Testing the Equivalence of Regular Languages

Marco Almeida Nelma Moreira Rogério Reis

Technical Report Series: DCC-2009-01
Version 1.0 June 2009

Departamento de Ciência de Computadores
&
Laboratório de Inteligência Artificial e Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL
Tel: 220 402 900 Fax: 220 402 950
http://www.dcc.fc.up.pt/Pubs/

Testing the Equivalence of Regular Languages

Marco Almeida* Nelma Moreira Rogério Reis
{mfa,nam,rvr}@ncc.up.pt

Abstract

The minimal deterministic finite automaton is generally used to determine regular languages equality. Antimirov and Mosses proposed a rewrite system for deciding regular expressions equivalence of which Almeida *et al.* presented an improved variant. Hopcroft and Karp proposed an almost linear algorithm for testing the equivalence of two deterministic finite automata that avoids minimisation.

In this paper we improve the best-case running time, present an extension of this algorithm to non-deterministic finite automaton, and establish a relationship between this algorithm and the one proposed in Almeida *et al.* We also present some experimental comparative results. All these algorithms are closely related with the recent coalgebraic approach to automata proposed by Rutten.

1 Introduction

The uniqueness of the minimal deterministic finite automaton for each regular language is in general used for determining regular languages equality. Whether the languages are represented by deterministic finite automata (DFA), non deterministic finite automata (NFA), or regular expressions (r.e.), the usual procedure uses the equivalent minimal DFA to decide equivalence. The best known algorithm, in terms of worst-case analysis, for DFA minimisation is loglinear [Hop71], and the equivalence problem is PSPACE-complete for both NFA and r.e. Based on the algebraic properties of regular expressions, Antimirov and Mosses proposed a terminating and complete rewrite system for deciding their equivalence [AM94]. In a paper about testing the equivalence of regular expressions, Almeida *et al.* [AMR08a] presented an improved variant of this rewrite system. As suggested by Antimirov and Mosses, and corroborated by further experimental results, a better average-case performance may be obtained.

Hopcroft and Karp [HK71] presented, in 1971, an almost linear algorithm for testing the equivalence of two DFAs that avoids their minimisation. Considering the merge of the two DFAs as a single one, the algorithm computes the finest right-invariant relation which identifies the initial states. The state equivalence relation that determines the minimal DFA is the coarsest relation in that condition.

We present some variants of Hopcroft and Karp's algorithm (**HK**) (Section 3), and establish a relationship with the one proposed in Almeida *et al.* (Section 4). In particular, we extend **HK** algorithm to NFAs and present some experimental comparative results (Section 5).

All these algorithms are also closely related with the recent coalgebraic approach to automata developed by Rutten [Rut03], where the notion of *bisimulation* corresponds to a

*Marco Almeida is funded by FCT grant SFRH/BD/27726/2006.

right-invariance. Two automata are bisimilar if there exists a bisimulation between them. For deterministic (finite) automata, the *coinduction proof principle* is effective for equivalence, *i.e.*, two automata are bisimilar if and only if they are equivalent. Both Hopcropt and Karp algorithm and Antimirov and Mosses method can be seen as instances of this more general approach (c.f. Corollary 1). This means that these methods may be easily extended to other Kleene Algebras, namely the ones that model program properties, and that have been successfully applied in formal program verification [Koz08].

2 Preliminaries

We recall here the basic definitions needed throughout the paper. For further details we refer the reader to the works of Hopcroft *et al.* [HMU00] and Kozen [Koz97].

An alphabet Σ is a nonempty set of symbols. A word over an alphabet Σ is a finite sequence of symbols of Σ. The empty word is denoted by ϵ and the length of a word w is denoted by $|w|$. The set $\Sigma^\star$ is the set of words over Σ. A language L is a subset of $\Sigma^\star$. If L_1 and L_2 are two languages, then $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$. The operator $\cdot$ is often omitted. A regular expression α over Σ represents a regular language $L(\alpha) \subseteq \Sigma^\star$ and is inductively defined by: $\emptyset$ is a r.e. and $L(\emptyset) = \emptyset$; ϵ is a r.e. and $L(\epsilon) = \{\epsilon\}$; $a \in \Sigma$ is a r.e. and $L(a) = \{a\}$; if α and β are regular expressions, $(\alpha + \beta)$, $(\alpha\beta)$ and $(\alpha)^\star$ are regular expressions, respectively with $L((\alpha + \beta)) = L(\alpha) \cup L(\beta)$, $L((\alpha\beta)) = L(\alpha)L(\beta)$ and $L((\alpha)^\star) = L(\alpha)^\star$. We adopt the usual convention that $\star$ has precedence over $\cdot$, which has higher precedence than $+$, and omit outer parentheses. The size of α is denoted by $|\alpha|$ and represents the number of symbols, operators, and parentheses in α. We denote by $|\alpha|_\Sigma$ the number of symbols in α. We define the *constant part* of α as $\varepsilon(\alpha) = \epsilon$ if $\epsilon \in L(\alpha)$, and $\varepsilon(\alpha) = \emptyset$ otherwise. Two regular expressions α and β are *equivalent*, and we write $\alpha \sim \beta$, if $L(\alpha) = L(\beta)$.

The algebraic structure $(RE, +, \cdot, \emptyset, \epsilon)$, where RE denotes the set of r.e. over Σ, constitutes an idempotent semiring, and, with the unary operator $\star$, a *Kleene algebra*. There are several well-known complete axiomatizations of Kleene algebras [Sal66, Koz94]. Let ACI denote the associativity, commutativity and idempotence of $+$.

A *nondeterministic finite automaton* (NFA) A is a tuple $(Q, \Sigma, \delta, I, F)$ where Q is finite set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ the transition relation, $I \subseteq Q$ the set of initial states, and $F \subseteq Q$ the set of final states. An NFA is *deterministic* (DFA) if for each pair $(q, a) \in Q \times \Sigma$ there exists at most one q' such that $(q, a, q') \in \delta$. The size of a NFA is $|Q|$. For $s \in Q$ and $a \in \Sigma$, we denote by $\delta(q, a) = \{p \mid (q, a, p) \in \delta\}$, and we can extend this notation to $x \in \Sigma^\star$, and to $R \subseteq Q$. For a DFA, we consider $\delta : Q \times \Sigma^\star \to Q$. The *language* accepted by A is $L(A) = \{x \in \Sigma^\star \mid \delta(I, x) \cap F \neq \emptyset\}$. Two NFAs A and B are *equivalent*, denoted by $A \sim B$ if they accept the same language. Given an NFA $A = (Q_N, \Sigma, \delta_N, I, F_N)$, we can use the *powerset construction* to obtain a DFA $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ equivalent to A, where $Q_D = 2^{Q_N}$, $q_0 = I$, for all $R \in Q_D$, $R \in F_D$ if and only $R \cap F_N \neq \emptyset$, and for all $a \in \Sigma$, $\delta_D(R, a) = \bigcup_{q \in R} \delta_N(q, a)$. This construction can be optimised by omitting states $R \in Q_D$ that are unreachable from the initial state.

Given a finite automaton $(Q, \Sigma, \delta, q_0, F)$, let $\varepsilon(q) = 1$ if $q \in F$ and $\varepsilon(q) = 0$ otherwise. We call a set of states $R \subseteq Q$ *homogeneous* if for every $p, q \in R$, $\varepsilon(p) = \varepsilon(q)$. A DFA is *minimal* if there is no equivalent DFA with fewer states. Two states $q_1, q_2 \in Q$ are said to be *equivalent*, denoted $q_1 \sim q_2$, if for every $w \in \Sigma^\star$, $\varepsilon(\delta(q_1, w)) = \varepsilon(\delta(q_2, w))$. Minimal DFAs are unique up to isomorphism. Given an DFA D, the equivalent minimal DFA $D/_\sim$

is called the *quotient automaton* of D by the equivalence relation $\sim$. The state equivalence relation $\sim$, is a special case of a right-invariant equivalence relation w.r.t. D, *i.e.*, a relation $\equiv \subseteq Q \times Q$ such that all classes of $\equiv$ are homogeneous, and for any $p, q \in Q$, $a \in \Sigma$ if $p \equiv q$, then $\delta(p,a)/_{\equiv} = \delta(q,a)/_{\equiv}$, where for any set S, $S/_{\equiv} = \{[s] \mid s \in S\}$. Finally, we recall that every equivalence relation $\equiv$ over a set S is efficiently represented by the partition of S given by $S/_{\equiv}$. Given two equivalence relations over a set S, $\equiv_R$ and $\equiv_T$, we say that $\equiv_R$ is *finer* then $\equiv_T$ (and $\equiv_T$ *coarser* then $\equiv_R$) if and only if $\equiv_R \subseteq \equiv_T$.

3 Testing finite automata equivalence

The classical approach to the comparison of DFAs relies on the construction of the minimal equivalent DFA. The best known algorithm for this procedure runs in $O(kn \log n)$ time [Hop71], for a DFA with n states over an alphabet of k symbols.

Hopcroft and Karp [HK71] proposed an algorithm for testing the equivalence of two DFAs that makes use of an almost $O(n)$ set merging method. This set merging algorithm assumes disjoint sets and is based on three functions: MAKE, FIND, and UNION.

Later, however, both the original authors and Tarjan [HU73, Tar75] showed that running-time of this algorithm is actually $O(m \log^\star n)$ for $m \geq n$ FIND operations intermixed with $n-1$ UNION[1] operations, where

$$\log^\star n = \min\{i \mid \underbrace{\log \log \cdots \log}_{\text{i times}}(n) \leq 1\}.$$

As we assume disjoint sets, it is possible to use both the *union by rank* and the *path compression* heuristics [CLRS03], thus achieving a running time complexity $O(m\alpha(n))$ for any sequence of m MAKE, UNION, or FIND operations of which n are MAKE operations. As $\alpha(n)$ relates to a functional inverse of the Ackermann function, it grows *very slowly* and we can consider it a constant.

3.1 The original Hopcroft and Karp algorithm

Let $A = (Q_1, \Sigma, p_0, \delta_1, F_1)$ and $B = (Q_2, \Sigma, q_0, \delta_2, F_2)$ be two DFAs, with $|Q_1| = n$, $|Q_2| = m$, and such that Q_1 and Q_2 are disjoint, *i.e.*, $Q_1 \cap Q_2 = \emptyset$. In order to simplify notation, we assume $Q = Q_1 \cup Q_2$, $F = F_1 \cup F_2$, and $\delta(p,a) = \delta_i(p,a)$ for $p \in Q_i$.

We begin by presenting the original algorithm by Hopcroft and Karp [AHU74] for testing the equivalence of two DFAs as Algorithm 1. It does not involve any minimisation process and can be made almost linear in the worst-case.

If A and B are equivalent DFAs, the algorithm computes the finest right-invariant equivalence relation over Q that identifies the initial states, p_0 and q_0. The associated set partition is built using the UNION-FIND method. This algorithm assumes disjoint sets and defines the three functions which follow.

- MAKE(i): creates a new set (singleton) for one element i (the identifier);
- FIND(i): returns the identifier S_i of the set which contains i;
- UNION(i, j, k): combines the sets identified by i and j in a new set $S_k = S_i \cup S_j$; S_i and S_j are destroyed.

[1] Referred to as MERGE by Hopcroft and Ullman.

A very important subtlety of the UNION operation is that the two combined sets are destroyed in the end.

This means that a function call such as UNION(p, q, q') makes $S_{q'} = S_p \cup S_q$, destroying S_p and S_q, which assures that $|S_{q'}| = |S_p| + |S_q|$ and that, in the lines 12–13 of the Algorithm 1

$$|\bigcup_i S_i| = |Q|.$$

It is clear that, disregarding the set operations, the worst-case time of the algorithm is $O(k(n+m))$, where $k = |\Sigma|$. Line 2 is executed exactly $n+m$ times. Because of the previously pointed out behaviour of the UNION procedure, lines 12–13 are executed a number of times which is bounded by $n + m$. The number of times that the *while* loop in the lines 5–11 is executed is limited by the total number of elements pushed to the stack S. Each time a pair of states is pushed onto the stack, two sets are merged (lines 9–11), and thus the total number of sets is decreased by one. As initially there are only $n+m$ sets (and again, because of the previously pointed out behaviour of the UNION procedure), at most $n + m - 1$ pairs are placed in the stack during the execution of the loop. Because this is executed for each symbol in the alphabet, the total execution time of the algorithm — not considering the set operations — is $O(k(n + m))$.

An arbitrary sequence of i MAKE, UNION, and FIND operations, j of which are MAKE operations in order to create the required sets, can be performed in worst-case time $O(i\alpha(j))$, where $\alpha(j)$ is related to a functional inverse of the Ackermann function, and, as such, grows *very* slowly. In fact, for every *practical* values of j (up to $2^{2^{2^{16}}}$), $\alpha(j) \leq 4$.

```
1  def HK(A, B):
2      for q ∈ Q: MAKE(q)
3      S = ∅
4      UNION(p0, q0, q0); PUSH(S, (p0, q0))
5      while (p, q) = POP(S):
6          for a ∈ Σ:
7              p' = FIND(δ(p, a))
8              q' = FIND(δ(q, a))
9              if p' ≠ q':
10                 UNION(p', q', q')
11                 PUSH(S, (p', q'))
12     if ∀Si∀p, q ∈ Si  ε(p) = ε(q): return True
13     else: return False
```

Algorithm 1: The original **HK** algorithm.

When applied to Algorithm 1, this set union algorithm allows for a worst-case time complexity of $O(k(n + m) + 3i\alpha(j)) = O(k(n + m) + 3(n + m)\alpha(n + m))$. Considering $\alpha(n + m)$ constant, the asymptotic running-time of the algorithm is $O(k(n + m))$. The correctness of this algorithm is proved in Section 4, Theorem 2.

3.2 Improved best-case running time

By altering the FIND function in order to create the set being looked for if it does not exist, *i.e.*, whenever FIND(i) fails, MAKE(i) is called and the set $S_i = \{i\}$ is created, we may add a *refutation* procedure earlier in the algorithm. This allows the algorithm to return as soon as it finds a pair of states such that one is final and the other is not, as there exists a word

recognized by one of the automata but not by the other, and thus, they are not equivalent. This alteration to the FIND procedure avoids the initialization of $m + n$ sets which may never actually be used. These modifications to Algorithm 1 are presented in Algorithm 2.

Although it does not change the worst-case complexity, the best-case analysis is considerably better, as it goes from $\Omega(k(n+m))$ to $\Omega(1)$. Not only it is possible to distinguish the automata by the first pair of states, but it is also possible to avoid the linear check in the lines 12–13.

The observed asymptotic behaviour of minimality of initially connected DFAs (ICDFAs) [AMR07], suggests that, when dealing with random DFAs, the probability of having two equivalent automata is very low, and a refutation method will be very useful (see Section 5).

We present a proof that the refutation method preserves the correctness of the algorithm in Lemma 1.

We also show in Section 3.3 that minor changes to this version of the algorithm allow it to be used with NFAs.

Lemma 1. *In line 5 of Algorithm 1, all the sets S_i are homogeneous if and only if all the pairs of states (p, q) pushed into the stack are such that $\varepsilon(p) = \varepsilon(q)$.*

Proof. Let us proceed by induction on the number l of times line 5 is executed. If $l = 1$, it is trivial. Suppose that lemma is true for the l^{th} time the algorithm executes line 5. If for all $a \in \Sigma$, the condition in line 9 is false, for the $(l+1)^{th}$ time the homogeneous character of the sets remains unaltered. Otherwise, it is clear that in lines 10–11, $S_{p'} \cup S_{q'}$ is homogeneous if and only if $\varepsilon(p') = \varepsilon(q')$. Thus the lemma is true. □

```
1   def HKi(A, B):
2       MAKE(p0); MAKE(q0)
3       S = ∅
4       UNION(p0, q0, q0); PUSH(S, (p0, q0))
5       while (p, q) = POP(S):
6           if ε(p) ≠ ε(q): return False
7           for a ∈ Σ:
8               p' = FIND(δ(p, a))
9               q' = FIND(δ(q, a))
10              if p' ≠ q':
11                  UNION(p', q', q')
12                  PUSH(S, (p', q'))
13      return True
```

Algorithm 2: **HK** algorithm with an early refutation step (**HKi**).

Theorem 1. *Algorithms 1 (**HK**) and 2 (**HKi**) are equivalent.*

Proof. By Lemma 1, if there is a pair of states (p, q) pushed into the stack such that $\varepsilon(p) \neq \varepsilon(q)$, then the algorithm can terminate and return *False*. That is exactly what Algorithm 2 does. □

3.3 Testing NFA equivalence

It is possible to extend Algorithm 2 to test the equivalence of NFAs. The basic idea is to embed the powerset construction into the algorithm, although this must be done with some caution. We call this algorithm **HKe**. As any DFA is a particular case of an NFA,

all the experimental results presented on Section 5 use Algorithm **HKe**, whether the finite automata being tested are deterministic or not.

Let $N_1 = (Q_1, \Sigma, \delta_1, I_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, I_2, F_2)$ be two NFAs. We assume that Q_1 and Q_2 disjoint, and, we make $Q_N = Q_1 \cup Q_2$, $F_N = F_1 \cup F_2$, and $\delta_N(p, a) = \delta_i(p, a)$ for $p \in Q_i$.

The function ε must be extended to sets of states in the following way:

$$\varepsilon(p) = 1 \Leftrightarrow \exists p' \in p : \varepsilon(p') = 1$$

where $p \subseteq Q$, and we need a new transition function

$$\Delta : 2^Q \times \Sigma \to 2^Q$$
$$\Delta(p, a) = \bigcup_{p' \in p} \delta(p', a).$$

Notice that when dealing with NFAs it is essential to use the idea described in Subsection 3.2 and adjust the FIND operation so that FIND(i) will create the set S_i if it does not yet exist. This way we avoid calling MAKE for each of the $2^{|Q|}$ sets, which would lead directly to the worst-case of the powerset construction. This extended version is presented in Algorithm 3.

```
1  MAKE(I1)
2  MAKE(I2)
3  S = ∅
4  UNION(I1, I2, I2)
5  PUSH(S, (I1, I2))
6  while (p, q) = POP(S):
7      if ε(p) = ε(q):
8          return False
9  for a ∈ Σ:
10     p' = FIND(Δ(p, a))
11     q' = FIND(Δ(q, a))
12     if p' ≠ q':
13         UNION(p', q', q')
14         PUSH(S, (p', q'))
15 return True
```

Algorithm 3: Hopcroft and Karp's algorithm extended to NFAs (**HKe**).

Lemma 2. *Algorithm 3 is an extension of Algorithm 2 which may be applied to NFAs as it embeds the powerset construction method.*

Proof. As the correction of the algorithm for testing the equivalence of DFAs is already done by Hopcroft *et. al* [AHU74], it suffices to show that the elements p and q (popped from the stack S) are the subsets of 2^Q which correspond to a single state in the associated DFA, just like in the powerset construction method. The proof follows by induction on the number of operations on the stack S.

Base: The sets I_1 and I_2 are pushed onto the stack. These correspond to the initial state of the DFAs equivalent to N_1 and N_2, respectively.

Induction: By induction hypothesis, we have that at the n^{th} call to POP(S) each of p and q are subsets of Q which correspond to a single state in the deterministic automaton equivalent to N_1 or N_2 (denoted by D_1 and D_2, respectively). Without loss of generality,

let us consider only p. Notice that, by definition, Δ corresponds to the transition function for the deterministic automaton in the powerset construction method. Thus the call $\Delta(p, a)$ returns the subset of 2^Q reachable from p by consuming the symbol a. This corresponds to the next "deterministic" state of either D_1 or D_2, and so we are embedding the powerset construction method in Algorithm 2. □

4 Relationship with Antimirov and Mosses' method

We present in this Section an algorithm to test the equivalence of regular expressions without converting them to the equivalent minimal automata and establish a relationship with the algorithms presented in the previous section.

4.1 Antimirov and Mosses' algorithm

The *derivative* [Brz64] of a r.e. α with respect to a *symbol* $a \in \Sigma$, denoted $a^{-1}(\alpha)$, is defined recursively on the structure of α as follows:

$$a^{-1}(\emptyset) = \emptyset; \qquad a^{-1}(\alpha + \beta) = a^{-1}(\alpha) + a^{-1}(\beta);$$
$$a^{-1}(\epsilon) = \emptyset; \qquad a^{-1}(\alpha\beta) = a^{-1}(\alpha)\beta + \varepsilon(\alpha)a^{-1}(\beta);$$
$$a^{-1}(b) = \begin{cases} \epsilon, & \text{if } b = a; \\ \emptyset, & \text{otherwise;} \end{cases} \qquad a^{-1}(\alpha^\star) = a^{-1}(\alpha)\alpha^\star.$$

This notion can be trivially extended to words, and considering r.e. modulo the ACI axioms, Brzozowski [Brz64] proved that, the set of derivatives of a r.e. α, $\mathcal{D}(\alpha)$, is finite. This result leads to the definition of *Brzozowski's automaton* which is equivalent to a given r.e. α: $D_\alpha = (\mathcal{D}(\alpha), \Sigma, \delta_\alpha, \alpha, F_\alpha)$ where $F_\alpha = \{d \in \mathcal{D}(\alpha) \mid \varepsilon(d) = \epsilon\}$, and $\delta_\alpha(d, a) = a^{-1}(d)$, for all $d \in \mathcal{D}(\alpha)$, $a \in \Sigma$.

Antimirov and Mosses [AM94] proposed a rewrite system for deciding the equivalence of two extended r.e. (with intersection), based on a complete axiomatization. This is a refutation method such that testing the equivalence of two r.e. corresponds to an iterated process of testing the equivalence of their derivatives. In the process, a Brzozowski's automaton is computed for each r.e. Not considering extended r.e., Algorithm 4 is a version of **AM**'s method, which was, essentially, the one proposed by Almeida *et al.* [AMR08a]. Further details about the notation, implementation, and comparison with the original rewrite system may be found in the cited article.

```
1  def AM(α, β):
2      S = {(α, β)}
3      H = ∅
4      while (α, β) = POP(S):
5          if ε(α) ≠ ε(β): return False
6          PUSH(H, (α, β))
7          for a ∈ Σ:
8              α' = a^{-1}(α)
9              β' = a^{-1}(β)
10             if (α', β') ∉ H: PUSH(S, (α', β'))
11     return True
```

Algorithm 4: A simplified version of algorithm **AM**.

4.2 A naïve HK algorithm

We now present a naïve version of the Algorithm 1. It will be useful to prove its correctness and to establish a relationship to the Antimirov and Mosses' method (**AM**). Let $A = (Q_1, \Sigma, p_0, \delta_1, F_1)$ and $B = (Q_2, \Sigma, q_0, \delta_2, F_2)$ be two DFAs, with $|Q_1| = n$ and $|Q_2| = m$, and Q_1 and Q_2 disjoint. Consider Algorithm 5. To prove the correctness we show that in H we collect the pairs of states of the relation R, defined below.

```
1  def HKn(A,B):
2      S = {(p0, q0)}
3      H = ∅
4      while (p, q) = POP(S):
5          PUSH(H, (p, q))
6          for a ∈ Σ:
7              p' = δ1(p, a)
8              q' = δ2(q, a)
9              if (p', q') ∉ H: PUSH(S, (p', q'))
10     for (p, q) in H:
11         if ε(p) ≠ ε(q): return False
12     return True
```

Algorithm 5: The algorithm **HKn**, a naïve version of **HK**.

Lemma 3. *In line* 5, $(p, q) \notin H$ *and no pair of states is ever removed from* H*.*

Proof. It is obvious that no pair of states is ever removed from H, as only PUSH operations are performed on H throughout the algorithm.

It is also easy to see that on line 5 $(p, q) \notin H$, as it suffices to notice that S and H are disjoint. The elements pushed into H on line 5 are popped from S immediately before. Only on line 9 are any elements, say (p', q'), pushed into S, and this only happens if $(p', q') \notin H$. □

Lemma 4. *The Algorithm 5 is terminating with time complexity* $O(knm)$*.*

Proof. The elements of S are are pairs of states (p, q), such that $p \in Q_1$ and $q \in Q_2$. This results in, at most, nm elements being pushed into S. The only PUSH operation on H — line 5 — is performed with elements popped from S and thus, H will also have at most nm elements. This assures termination.

For each element in S, lines 6–9 are executed once for each element of Σ. As the loop in lines 10–11 is executed at most nm times, this results in a running time complexity of $O(knm)$. □

Lemma 5. *In Algorithm 5, for all* $(p, q) \in Q_1 \times Q_2$*,* $(p, q) \in S$ *in a step* $k > 0$ *if and only if* $(p, q) \in H$ *for some step* $k' > k$*.*

Proof. We start by recalling, as shown on Lemma 3, that S and H are disjoint. It is obvious that if $(p, q) \in S$ in a step k of Algorithm 5, then $(p, q) \in H$ for any $k' > k$. Simply observe that elements are only pushed into H after being popped from S — lines 4–5. For the same reason, if some element $(p, q) \in H$ at step k', it had to be in S at some step $k < k'$. □

Definition 1. *Let* R *be defined as follows:*

$$R = \{(p, q) \in Q_1 \times Q_2 \mid \exists x \in \Sigma^\star : \delta_1(p_0, x) = p \wedge \delta_2(q_0, x) = q\}.$$

Lemma 6. *For all* $(p,q) \in Q_1 \times Q_2$, $(p,q) \in S$ *at some step of Algorithm 5, if and only if* $(p,q) \in R$.

Proof. Let $(p,q) \in R$, *i.e.*, $\exists w : \delta_1(p_0,w) = p \wedge \delta_2(q_0,w) = q$. The proof will follow by induction on the size of the word w.

Base: $\delta_1(p_0,\epsilon) = p_0$, $\delta_2(q_0,\epsilon) = q_0$, and $(p_0,q_0) \in S$ already on line 2.

Induction: Let $w = ua$ such that $\delta_1(p_0,u) = p$ and $\delta_2(q_0,u) = q$. By induction hypothesis, we know that $(p,q) \in S$. On lines 7–9, $p' = \delta_1(p,a)$ and $q' = \delta_2(q_0,a)$ will be calculated and added to S unless $(p',q') \in H$. In this case, however, by Lemma 5 $(p',q') \in S$ at some previous step of the algorithm.

Conversely, and because new elements are only added to S on line 9, $(p,q) \in S$ only if some word w is such that $\delta_1(p_0,w) = p \wedge \delta_2(q_0,w) = q$. □

Lemma 7. *In line* 10, *for all* $(p,q) \in Q_1 \times Q_2$, $(p,q) \in R$ *if and only if* $(p,q) \in H$.

Proof. Suppose that $(p,q) \in R$. Then there exists a $w \in \Sigma^\star$, such that $\delta_1(p_0,w) = p$ and $\delta_2(q_0,w) = q$. The proof proceeds by induction on the length of the word w. If $|w| = 0$, then $w = \epsilon$, $\delta_1(p_0,\epsilon) = p_0$, $\delta_2(q_0,\epsilon) = q_0$, and $(p_0,q_0) \in H$. Let $w = ya$ with $a \in \Sigma$ and $y \in \Sigma^\star$. Then $\delta_1(p_0,w) = \delta_1(\delta_1(p_0,y),a) = \delta_1(p',a)$ and $\delta_1(q_0,w) = \delta_1(\delta_1(q_0,y),a) = \delta_1(q',a)$, for some $p' \in Q_1$ and $q' \in Q_2$. By the induction hypothesis, $(p',q') \in H$ and, by Lemma 5 there exists a step k such that $(p',q') \in S$. Thus $(p,q) \in H$ from a step $k' > k$ on. Reciprocally, if $(p,q) \in H$, by Lemma 5 and Lemma 6 $(p,q) \in R$. □

Considering Lemma 6 and Lemma 7, the following theorem ensures the correctness of Algorithm 5.

Theorem 2. $A \sim B$ *if and only if for all* $(p,q) \in R$, $\varepsilon(p) = \varepsilon(q)$.

Proof. Suppose, by absurd, that A and B are not equivalent and that the condition holds. Then, there exists $w \in \Sigma^\star$ such that $\varepsilon(\delta(p_0,w)) \neq \varepsilon(\delta(q_0,w))$. But in that case there is a contradiction because $(\delta(p_0,w),\delta(q_0,w)) \in R$. On the other hand, if there exists a $(p,q) \in R$ such that $\varepsilon(p) \neq \varepsilon(q)$, obviously A and B are not equivalent. □

The relation R can be seen as a relation on $(Q_1 \cup Q_2)^2$ which is reflexive and symmetric. Its transitive closure $R^\star$ is an equivalence relation.

Lemma 8. $\forall (p,q) \in R$, $\varepsilon(p) = \varepsilon(q)$ *if and only if* $\forall (p,q) \in R^\star$, $\varepsilon(p) = \varepsilon(q)$.

Proof. Let $(p,q),(q,r) \in R$. Since $R^\star$ is the transitive closure of R, $(p,r) \in R^\star$ and if $\varepsilon(p) = \varepsilon(q)$, then $\varepsilon(p) = \varepsilon(r)$. On the other hand, as $R \subseteq R^\star$, if $\varepsilon(p) = \varepsilon(q)$ $\forall (p,q) \in R^\star$, the same will be true for every $(p,q) \in R$. □

Corollary 1. $A \sim B$ *if and only if* $\forall (p,q) \in R^\star$, $\varepsilon(p) = \varepsilon(q)$.

The Algorithm **HK** computes $R^\star$ by starting with the finest partition in $Q_1 \cup Q_2$ (the identity). And if $A \sim B$, $R^\star$ is a right-invariance.

Corollary 2. *Algorithm 5 and Algorithm 1 are equivalent.*

4.3 Equivalence of the two methods

The Algorithm 5 can be modified to a *earlier refutation* version, as in Algorithm 2. In order to do so, we remove lines 10–11, and we insert a line equal to line 7 of Algorithm 2, before line 4. It is then obvious that Algorithm 4 corresponds to Algorithm 5 applied to Brzozowski's automata of two r.e., where these DFAs are incrementally constructed during the algorithm's execution. In particular, the halting conditions are the same considering the definition of final states in a Brzozowski's automaton.

It is possible to use Algorithm 4 to get a DFA from each of the regular expressions α and β in the following way. Let $D_\alpha = (Q_\alpha, \Sigma, \delta_\alpha, q_\alpha, F_\alpha)$ and $D_\beta = (Q_\beta, \Sigma, \delta_\beta, q_\beta, F_\beta)$ the equivalent DFAs to α e β, respectively. They are constructed in the following way:

- initialize $Q_\alpha = \{\alpha\}$, $Q_\beta = \{\beta\}$;
- $q_\alpha = \alpha$, $q_\beta = \beta$;
- for each instruction $\alpha' = a^{-1}(\alpha)$, add the transition $\delta_\alpha(\alpha, a) = \alpha'$ and make $Q_\alpha = Q_\alpha \cup \{\alpha'\}$ (same for β and β');
- whenever $\varepsilon(\alpha) = 1$, make $F_\alpha = F_\alpha \cup \{\alpha\}$ (same for β).

To prove the equivalence of the Algorithms 1 and 4, we will first apply Theorem 1 to Algorithm 5 in order to transform it in a refutation procedure. This modified version is presented as Algorithm 6. We will then show that Algorithm 4 actually embeds Hopcroft and Karp's method while constructing the equivalent DFAs.

```
1  S = {(p0, q0)}
2  H = ∅
3  while (p, q) = POP(S):
4      if ε(p) ≠ ε(q): return False
5      PUSH(H, (p, q))
6      for a ∈ Σ:
7          p' = δ1(p, a)
8          q' = δ2(q, a)
9          if (p', q') ∉ H:
10             PUSH(S, (p', q'))
11 return True
```

Algorithm 6: A naive version of Hopcroft and Karp's algorithm with refutation.

To verify the equivalence of Algorithms 6 and 4, the following observations should be enough.

The instructions

$$\alpha' = a^{-1}(\alpha); \quad \beta' = a^{-1}(\beta)$$

from Algorithm 4 are trivially equivalent to

$$p' = \delta_1(p, a); \quad q' = \delta_2(q, a)$$

in Algorithm 6, by the very definition of the method which constructs the equivalent DFAs.

The halting conditions are also the same. As $p \in F_\alpha$ if and only if $\varepsilon(\alpha) = 1$, we know that $\varepsilon(\alpha') \neq \varepsilon(\beta')$ if and only if $\varepsilon(p') \neq \varepsilon(q')$ when we consider the DFAs associated to each of the regular expressions, such that $p' \in Q_\alpha$, $q' \in Q_\beta$.

Theorem 3. *Algorithm 4 (**AM**) corresponds to Algorithm 5 (**HKn**) applied to Brzozowski's automata of two r.e.*

4.4 Improving Algorithm AM with Union-Find

Considering the Theorem 3 and the Corollary 2, we can improve the Algorithm 4 (**AM**) for testing the equivalence of two r.e. α and β, by considering Algorithm 1 applied to the Brzozowski's automata correspondent to the two r.e. Instead of using a stack (H) in order to keep an history of the pairs of regular expressions which have already been tested, we can build the correspondent equivalence relation $R^\star$ (as defined for Lemma 8). Two main changes must be considered:

- One must ensure that the sets of derivatives of each regular expression are disjoint. For that we consider their disjoint sum, where derivatives w.r.t. a word u are represented by tuples $(u^{-1}(\alpha), 1)$ and $(u^{-1}(\beta), 2)$, respectively.
- In the UNION-FIND method, the FIND operation needs an equality test on the elements of the set. Testing the equality of two r.e.— even syntactic equality — is already a computationally expensive operation, and tuple comparison will be even slower. On the other hand, integer comparison, can be considered to be $O(1)$. As we know that each element of the set is unique, we may consider some hash function which assures that the probability of collision for these elements is extremely low. This allows us to safely use the hash values as the elements of the set, and thus, arguments to the FIND operation, instead of the r.e. themselves. This is also a natural procedure in the implementations of conversions from r.e. to automata.

We call **equivUF** to the resulting algorithm. The experimental results are presented on Table 3, Section 5.

4.5 Worst-case complexity analysis

In Almeida *et al.* [AMR08a] the algorithm **AM** was improved by considering partial derivatives [Ant96]. The resulting algorithm (**equivP**) can be seen as the algorithm **HKe** applied to the partial derivatives NFA of a r.e. We present a lower bound for the worst-case complexity of this algorithm by exhibiting a family of r.e. for which the comparison method can be exponential on the number of alphabetical symbols $|\alpha|_\Sigma$ of a r.e. α. We will proceed by showing that the partial derivatives NFA N of a r.e. α is such that $|N| \in O(|\alpha|_\Sigma)$ and the number of states of the smallest equivalent DFA is exponential on $|N|$.

Figure 1 presents a classical example of a bad behaved case of the powerset construction, by Hopcroft *et al.* [HMU00]. Although this example does not reach the 2^n states bound, the smallest equivalent DFA has exactly 2^{n-1} states.

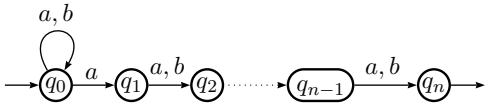

Figure 1: NFA which has no equivalent DFA with less than 2^n states.

Consider the r.e. family $\alpha_\ell = (a+b)^\star a(a+b)^\ell$, where $|\alpha_\ell|_\Sigma = 3 + 2\ell = m$. It is easy to see that the NFA in Figure 1 is obtained directly from the application of the **AM** method to α_ℓ, with the corresponding partial derivatives presented on Figure 2. The set of the partial derivatives $PD(\alpha_\ell) = \{\alpha_\ell, (a+b)^\ell, \ldots, (a+b), \epsilon\}$ has $\ell + 2 = \frac{m+1}{2}$ elements, which corresponds to the size of the obtained NFA. The equivalent minimal DFA has $2^{\ell+1} = 2^{\frac{m-1}{2}}$ states.

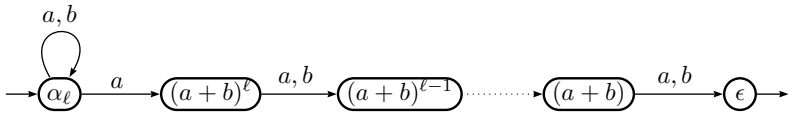

Figure 2: NFA obtained from the r.e. α using the **AM** method.

5 Experimental results

In this section we present some experimental results of the previously discussed algorithms applied to DFAs, NFAs, and r.e. We also include the same results of the tests using Hopcroft's (**Hop**) and Brzozowski's (**Brz**) [Brz63] automata minimization algorithms. The random DFAs were generated using publicly available tools[2][AMR07]. The NFAs dataset was obtained with a set of tools described by Almeida *et al.* [AMR08b].

All the algorithms were implemented in the `Python` programming language. The tests were executed in the same computer, an Intel® Xeon® 5140 at 2.33GHz with 4GB of RAM.

	$n=5$						$n=50$					
	$k=2$			$k=50$			$k=2$			$k=50$		
Alg.	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**
	Eff.	Total	Avg.	Eff.	Total	Avg.	Eff.	Total	Avg.	Eff.	Total	Avg.
Hop	5.3	7.3	-	85.2	91.0	-	566.8	572	-	17749.7	17787.5	-
Brz	25.5	28.0	-	1393.6	1398.9	-	-	-	-	-	-	-
HK	2.3	4.0	8.9	25.3	28.9	9.0	23.2	28.9	98.9	317.5	341.6	99.0
HKe	0.9	2.1	2.4	5.4	10.5	2.4	1.4	5.9	2.6	14.3	34.9	3.4
HKs	0.6	1.3	2.4	2.8	4.6	2.4	0.8	2.0	2.7	9.1	21.3	3.4
HKn	0.7	2.2	3.0	51.5	56.2	29.7	1.3	6.8	3.7	29.4	51.7	15.4

Table 1: Running times for tests with complete accessible DFAs.

Table 1 shows the results of experimental tests with 10.000 pairs of complete ICDFAs. Due to space constraints, we only present the results for automata with $n \in \{5, 50\}$ states over an alphabet of $k \in \{2, 50\}$ symbols. Clearly, the methods which do not rely in minimisation processes are *a lot* faster. Below (Eff.) appears the *effective* time spent by the algorithm itself while below (Total) we show the *total* time spent, including overheads, such as making a DFA complete, initializing auxiliary data structures, etc. All times are expressed in seconds, and the algorithms that were not finished after 10 hours are accordingly signaled. The algorithm **Brz** is by far the slowest. The algorithm **Hop**, although faster, is still several orders of magnitude slower than any of the algorithms of the previous sections. We also present the average number of iterations (Iter.) used by each of the versions of algorithm **HK**, per pair of automata. Clearly, the refutation process is an advantage. **HKn** running times show that a linear set merging algorithm (such as UNION-FIND) is by far a better choice than a simple history (set) with pairs of states. **HKs** is a version of **HKe** which uses the automata string representation proposed by Almeida *et al.* [AMR07, RMA05]. The simplicity of the representation seemed to be quite suitable for this algorithm, and actually cut down both running times to roughly half. This is an example of the impact that a good data structure may have on the overall performance of this algorithm.

Table 2 shows the results of applying the same set of algorithms to NFAs. The testing conditions and notation are as before, adding only the *transition density* d as a new variable, which we define as the ratio of the number of transitions over the total number of possible

[2]http://www.ncc.up.pt/FAdo/node1.html

	$n = 5$						$n = 50$					
	$k = 2$			$k = 20$			$k = 2$			$k = 20$		
Alg.	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**
	Eff.	Total	Avg.	Eff.	Total	Avg.	Eff.	Total	Avg.	Eff.	Total	Avg.
Transition Density $d = 0.1$												
Hop	10.3	12.5	-	1994.7	2003.2	-	660.1	672.9	-	-	-	-
Brz	8.4	10.6	-	866.6	876.2	-	264.5	278.4	-	-	-	-
HKe	0.8	2.9	2.2	8.4	19	4	24.4	37.8	10.2	-	-	-
Transition Density $d = 0.5$												
Hop	17.9	19.8	-	2759.4	2767.5	-	538.7	572.6	-	-	-	-
Brz	14.4	16	-	2189.3	2191.6	-	614.9	655.7	-	-	-	-
HKe	2.6	4.3	4.9	36.3	47.3	10.3	6.8	48.9	2.5	294.6	702.3	11.5
Transition Density $d = 0.8$												
Hop	12.5	14.3	-	376.9	385.5	-	1087.3	1134.2	-	-	-	-
Brz	14	15.8	-	177	179.6	-	957.5	1014.3	-	-	-	-
HKe	1.4	3.2	2.7	39	49.9	10.7	7.3	64.8	2.5	440.5	986.6	11.5

Table 2: Running times for tests with 10.000 random NFAs.

transitions (kn^2). Although it is clear that **HKe** is faster, by at least one order of magnitude, than any of the other algorithms, the peculiar behaviour of this algorithm with different transition densities is not easy to explain. Considering the simplest example of 5 states and 2 symbols, the dataset with a transition density $d = 0.5$ took roughly twice as long as those with $d \in \{0.1, 0.8\}$. On the other extreme, making $n = 50$ and $k = 2$, the hardest instance was $d = 0.1$, with the cases where $d \in \{0.5, 0.8\}$ present similar running times almost five times faster. In our largest test, with $n = 50$ and $k = 20$, neither **Hop** nor **Brz** finished within the imposed time limit. Again, $d = 0.1$ was the hardest instance for **HKe**, which also did not finish within the time limit, although the cases where $d \in \{0.5, 0.8\}$ present similar running times.

	Size/Alg.	**Hop**	**Brz**	**AM**	**Equiv**	**EquivP**	**HKe**	**EquivUF**
$k = 2$	10	21.025	19.06	26.27	7.78	5.512	7.27	5.10
	50	319.56	217.54	297.23	36.13	28.05	64.12	28.69
	75	1043.13	600.14	434.89	35.79	23.46	139.12	60.09
	100	7019.61	1729.05	970.36	60.76	48.29	183.55	124.00
$k = 5$	10	42.06	25.99	32.73	9.96	7.25	8.69	6.48
	50	518.16	156.28	205.41	33.75	26.84	67.7	21.53
	75	943.65	267.12	292.78	35.09	25.17	161.84	28.61
	100	1974.01	386.72	567.39	54.79	45.41	196.13	37.02
$k = 10$	10	61.60	31.04	38.27	10.87	8.39	9.26	7.47
	50	1138.28	198.97	184.93	34.93	28.95	72.95	22.60
	75	2012.43	320.37	271.14	35.77	26.92	195.88	30.61
	100	4689.38	460.84	424.67	52.97	44.58	194.01	39.23

Table 3: Running times (seconds) for tests with 10.000 random r.e.

Table 3 presents the running times of the application of **HKe** to r.e. and their comparison with the algorithms presented by Almeida *et al.* [AMR08a], where **equiv** and **equivP** are the functional variants of the original **AM** algorithm. **equivUF** is the UNION-FIND improved version of **equivP**. Although the results indicate that **HKe** is not as fast as the direct comparison methods presented in the cited paper, it is clearly faster than any minimisation process. The improvements of **equivUF** over **equivP** are not significant (it is actually considerably slower for r.e. of length 100 with 2 symbols). We suspect that this is related to some optimizations applied by the `Python` interpreter. We state this based on the fact that

when both algorithms are executed using a profiler, **equivUF** is almost twice faster than **equivP** on most tests.

We have no reason to believe that similar tests with different implementations of these algorithms would produce significantly different ordering of its running times from the one here presented. However, it is important to keep in mind, that these are experimental tests that greatly depend on the hardware, data structures, and several implementation details (some of which, such as compiler optimizations, we do not utterly control).

6 Conclusions

As minimality or equivalence for (finite) transition systems is in general intractable, right-invariant relations (bisimulations) have been extensively studied for nondeterministic variants of these systems. When considering deterministic systems, however, those relations provide non-trivial improvements. We presented several variants of a method by Hopcroft and Karp for the comparison of DFAs which does not use automata minimization. By placing a refutation condition earlier in the algorithm we may achieve better running times in the average case. This is sustained by the experimental results presented in the paper. We extended this algorithm to handle NFAs. Using Brzozowski's automata, we showed that a modified version of Antimirov and Mosses' method translates directly to Hopcroft and Karp's algorithm.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AM94] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. In G. Rozenberg and A. Salomaa, editors, *Developments in Language Theory*, pages 195 – 209. World Scientific, 1994.

[AMR07] M. Almeida, N. Moreira, and R. Reis. Enumeration and generation with a string automata representation. *Theoret. Comput. Sci.*, 387(2):93–102, 2007.

[AMR08a] M. Almeida, N. Moreira, and R. Reis. Antimirov and Mosses's rewrite system revisited. In O. Ibarra and B. Ravikumar, editors, *CIAA 2008*, number 5448 in LNCS, pages 46–56. Springer-Verlag, 2008.

[AMR08b] M. Almeida, N. Moreira, and R. Reis. On the performance of automata minimization algorithms. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *CiE 2008: Abstracts and extended abst. of unpublished papers*, 2008.

[Ant96] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoret. Comput. Sci.*, 155(2):291–319, 1996.

[Brz63] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In J. Fox, editor, *Proc. of the Sym. on Math. Theory of Automata*, volume 12 of *MRI Symposia Series*, pages 529–561, NY, 1963.

[Brz64] J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, October 1964.

[CLRS03] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT, 2003.

[HK71] J. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California, 1971.

[HMU00] J. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2000.

[Hop71] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Proc. Inter. Symp. on Theo. of Mach. and Comp.*, pages 189–196. AP, 1971.

[HU73] J. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM J. Comput.*, 2(4):294–303, 1973.

[Koz94] D. C. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[Koz97] D. C. Kozen. *Automata and Computability*. Undergrad. Texts in Computer Science. Springer-Verlag, 1997.

[Koz08] D. Kozen. On the coalgebraic theory of Kleene algebra with tests. Computing and Information Science Technical Reports `http://hdl.handle.net/1813/10173`, Cornell University, May 2008.

[RMA05] R. Reis, N. Moreira, and M. Almeida. On the representation of finite automata. In C. Mereghetti, B. Palano, G. Pighizzini, and D.Wotschke, editors, *Proc. of DCFS'05*, pages 269–276, Como, Italy, 2005.

[Rut03] J.J.M.M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoret. Comput. Sci.*, 208(1–3):1–53, 2003.

[Sal66] A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the Association for Computing Machinery*, 13(1):158–169, 1966.

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22(2):215 – 225, April 1975.