x]\!](\epsilon) = o(x)$$
$$[\![x]\!](a \cdot w) = [\![t(x)(a)]\!](w)$$

The language $[\![x]\!]$ is called the language accepted by x. Given two automata (S_1, o_1, t_1) and (S_2, o_2, t_2), the states $x_1 \in S_1$ and $x_2 \in S_2$ are said to be *language equivalent* (written $x_1 \sim x_2$) iff they accept they same language.

In the following, we will always consider the problem of checking the equivalence of states of one single and fixed automaton (S, o, t). We do not loose generality since for any two DFA (S_1, o_1, t_1) and (S_2, o_2, t_2) it is always possible to build the automaton $(S_1 \uplus S_2, o_1 \uplus o_2, t_1 \uplus t_2)$ with

$$o_1 \uplus o_2(x) = \begin{cases} o_1(x) & \text{if } x \in S_1 \\ o_2(x) & \text{if } x \in S_2 \end{cases} \qquad t_1 \uplus t_2(x) = \begin{cases} t_1(x) & \text{if } x \in S_1 \\ t_2(x) & \text{if } x \in S_2 \,, \end{cases}$$

where the language accepted by every state $x \in S_1 \uplus S_2$ is the same as the language accepted by the same state in the original automaton (S_i, o_i, t_i).

For this reason, we also work with automata without explicit initial states: we focus on the equivalence of two arbitrary states of a fixed DFA.

2.1 Proving language equivalence via coinduction

We first define a notion of bisimulation on states. We make explicit the underlying notion of progression which we need in the sequel for up to techniques.

Definition 1 (Progression, Bisimulation)**.** *Given two relations $R, R' \subseteq S \times S$ on states, R* progresses to *R', denoted $R \rightarrowtail R'$ if whenever $x\ R\ y$ then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in A$, $t(x)(a)\ R'\ t(y)(a)$.*

A bisimulation *is a relation R such that $R \rightarrowtail R$.*

As expected, the bisimulation proof technique is sound and complete w.r.t. language equivalence:

Proposition 1. *Two states are language equivalent iff there exists a bisimulation that relates them.*

Proof. Let $R_{[\![-]\!]}$ be the relation $\{(x, y) \mid [\![x]\!] = [\![y]\!]\}$. We prove that $R_{[\![-]\!]}$ is a bisimulation. If $x\ R_{[\![-]\!]}\ y$, then $o(x) = [\![x]\!](\epsilon) = [\![y]\!](\epsilon) = o(y)$. Moreover, for all $a \in A$ and $w \in A^*$, $[\![t(x)(a)]\!](w) = [\![x]\!](a \cdot w) = [\![y]\!](a \cdot w) = [\![t(y)(a)]\!](w)$ that means $[\![t(x)(a)]\!] = [\![t(y)(a)]\!]$, that is $t(x)(a)\ R_{[\![-]\!]}\ t(y)(a)$.

We now prove the other direction. Let R be a bisimulation. We want to prove that $x\ R\ y$ entails $[\![x]\!] = [\![y]\!]$, i.e., for all $w \in A^*$, $[\![x]\!](w) = [\![y]\!](w)$. We proceed by induction on w. For $w = \epsilon$, we have $[\![x]\!](\epsilon) = o(x) = o(y) = [\![y]\!](\epsilon)$. For $w = a \cdot w'$, since R is a bisimulation, we have $t(x)(a)\ R\ t(y)(a)$ and thus $[\![t(x)(a)]\!](w') = [\![t(y)(a)]\!](w')$ by induction. This allows us to conclude since $[\![x]\!](a \cdot w') = [\![t(x)(a)]\!](w')$ and $[\![y]\!](a \cdot w') = [\![t(y)(a)]\!](w')$. □

2.2 Naive algorithm

Figure 1 shows a naive version of Hopcroft and Karp's algorithm for checking language equivalence of the states x and y of a deterministic finite automaton (S, o, t). Starting from x and y, the algorithm builds a relation R that, in case of success, is a bisimulation. In order to do that, it employs the set (of pairs of states) *todo* which, intuitively, at any step of the execution, contains the pairs (x', y') that must be checked: if (x', y') already belongs to R, then it has already been checked and nothing-else should be done. Otherwise, the algorithm checks if x' and y' have the same outputs (i.e., if both are final or not). If $o(x') \neq o(y')$, then x and y are different (because there exists $w \in A^*$ such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$). If $o(x') = o(y')$, then the algorithm inserts (x, y) in R and, for all $a \in A$, the pairs $(t(x)(a), t(y)(a))$ in *todo*.

Naive(x,y)

```
(1)  R is empty; todo is empty;
(2)  insert (x, y) in todo;
(3)  while todo is not empty
 {
      (3.1)  extract (x', y') from todo;
      (3.2)  if (x', y') ∈ R then skip, else {
      (3.3)  if o(x') ≠ o(y') then return false, else {
      (3.4)  for all a ∈ A, insert (t(x')(a), t(y')(a)) in todo;
      (3.5)  insert (x', y') in R; }}
 }
(4)  return true;
```

Figure 1: Naive algorithm for checking the equivalence of states x and y of a DFA (S, o, t); R and *todo* are sets of pairs of states.

For the time being, we avoid to discuss which data structures are convenient for implementing R and *todo* (as well as any complexity issue), but we just focus our attention on the correctness of the algorithm. Just notice that the algorithm terminates since a new pair is added to R at each iteration, and there are finitely many such pairs. (In the sequel, when enumerating iterations, we ignore those where a pair from *todo* is already in R so that there is nothing to do—we can moreover notice that when the algorithm returns true, it necessarily went $1 + |A| \cdot |R|$ times in loop `(3)`, where $|A|$ and $|R|$ respectively denote the size of the alphabet and of the produced relation R.)

Proposition 2. *For all states x and y, we have $x \sim y$ iff* `Naive(`x`,`y`)`.

Proof. We first observe that if `Naive(`x`,`y`)` returns *true* then the relation R that is built before arriving to step `(4)` is a bisimulation. Indeed, the following proposition is an invariant for the loop corresponding to step `(3)`:

$$R \rightarrowtail R \cup \mathit{todo}$$

This invariant is preserved since at any iteration of the algorithm a pair (x', y') is removed from *todo* and inserted in R unless it was already present (after checking that $o(x') = o(y')$ and adding the corresponding derivatives to *todo*). Since *todo* is empty at the end of the loop, we actually have $R \rightarrowtail R$, i.e., R is a bisimulation. By Proposition 1, we deduce $x \sim y$.

We now prove that if `Naive(`x`,`y`)` returns false, then $x \not\sim y$. Note that for all (x', y') inserted in *todo*, there exists a word $w \in A^*$ such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$. Since $o(x') \neq o(y')$, then $[\![x']\!](\epsilon) \neq [\![y']\!](\epsilon)$ and thus $[\![x]\!](w) = [\![x']\!](\epsilon) \neq [\![y']\!](\epsilon) = [\![y]\!](w)$, that is $x \not\sim y$. □

Since both Hopcroft and Karp's algorithm and the one we introduce in Section 3 are simple variations of this naive one, it is important to illustrate its

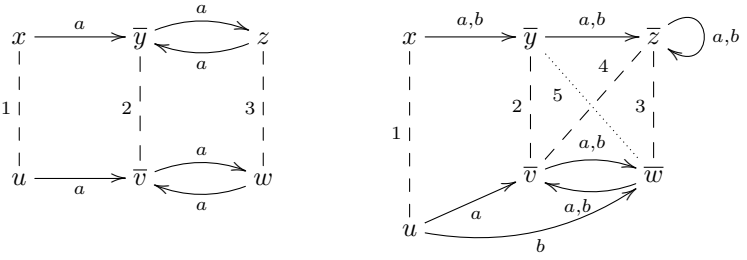


Figure 2: Checking for DFA equivalence, naively.

execution with an example. Consider the DFA with input alphabet $A = \{a\}$ in the left-hand side of Figure 2, and suppose we want to check if x and u are language equivalent. During the initialisation, (x, u) is inserted in *todo*. At the first iteration (of cycle `(3)`), since $o(x) = 0 = o(u)$, (x, u) is inserted in R and (y, v) in *todo*. At the second iteration, since $o(y) = 1 = o(v)$, (u, v) is inserted in R and (z, w) in *todo*. At the third iteration, since $o(z) = 0 = o(w)$, (z, w) is inserted in R and (y, v) in *todo*. At the fourth iteration, since (y, v) is already in R, then the algorithm does nothing. Since there are no more pairs to check (in *todo*), the relation R is a bisimulation and the algorithm terminates returning *true*.

The iterations of the algorithm are concisely described by the numbered dashed lines in Figure 2. The line i means that the connected pair is inserted in R at iteration i.

Remark 1. *Minimization [4, 2] is an alternative way of checking language equivalence; there are two main differences.*

1. *Minimization algorithms equate all the language equivalent states of a given automaton. The above naive algorithm instead checks the equivalence of only two states and, at the end, not all the equivalent states are in the relation R. For instance, the states x and w of the left-hand side example from Figure 2 are also language equivalent, but they are not in the relation R computed by* `Naive(x,u)`.

2. *Minimization algorithms require to know from the beginning the whole automaton (X, o, t), while* `Naive` *can be executed "on the fly" [3]: it can be executed on a lazy DFA, which is constructed on demand. This property is essential for the algorithm that we introduce in Section 3.*

2.3 Hopcroft and Karp's algorithm

The previous naive algorithm is approximately quadratic in the number of states of the DFA: a new pair is added to R at each iteration, and there are only n^2 such pairs, where $n = |S|$ is the number of states of the DFA.

HK(x,y)

```
(1)  R is empty; todo is empty;
(2)  insert (x, y) in todo;
(3)  while todo is not empty
 {
      (3.1)  extract (x', y') from todo;
      (3.2)  if (x', y') ∈ e(R) then skip, else {
      (3.3)  if o(x') ≠ o(y') then return false, else {
      (3.4)  for all a ∈ A, insert (t(x')(a), t(y')(a)) in todo;
      (3.5)  insert (x', y') in R; }}
 }
(4)  return true;
```

Figure 3: Hopcroft and Karp's algorithm for checking the equivalence of states x and y of a DFA (S, o, t); R and *todo* are sets of pairs of states.

To make this algorithm (almost) linear, Hopcroft and Karp actually use a union-find data structure to record a set of equivalence classes rather than a set of visited pairs. With respect to Figure 1, it suffices to update steps `3.2` and `3.5` as follows:

```
(3.2)  if equiv(R, x', y') then ...
(3.5)  merge(R, x', y')
```

As a consequence, their algorithm may stop earlier, when an encountered pair of states is not already in R but in its reflexive, symmetric, and transitive closure. For instance in the right-hand side example from Figure 2, we can skip the fifth pair (y, w), since y and w already belong to the same equivalence class according to the previous four pairs. More generally, there can be at most n iterations (two equivalence classes are merged at each iteration); the algorithm is thus almost linear: by using a union-find data structure, steps `3.2` and `3.5` can be performed in almost constant time [13].

Let $e(R)$ be the symmetric, reflexive, and transitive closure of a binary relation R on states; an alternative way of presenting this algorithm, without considering the concrete union-find data-structure, consists in simply replacing step `3.2` with

```
(3.2)  if (x', y') ∈ e(R) then ...
```

The whole algorithm, named `HK`, is given in Figure 3. We now show that this actually corresponds to using an "up-to technique" to improve the coinductive proof method.

2.4 Bisimulations up-to

Definition 2 (Bisimulation up-to)**.** *Let $f\colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ be a function on relations on S. A relation R is a* bisimulation up to f *if $R \rightarrowtail f(R)$, i.e.,*

whenever $x\ R\ y$ then

1. *$o(x) = o(y)$ and*
2. *for all $a \in A$, $t(x)(a)\ f(R)\ t(y)(a)$.*

With this definition, Hopcroft and Karp's algorithm just consists in building a bisimulation up to e. To prove the correctness of the algorithm it suffices to show that any bisimulation up to e is contained in a bisimulation. We use for that the notion of compatible function [11, 9]:

Definition 3 (Compatible function)**.** *A function $f\colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ on relations on S is* compatible *if it preserves progressions: for all R, R',*

$$R \rightarrowtail R' \text{ entails } f(R) \rightarrowtail f(R')\,.$$

Theorem 1 (Correctness of compatible functions)**.** *Let f be a compatible function. Any bisimulation up to f is contained in a bisimulation.*

Proof. Suppose that R is a bisimulation up to f: $R \rightarrowtail f(R)$. Using compatibility of f and by a simple induction on n, we get $\forall n, f^n(R) \rightarrowtail f^{n+1}(R)$. Therefore, we have

$$\bigcup_n f^n(R) \rightarrowtail \bigcup_n f^n(R)\,,$$

i.e., $f^\omega(R) = \bigcup_n f^n(R)$ is a bisimulation. This latter relation trivially contains R, by taking $n = 0$. □

We could prove directly that e is a compatible function; we however take a detour to ease our correctness proof for the algorithm we propose in Section 3.

Proposition 3 (Compositionality of compatible functions)**.** *The following functions are compatible:*

1. *the identity function* $\mathrm{id} : R \mapsto R$*;*
2. *the composition $f \circ g : R \mapsto f(g(R))$ of compatible functions f and g;*
3. *the union $\bigcup F : R \mapsto \bigcup_{f \in F} f(R)$ of an arbitrary family F of compatible functions.*

Proof. The first two points are straightforward; for the last one, assume that F is a family of compatible functions. Suppose that $R \rightarrowtail R'$; for all $f \in F$, we have $f(R) \rightarrowtail f(R')$ so that $\bigcup_{f \in F} f(R) \rightarrowtail \bigcup_{f \in F} f(R')$. □

As a consequence, the iteration f^ω of a compatible function f is compatible.

Lemma 1. *The following functions are compatible:*

1. *the constant to identity function $r : R \mapsto \{(x, x) \mid \forall x\}$;*

2. *the converse function* $s : R \mapsto \{(y, x) \mid x\ R\ y\}$;

3. *the squaring function* $t : R \mapsto \{(x, z) \mid \exists y, x\ R\ y \wedge y\ R\ z\}$;

Intuitively, given a relation R, $(s \cup \mathrm{id})(R)$ is the symmetric closure of R, $(r \cup s \cup \mathrm{id})(R)$ is its reflexive and symmetric closure, and $(r \cup s \cup t \cup \mathrm{id})^\omega(R)$ is its symmetric, reflexive and transitive closure: we have $e = (r \cup s \cup t \cup \mathrm{id})^\omega$. Another way to understand this decomposition of the symmetric, reflexive, and transitive closure function (e) is to recall that for a given R, $e(R)$ can be defined inductively by the following rules:

$$\frac{}{x\ e(R)\ x}\ r \qquad \frac{x\ e(R)\ y}{y\ e(R)\ x}\ s \qquad \frac{x\ e(R)\ y \qquad y\ e(R)\ z}{x\ e(R)\ z}\ t \qquad \frac{x\ R\ y}{x\ e(R)\ y}\ \mathrm{id}$$

Therefore, together with Proposition 3, Lemma 1 ensures that e is compatible.

Corollary 1. *For all states x and y, we have $x \sim y$ iff* `HK(x,y)`.

Proof. Same proof as for Proposition 2, by using the invariant $R \rightarrowtail e(R) \cup \mathit{todo}$ for the loop. We deduce that R is a bisimulation up to e after the loop. Since e is compatible, R is contained in a bisimulation by Theorem 1. □

As an example, take the automaton on the right of Figure 2. While the naive algorithm constructs the relation

$$R_{Naive} = \{(x, u), (y, v), (z, w), (z, v), (y, w)\},$$

which is a bisimulation, Hopcroft and Karp's algorithm stops one step earlier, resulting in the relation

$$R_{HK} = \{(x, u), (y, v), (z, w), (z, v)\},$$

which is not a bisimulation (because $(x, u) \in R_{HK}$, $x \xrightarrow{b} y$, $u \xrightarrow{b} w$ and $(y, w) \notin R_{HK}$), but a bisimulation up to e (since $(y, w) \in e(R_{HK})$).

Remark 2. *Observe that unlike with the naive algorithm, the relation R built from Hopcroft and Karp's algorithm might change depending on the order in which the pairs (x', y') are processed from the todo list (step* `(3.1)`*). For instance, after inserting (x, u) in R, we might insert (y, w), then (z, v) and finally (z, w), resulting in the following relation:*

$$R'_{HK} = \{(x, u), (y, w), (z, v), (z, w)\}$$

We can however notice that $e(R_{HK}) = e(R'_{HK}) = e(R_{Naive})$. This actually holds in general, whatever the order in which the todo list is processed: we always have

$$R_{HK} \subseteq R_{Naive} \subseteq e(R_{HK})$$

(the first containment holds by definition of the algorithm, the second holds because $e(R_{HK})$ is a bisimulation—proof of Theorem 1, we have $e^\omega = e$). It follows that $e(R_{HK}) = e(R_{Naive})$, e being monotonic and idempotent.

Since $e(R_{HK})$ is obtained by merging equivalence classes, this means that the number of iterations required by `HK` does not depend on the order in which the pairs are processed. This latter property will not hold anymore in the algorithm that we will introduce in Section 3, so that the policy for choosing (x', y') in the step `(3.1)` will be relevant for the efficiency of the algorithm.

3 Optimised algorithm for NFA

We now introduce our optimised algorithm for non deterministic finite automata (NFA). We start with standard definitions about semi-lattices, NFA, determinisation, and language equivalence for NFA.

A *semi-lattice with bottom* $(X, +, 0)$ consists of a set X and a binary operation $+\colon X \times X \to X$ that is associative, commutative, idempotent (ACI) and has $0 \in X$ (the bottom) as identity. Since we will always consider semi-lattices with bottom, hereafter we will avoid to specify every time "with bottom", but we will just write "semi-lattice". Given two semi-lattices $(X_1, +_1, 0_1)$ and $(X_2, +_2, 0_2)$, an homomorphism of semi-lattices $f\colon (X_1, +_1, 0_1) \to (X_2, +_2, 0_2)$ is a function $f\colon X_1 \to X_2$ such that for all $x, y \in X_1$, $f(x +_1 y) = f(x) +_2 f(y)$ and $f(0_1) = 0_2$. The set $2 = \{0, 1\}$ is a semi-lattice when taking $+$ to be the ordinary Boolean or. Also the set of all languages 2^{A^*} carries a semi-lattice where $+$ is the union of languages and 0 is the empty language. More generally, for any set X, $\mathcal{P}(X)$ is a semi-lattice where $+$ is the union of sets and 0 is the empty set.

In the rest of the paper we will indiscriminately use 0 to denote the element $0 \in 2$, the empty language in 2^{A^*} and the empty set in $\mathcal{P}(X)$. Analogously, $+$ will denote the "Boolean or" in 2, the union of languages in 2^{A^*} and the union of sets in $\mathcal{P}(X)$.

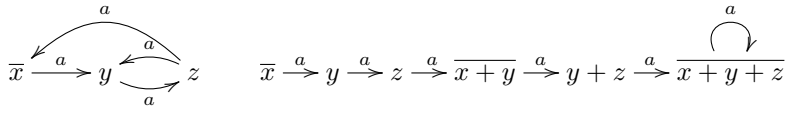
A non-deterministic finite automaton (NFA) over the input alphabet A is a triple (S, o, δ), where S is a finite set of states, $o\colon S \to 2$ is the output function (as for DFA), and $\delta\colon S \to \mathcal{P}(S)^A$ is the transition relation which assigns to each state $x \in S$ and input letter $a \in A$ a set of possible successor states.

The *powerset construction* transforms every NFA (S, o, δ) into the DFA $(\mathcal{P}(S), o^\sharp, \delta^\sharp)$ where $o^\sharp\colon \mathcal{P}(S) \to 2$ and $\delta^\sharp\colon \mathcal{P}(S) \to \mathcal{P}(S)^A$ are defined for all $X \in \mathcal{P}(S)$ as

$$o^\sharp(X) = \begin{cases} o(x) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ o^\sharp(X_1) + o^\sharp(X_2) & \text{if } X = X_1 + X_2 \end{cases}$$

$$\delta^\sharp(X)(a) = \begin{cases} \delta(x)(a) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ \delta^\sharp(X_1)(a) + \delta^\sharp(X_2)(a) & \text{if } X = X_1 + X_2 \end{cases}$$

For an example consider the NFA (S, o, δ) depicted on the left below. Part of the corresponding DFA is depicted on the right, where we use a new notation: states are denoted by expressions of the form $x_1 + \cdots + x_n$ corresponding to the set $\{x_1, \ldots, x_n\}$ (thus x corresponds to $\{x\}$ and 0 to the empty set). Like previously, expressions are overlined iff they are final states.



Observe that the state z makes one single a-transition going into $x + y$. This state is final, since $o^\sharp(x + y) = o^\sharp(x) + o^\sharp(y) = o(x) + o(y) = 1 + 0 = 1$. Moreover it makes an a-transition into $\delta^\sharp(x + y)(a) = \delta^\sharp(x)(a) + \delta^\sharp(y)(a) = \delta(x)(a) + \delta(y)(a) = y + z$.

The language accepted by the states of a NFA (S, o, δ) can be conveniently defined via the powerset construction: the language accepted by $x \in S$ is the language accepted by the singleton $\{x\}$ in the DFA $(\mathcal{P}(S), o^\sharp, \delta^\sharp)$, in symbols $[\![\{x\}]\!]$. Therefore, in the following, instead of considering the problem of language equivalence of states of the NFA, we will focus on language equivalence of *sets* of states of the NFA: given the sets of states X and Y in $\mathcal{P}(S)$, we say that X and Y are language equivalent ($X \sim Y$) iff $[\![X]\!] = [\![Y]\!]$. This is exactly what happens in classical automata theory where NFA are equipped with a set of initial states.

It is worth to note that, by definition, both $o^\sharp$ and $\delta^\sharp$ are semi-lattices homomorphisms. This property will be fundamental in Lemma 2 for proving the soundness of the up-to technique we are going to introduce. Moreover it induces *compositionality* of language equivalence, as stated by following theorem.

Theorem 2. *Let (S, o, δ) be a non-deterministic automaton and $(\mathcal{P}(S), o^\sharp, \delta^\sharp)$ be the corresponding deterministic automaton obtained through the powerset construction. The function $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ is a semi-lattice homomorphism, that is, for all $X_1, X_2 \in \mathcal{P}(S)$,*

$$[\![X_1 + X_2]\!] = [\![X_1]\!] + [\![X_2]\!] \quad \text{and} \quad [\![0]\!] = 0\,.$$

Proof. We prove that for all words $w \in A^*$, $[\![X_1 + X_2]\!](w) = [\![X_1]\!](w) + [\![X_2]\!](w)$, by induction on w.

- for ϵ, we have:

$$[\![X_1 + X_2]\!](\epsilon) = o^\sharp(X_1 + X_2) = o^\sharp(X_1) + o^\sharp(X_2) = [\![X_1]\!](\epsilon) + [\![X_2]\!](\epsilon)\,.$$

- for $a \cdot w$, we have:

$$\begin{aligned}
&[\![X_1 + X_2]\!](a \cdot w) \\
&= [\![\delta^\sharp(X_1 + X_2)(a)]\!](w) && \text{(by definition)} \\
&= [\![\delta^\sharp(X_1)(a) + \delta^\sharp(X_2)(a)]\!](w) && \text{(by definition)} \\
&= [\![\delta^\sharp(X_1)(a)]\!](w) + [\![\delta^\sharp(X_2)(a)]\!](w) && \text{(by induction hypothesis)} \\
&= [\![X_1]\!](a \cdot w) + [\![X_2]\!](a \cdot w)\,. && \text{(by definition)}
\end{aligned}$$

For the second part, we prove that for all words $w \in A^*$, $[\![0]\!](w) = 0$, again by induction on w. *Base case:* $[\![0]\!](\epsilon) = o^\sharp(0) = 0$. *Inductive case:* $[\![0]\!](a \cdot w) = [\![\delta^\sharp(0)(a)]\!](w) = [\![0]\!](w)$ that by induction hypothesis is 0. □

In order to check if the sets of states X and Y of an NFA (S, o, δ) are language equivalent, we can employ bisimulation on the DFA $(\mathcal{P}(S), o^\sharp, \delta^\sharp)$. In other words, a bisimulation for a NFA (S, o, δ) is a relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, such that whenever $X\ R\ Y$ then

1. $o^\sharp(X) = o^\sharp(Y)$ and
2. for all $a \in A$, $\delta^\sharp(X)(a)\ R\ \delta^\sharp(Y)(a)$.

Since this is just the old definition of bisimulation (Definition 1) applied on $(\mathcal{P}(S), o^\sharp, \delta^\sharp)$, it is immediate to see that $X \sim Y$ iff there exists a bisimulation that relates them.

Remark 3 (Linear time v.s. branching time)**.** *It is important not to confuse these bisimulation relations with the standard Milner-and-Park bisimulations [7] (which* strictly *imply language equivalence): in a standard bisimulation R if the following states x and y are in R,*

then each x_i should be in R with some y_j (and vice-versa). Here, instead, we first transform the transition relation into

$$x \xrightarrow{a} x_1 + \cdots + x_n \qquad\qquad y \xrightarrow{a} y_1 + \cdots + y_m\ ,$$

using the powerset construction, and then we require that $x_1 + \cdots + x_n$ and $y_1 + \cdots + y_m$ are related by R.

3.1 Bisimulation up to congruence

As explained in the introduction, we rely on the notion of bisimulation up to congruence. More precisely, this notion of congruence has to be understood w.r.t. set theoretic union of sets of states (+). We start with the following notion of congruence closure:

Definition 4 (Congruence closure)**.** *Let* $u\colon \mathcal{P}(\mathcal{P}(S)\times\mathcal{P}(S)) \to \mathcal{P}(\mathcal{P}(S)\times\mathcal{P}(S))$ *be the function on relations on sets of states defined as:*

$$R \mapsto \{(X_1 + X_2,\ Y_1 + Y_2) \mid X_1\ R\ Y_1 \wedge X_2\ R\ Y_2\}\,.$$

The function $c = (r \cup s \cup t \cup u \cup \mathrm{id})^\omega$ *is called the* congruence closure *function.*

Intuitively, $c(R)$ is the smallest equivalence relation which is a congruence with respect to the operation $+$ and which includes R. It could alternatively be defined inductively using the rules r, s, t, and id from the previous section, and the following one:

$$\frac{X_1\ c(R)\ Y_1 \qquad X_2\ c(R)\ Y_2}{X_1 + X_2\ c(R)\ Y_1 + Y_2}\ u$$

(Note that we do not include a rule for the constant 0 since it is subsumed by reflexivity (r).) Here is a concrete example; consider the following relation:

$$R = \{(x,\ y + z),\ (u,\ y + v)\}\,.$$

By summing these two pairs using rule u, we deduce $x + u\ c(R)\ y + z + v$, and since $u\ R\ y + v$, we also get $x + u\ c(R)\ z + u$. We can deduce many other equations; in fact, $c(R)$ defines the following partition of sets of states:

$$\{0\} \qquad \{y\} \qquad \{z\} \qquad \{v\} \qquad \{z + v\}$$

$$\{x,\ y + z,\ x + z,\ x + y,\ x + y + z\} \qquad \{u,\ y + v,\ u + v,\ y + u,\ y + u + v\}$$

$$\{x + u,\ z + u,\ y + z + v,\ \text{and the 14 remaining sets}\}\,.$$

Lemma 2. *The function* u *is compatible.*

Proof. Suppose that $R \rightarrowtail R'$, we have to show $u(R) \rightarrowtail u(R')$. Suppose that $X\ u(R)\ Y$, i.e., $X = X_1 + X_2$ and $Y = Y_1 + Y_2$ for some X_1, X_2, Y_1, Y_2 such that $X_1\ R\ Y_1$ and $X_2\ R\ Y_2$. By assumption, we have

$$\begin{aligned} o^\sharp(X_1) &= o^\sharp(Y_1) & o^\sharp(X_2) &= o^\sharp(Y_2) \\ t^\sharp(X_1)(a)\ R'\ t^\sharp(Y_1)(a) & & t^\sharp(X_2)(a)\ R'\ t^\sharp(Y_2)(a) & \qquad (\text{for all } a \in A) \end{aligned}$$

Since $o^\sharp$ and $t^\sharp$ are homomorphisms, we deduce

$$\begin{aligned} o^\sharp(X_1 + X_2) &= o^\sharp(Y_1 + Y_2) \\ t^\sharp(X_1 + X_2)(a)\ u(R')\ t^\sharp(Y_1 + Y_2)(a) & \qquad (\text{for all } a \in A) \end{aligned}$$

□

Theorem 3. *Any bisimulation up to* c *is contained in a bisimulation.*

Proof. By Theorem 1, it suffices to show that c is compatible, which follows from Lemmas 1 and 2, and Proposition 3. □

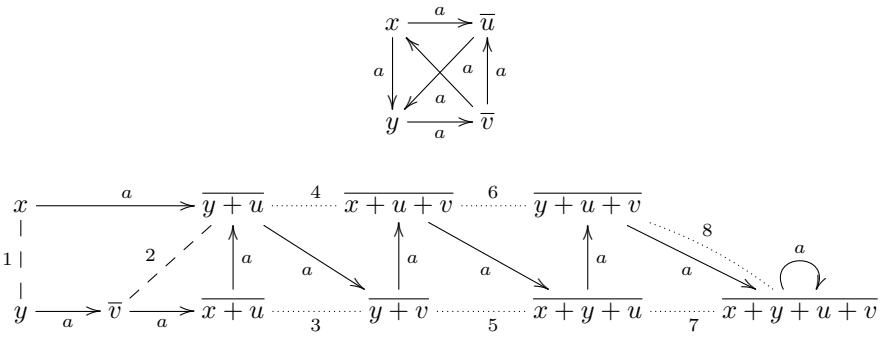


Figure 4: Bisimulations up to congruence, on a single letter NFA.

We already gave an example of bisimulation up to context in the introduction, which is a particular case of bisimulation up to congruence (up to context corresponds to using the function $(u \cup \mathrm{id})^\omega$, where we do not close the given relation under reflexivity, symmetry, and transitivity).

A more involved example illustrating the use of all ingredients of the congruence closure function (c) is given in Figure 4. The relation R expressed by the dashed numbered lines (formally $R = \{(y, x), (v, y + u)\}$) is neither a bisimulation, nor a bisimulation up to equivalence, since $v \to x + u$ and $y + u \to y + v$, but $(x + u, y + v) \notin e(R)$. However, R is a bisimulation up to congruence: we have $(x + u, y + v) \in c(R)$:

$$\begin{aligned} x + u \; c(R) \; & y + u && ((y, x) \in R) \\ c(R) \; & y + y + u && (+ \text{ is idempotent}) \\ c(R) \; & y + v && ((v, y + u) \in R) \end{aligned}$$

In contrast, we need eight pairs to get a bisimulation up to e containing the pair (y, x): this is the relation depicted with both dashed and dotted lines in Figure 4.

3.2 Optimised algorithm for NFA

As expected, this up-to congruence proof technique can be turned into an algorithm `HKC` for checking equivalence of sets of states in an NFA. The code is given in Figure 5. It basically corresponds to Hopcroft and Karp's algorithm (Figure 3), except that:

1. the states of the underlying DFA are computed on the fly, by the powerset construction;

2. we use the up-to congruence technique in step `3.2`.

HKC(X,Y)

```
(1)  R is empty; todo is empty;
(2)  insert (X, Y) in todo;
(3)  while todo is not empty
 {
      (3.1)  extract (X', Y') from todo;
      (3.2)  if (X', Y') ∈ c(R) then skip, else {
      (3.3)  if o♯(X') ≠ o♯(Y') then return false, else {
      (3.4)  for all a ∈ A, insert (δ♯(X')(a), δ♯(Y')(a)) in todo;
      (3.5)  insert (X', Y') in R; }}
 }
(4)  return true;
```

Figure 5: On the fly and up-to congruence variant of Hopcroft and Karp's algorithm, for checking the equivalence of sets of states X and Y of a NFA (S, o, δ).

Corollary 2. *For all sets of states X and Y, we have $X \sim Y$ iff* HKC(X,Y).

Proof. Same proof as for Proposition 2, by using the invariant $R \rightarrowtail c(R) \cup todo$ for the loop. We deduce that R is a bisimulation up to c after the loop. We conclude with Theorem 3. □

3.3 Computing the congruence closure

In the algorithm of Figure 5, we need to check whether some pairs belong to the congruence closure of the current relation R (step 3.2). We present here a simple solution, based on rewriting modulo associativity, commutativity, and idempotence (ACI).

The idea is to look each pair (X, Y) in the relation R as a pair of rewriting rules, which we will use to compute normal forms for sets of states:

$$X \to X + Y$$
$$Y \to X + Y \ .$$

Indeed, by idempotence, $X\ R\ Y$ entails $X\ c(R)\ Y\ c(R)\ X + Y$.

Definition 5. *Let $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ be a relation on sets of states. The* rewriting relation $\rightsquigarrow_R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ *is the smallest irreflexive relation defined by the following rules:*

$$\frac{X\ R\ Y}{X \rightsquigarrow_R X + Y} \qquad \frac{X\ R\ Y}{Y \rightsquigarrow_R X + Y} \qquad \frac{Z \rightsquigarrow_R Z'}{U + Z \rightsquigarrow_R U + Z'}$$

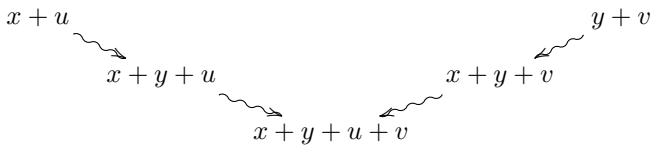
Lemma 3. *The rewriting relation $\rightsquigarrow_R$ is convergent and contained in $c(R)$.*

Proof. We have that $Z \rightsquigarrow_R Z'$ implies $|Z'| > |Z|$, where $|X|$ denotes the cardinality of the set X (note that $\rightsquigarrow_R$ is irreflexive). Since $|Z'|$ is bounded by $|S|$, the number of states of the NFA, the relation $\rightsquigarrow_R$ is strongly normalising. We can also check that whenever $Z \rightsquigarrow_R Z_1$ and $Z \rightsquigarrow_R Z_2$, either $Z_1 = Z_2$ or there is some Z' such that $Z_1 \rightsquigarrow_R Z'$ and $Z_2 \rightsquigarrow_R Z'$. Therefore, $\rightsquigarrow_R$ is convergent.

Finally, if $Z \rightsquigarrow_R Z'$ then there exists $(X, Y) \in (s \cup \mathrm{id})(R)$ such that $Z = Z + X$ and $Z' = Z + Y$. Therefore $Z\ c(R)\ Z'$ and, thus, $\rightsquigarrow_R$ is contained in $c(R)$. □

In the sequel, we denote by $X{\downarrow_R}$ the normal form of a set X w.r.t. $\rightsquigarrow_R$.

Intuitively, the normal form of a set is the largest set of its equivalence class. Recalling the example from Figure 4, the common normal form of $x + u$ and $y + v$ can be computed as follows (R is the relation $\{(y, x), (v, y + u)\}$):



Lemma 4. *Let $X, Y \in \mathcal{P}(S)$, we have $(X + Y){\downarrow_R} = (X{\downarrow_R} + Y{\downarrow_R}){\downarrow_R}$.*

Proof. Follows from confluence (Lemma 3) and from the fact that for all Z, Z', U, $Z \rightsquigarrow_R Z'$ entails $U + Z \rightsquigarrow_R U + Z'$. □

Theorem 4. *We have $(X, Y) \in c(R)$ iff $X{\downarrow_R} = Y{\downarrow_R}$.*

Proof. From left to right. We proceed by induction on the derivation of $(X, Y) \in c(R)$. The cases for rules r, s, and t are straightforward. For rule id, suppose that $X\ R\ Y$, we have to show $X{\downarrow_R} = Y{\downarrow_R}$:

- if $X = Y$, we are done;
- if $X \subsetneq Y$, then $X \rightsquigarrow_R X + Y = Y$;
- if $Y \subsetneq X$, then $Y \rightsquigarrow_R X + Y = X$;
- if neither $Y \subseteq X$ nor $X \subseteq Y$, then $X, Y \rightsquigarrow_R X + Y$.

(In the last three cases, we conclude by confluence—Lemma 3.)
For rule u, suppose by induction that $X_i{\downarrow_R} = Y_i{\downarrow_R}$ for $i \in 1, 2$; we have to show that $(X_1 + Y_1){\downarrow_R} = (X_2 + Y_2){\downarrow_R}$. This follows by Lemma 4.

From right to left. By Lemma 3, we have $X\ c(R)\ X{\downarrow_R}$ for any set X, so that $X\ c(R)\ X{\downarrow_R} = Y{\downarrow_R}\ c(R)\ Y$. □

In the corresponding normalisation algorithm, each pair of R may be used only once as a rewriting rule. However, we do not know in advance in which order to apply these rules. Therefore, checking whether $(X, Y) \in c(R)$ with this algorithm requires time proportional to $r^2 n$, where $r = |R|$ is the size of the

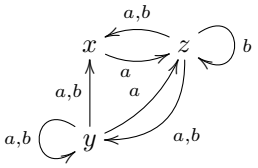
relation R, and $n = |S|$ is the number of states of the NFA (assuming linear time complexity for set-theoretic union and containment of sets of states).

There is room for optimisation; we could try for instance to normalise the set of rewriting rules when adding new rules (step `3.5`), or to optimise rewriting rules during normalisation (like it is done with the union-find data structure, where paths are compressed during the find operation). We could also look for better data structures to represent congruence classes. We leave this for future work: this is orthogonal to the ideas presented here.

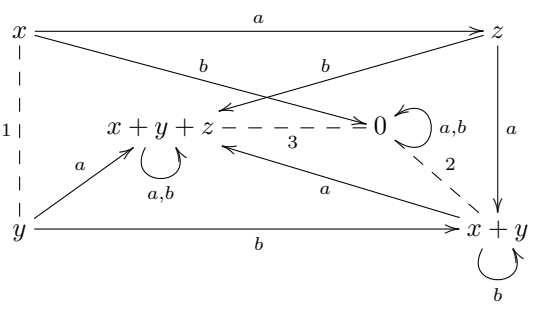
3.4 Heuristics matters

Like Hopcroft and Karp's algorithm for DFA (Figure 3), our algorithm for NFA produces a relation which is not a bisimulation, only a bisimulation up-to (here, up to congruence). The same argument as in Remark 2 can be made: while the produced relation depends on the order in which *todo* is processed, its congruence closure, which is a bisimulation, does not. However, unlike from Hopcroft and Karp's algorithm, the number of steps required to build this bisimulation up-to *does* depend on this choice.

Consider for instance the following NFA over the alphabet $A = \{a, b\}$:



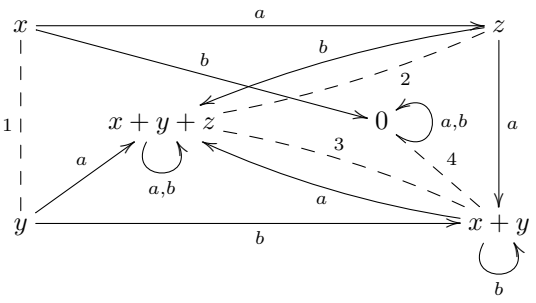
Starting from the singleton sets $\{x\}$ and $\{y\}$, the algorithm may compute a bisimulation up to congruence containing three or four pairs. If we start with the b-transitions, we reach the pair $(0, x + y)$, which imposes strong constraints (namely, both x and y are empty) so that most subsequent visited pairs are already in the congruence closure:



	pairs
ϵ	$x - y$
b	$0 - x + y$
a	$z \dots x + y + z$
bb	$0 \dots x + y$
ba	$0 - x + y + z$
bab	$0 \dots x + y + z$
baa	$0 \dots x + y + z$

(On the right-hand side, we list the visited pairs, marking them with a dash ($-$) when they are added to R, and with dots (...) when they already belong to $c(R)$.) On the contrary, if we delay the processing of $(0, x + y)$, the other pairs

we add to R are less constraining, so that we need one more step, as illustrated below:



	pairs
ϵ	$x - y$
a	$z - x + y + z$
aa	$x + y - x + y + z$
aaa	$x + y + z \dots x + y + z$
aab	$x + y \dots x + y + z$
ab	$x + y + z \dots x + y + z$
b	$0 - x + y$
ba	$0 \dots x + y + z$
bb	$0 \dots x + y$

While the order in which letters of the alphabet should be processed to be optimal seems hard to guess, empiric experiments tend to show that it is usually better to explore the underlying DFA in a breadth first manner rather than in depth first. Other heuristics are possible, like guessing which pairs will impose strong constraints in the congruence closure of the relation R, or which ones will result in small sub-DFA. We leave the study of such ideas for future work.

4 Complexity hints

We do not know how to properly analyse the complexity of our algorithm, for several reasons:

1. it depends on the heuristic we use to decide in which order to process the pairs in *todo*;
2. it depends on the algorithm we use to check whether a pair belongs to the congruence closure of a finite relation, and the algorithm we proposed in Section 3.3 to this end can certainly improved;
3. the worst case complexity might not be representative of the actual behaviour in practice (this holds also for Hopcroft and Karp's algorithm: NFA which produce very large DFA by the powerset construction are not so frequent);
4. an average-case analysis seems out of reach: we found no results about Hopcroft and Karp's algorithm in the literature, here we would moreover need to understand the average size of the minimal relation generating a given congruence (this is trivial for equivalences).

Therefore, we only provide partial answers here: we first give an example showing that `HKC` can be exponentially better than `HK`; we then show a bad case, where both algorithms behave the same and require exponential time; we finally give experimental data obtained on all small NFA and on larger random NFA.

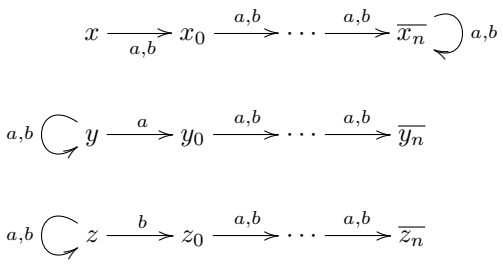


Figure 6: Cases where `HKC` is linear while `HK` requires exponential time.

In all cases, we focus on the size of the constructed bisimulation up-to: this does not depend on the implementation of the congruence closure, and this number is strongly related to the number of iterations (as mentioned before Proposition 2, the main loop is executed $1 + |A| \cdot |R|$ times, where $|A|$ and $|R|$ respectively denote the size of the alphabet and of the produced relation R).

4.1 Good cases

Consider the family of NFA given in Figure 6, where n is an arbitrary natural number. They intuitively correspond to the following regular expressions, and we have we have $x \sim y + z$.

$$x : (a+b)^{n+1} \cdot (a+b)^\star$$
$$y : (a+b)^\star \cdot a \cdot (a+b)^n$$
$$z : (a+b)^\star \cdot b \cdot (a+b)^n$$

The top automaton (x) is already deterministic, and the two other ones (y and z) yield exponential minimal DFA (in fact, starting from state y or z, the powerset construction results in a minimal DFA which has 2^{n+1} states; starting from state $y + z$, it yields a DFA with $2^{n+2} - 1$ states which is not minimal—x is the minimal DFA in this case).

While the standard algorithm (`HK`) requires $2^{n+2} - 1$ steps to prove that $x \sim y + z$ (all states of both DFA have to be mentioned in a bisimulation up to equivalence), our algorithm (`HKC`) requires only $2n + 3$ steps with a breadth first heuristic for picking pairs from *todo*. Indeed, it constructs the following relation:

$$\begin{aligned} R_n = \{&(x, y+z)\} \\ &\cup \{(x_i, y + y_0 + \cdots + y_i + z) \mid 0 \le i \le n\} \qquad (1) \\ &\cup \{(x_i, y + y_0 + \cdots + y_{i-1} + z_i + z) \mid 0 \le i \le n\}\ . \qquad (2) \end{aligned}$$

We have $|R_n| = 2n + 3$ and we can check that R_n is a bisimulation up to congruence:

Lemma 5. *For all natural numbers n, R_n is a bisimulation up to c for the NFA depicted in Figure 6.*

Proof. First notice that

$$y + y_0 + z \;\; c(R_n)\;\; y + z_0 + z\,, \tag{$\dagger$}$$

since these two sets are related to x_0 by R_n, and

$$x_i \;\; c(R_n)\;\; y + y_0 + \cdots + y_i + z_i + z\,, \tag{$\ddagger$}$$

by summing up pairs (1) and (2) and using idempotence. We then consider each kind of pair of R_n separately:

- $(x, y + z)$: we have $t^\sharp(x)(a) = x_0\ R_n\ y + y_0 + z = t^\sharp(y + z)(a)$, similarly, $t^\sharp(x)(b) = x_0\ R_n\ y + z_0 + z = t^\sharp(y + z)(b)$.
- $(x_i, y + y_0 + \cdots + y_i + z)$ for $i < n$: we have

$$\begin{aligned}
t^\sharp(x_i)(a) &= x_{i+1} \\
&\quad R_n\ y + y_0 + \cdots + y_i + y_{i+1} + z && \text{(by (1))} \\
&\quad = t^\sharp(y + y_0 + \cdots + y_i + z)(a)\,, \text{ and} \\
t^\sharp(x_i)(b) &= x_{i+1} \\
&\quad R_n\ y + y_0 + \cdots + y_i + y_{i+1} + z && \text{(by (1))} \\
&\quad c(R_n)\ y + y_1 + \cdots + y_i + y_{i+1} + z_0 + z && \text{(by ($\dagger$))} \\
&\quad = t^\sharp(y + y_0 + \cdots + y_i + z)(b)\,;
\end{aligned}$$

- $(x_i, y + y_0 + \cdots + y_{i-1} + z_i + z)$ for $i < n$: we have

$$\begin{aligned}
t^\sharp(x_i)(a) &= x_{i+1} \\
&\quad R_n\ y + y_0 + \cdots + y_i + z_{i+1} + z && \text{(by (1))} \\
&\quad = t^\sharp(y + y_0 + \cdots + y_{i-1} + z_i + z)(a)\,, \text{ and} \\
t^\sharp(x_i)(b) &= x_{i+1} \\
&\quad R_n\ y + y_0 + \cdots + y_i + z_{i+1} + z && \text{(by (2))} \\
&\quad c(R_n)\ y + y_1 + \cdots + y_i + z_{i+1} + z_0 + z && \text{(by ($\dagger$))} \\
&\quad = t^\sharp(y + y_0 + \cdots + y_{i-1} + z_i + z)(b)\,;
\end{aligned}$$

- $(x_n, y + y_0 + \cdots + y_n + z)$: we have

$$\begin{aligned}
t^\sharp(x_n)(a) &= x_n \\
&\quad R_n\ y + y_0 + \cdots + y_n + z && \text{(by (1))} \\
&\quad = t^\sharp(y + y_0 + \cdots + y_n + z)(a)\,, \text{ and} \\
t^\sharp(x_n)(b) &= x_n \\
&\quad R_n\ y + y_0 + \cdots + y_n + z && \text{(by (1))} \\
&\quad c(R_n)\ y + y_1 + \cdots + y_n + z_0 + z && \text{(by ($\dagger$))} \\
&\quad = t^\sharp(y + y_0 + \cdots + y_n + z)(b)\,;
\end{aligned}$$

- $(x_n, y + y_0 + \cdots + y_{n-1} + z_n + z)$: we have

$$\begin{aligned}
t^\sharp(x_n)(a) &= x_n \\
&\quad c(R_n)\ y + y_0 + \cdots + y_n + z_n + z && \text{(by } (\ddagger)\text{)} \\
&\quad = t^\sharp(y + y_0 + \cdots + y_{n-1} + z_n + z)(a)\,, \text{ and} \\
t^\sharp(x_n)(b) &= x_n \\
&\quad c(R_n)\ y + y_0 + \cdots + y_n + z_n + z && \text{(by } (\ddagger)\text{)} \\
&\quad c(R_n)\ y + y_1 + \cdots + y_n + z_n + z_0 + z && \text{(by } (\dagger)\text{)} \\
&\quad = t^\sharp(y + y_0 + \cdots + y_{n-1} + z_n + z)(b)\,. && \square
\end{aligned}$$

Considering heuristics, notice that the breadth-first strategy is crucial to get this behaviour: with a depth-first strategy, we would not add the pairs $(x_0, y + y_0 + z)$ and $(x_0, y + z_0 + z)$ from the beginning, and these pairs, which entail $(\dagger)$, are used to skip most pairs appearing during the unfolding of the DFA. Indeed, using to depth-first strategy, we get bisimulations up to congruence whose size is exponential—although they are smaller than the bisimulations up to equivalence obtained with `HK`.

4.2 Bad cases

Even though the previous example shows that our algorithm can be polynomial where the standard Hopcroft and Karp's algorithm is exponential, the problem of checking language equivalence of NFA is PSPACE-complete [6], and NP-complete when restricted to the one-letter case. We now give an example in the one-letter case where our algorithm requires exponential time (a nice property in the one-letter case is that *todo* contains at most one element, which means that we do not have to chose a heuristic for extracting pairs from that set).

The example is given in Figure 7; it consists of the DFA corresponding to the full language (state x) which we compare with the parallel composition of cycles of a length varying in $[1..n]$, for a given natural number n. In regular expressions syntax, it amounts to the following (trivial) equation:

$$a^\star = \sum_{i=1}^{n} (a^i)^\star \ .$$

For any natural number k, let S_k be the following set of states:

$$S_k = \sum_{i=1}^{n} x^i_{k \bmod i} \ .$$

For all k, we have $S_k \xrightarrow{a} S_{k+1}$, and this sequence is periodic, of period $p = \mathrm{lcm}[1..n]$, the least common multiplier of the first n positive numbers (known to be greater than 2^n for $n > 7$).

By running Hopcroft and Karp's algorithm between x and S_0, we get a bisimulation up to equivalence in p steps (this relation is actually a bisimulation,

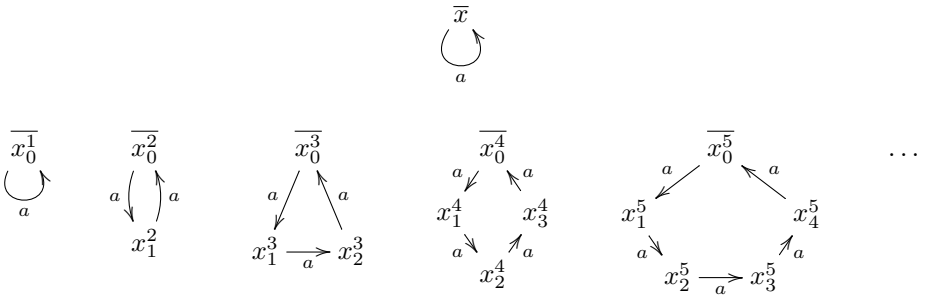


Figure 7: Bad case, where `HK` and `HKC` behave the same and require exponential time with one-letter automata

the up-to equivalence technique is not used). We can show that the optimised algorithm behaves exactly the same: the up-to congruence technique does not help. Intuitively, at the j-th iteration, we have

$$R = \{(x, S_k) \mid k < j\} \ ;$$

therefore if $j < p$ then (x, S_j) does not belong to R. This pair does not belong to $c(R)$ either: $x{\downarrow_R} = x + \sum_{k<j} S_k$ while S_j is in normal form and does not contain x.

Remark 4. *Note that if we merge states x and x_0^1 in this example, the standard algorithm still requires* $\mathrm{lcm}[1..n]$ *steps, but the optimised one only requires n steps: at step $n-1$, we have*

$$R = \{(x_0^0, S_k) \mid k < n\} \ ,$$

and now, the pair (x_0^0, S_n) belongs to $c(R)$: $x_0^0{\downarrow_R} \subseteq S_n{\downarrow_R}$, since x_0^0 belongs to S_n; and $x_0^0{\downarrow_R}$ is the full state, i.e., $\{x_i^j \mid i < j\}$, since we went through all states of each cycle. Therefore, $x_0^0{\downarrow_R} = S_n{\downarrow_R}$, and thus x_0^0 $c(R)$ S_n by Lemma 4.

4.3 Experimental data

We performed an exhaustive simulation for small NFA with one letter. The results are summarised in the table below: for each line, we ran the two algorithms on all NFA, storing the size of the largest bisimulation up-to obtained in this way (i.e., the worst case).

$\lvert S\rvert$	worst case	
	`HK`	`HKC`
3	5	3
4	12	5
5	17	7
6	26	9

These exhaustive computations are not tractable for larger sizes, and the apparent linear behaviour for `HKC` is a trap, as the example from Section 4.2 shows. To get an intuition of the behaviour of our algorithm on larger NFA, we performed a few tests on random automata:

$\|S\|$	`HK`		`HKC`	
	mean	median	mean	median
20	710.3	653.0	15.5	14.0
30	4884.1	4367.0	19.4	19.0

Here we worked with an alphabet of three letters, the probability to have a transition with a given label between two nodes is 10% in the first line, 5% in the second one (we chose these values to avoid getting NFA which mostly degenerate into trivial DFA). In both cases the mean and the median values were computed based on 1000 experiments, and we used a breadth-first heuristic in the implementation of `HKC`.

These preliminary experimental results look really promising; we would like to understand whether it is possible to assess them more formally.

Acknowledgements.

We are grateful to Marcello Bonsangue, Jan Rutten, and Alexandra Silva for the helpful discussions we had.

References

[1] M. Bonsangue A. Silva, F. Bonchi and J. Rutten. Generalizing the powerset construction, coalgebraically. In *Proc. FSTTCS*, volume 8 of *LIPIcs*, pages 272–283. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[2] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical Theory of Automata*, volume 12(6), pages 529–561. Polytechnic Press, NY, 1962.

[3] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1(2/3):251–273, 1992.

[4] J. E. Hopcroft. An n log n algorithm for minimizing in a finite automaton. In *Proc. International Symposium of Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[5] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell University, December 1971.

[6] A.R. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proc. STOC*, pages 1–9. ACM, 1973.

[7] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[8] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I/II. *Information and Computation*, 100(1):1–77, 1992.

[9] D. Pous. Complete lattices and up-to techniques. In *Proc. APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 351–366. Springer, 2007.

[10] J. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proc. CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 194–218. Springer, 1998.

[11] D. Sangiorgi. On the Bisimulation Proof Method. *Journal of Mathematical Structures in Computer Science*, 8:447–479, 1998.

[12] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[13] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.