Journal of Automata, Languages and Combinatorics **u** (v) w, x–y

Testing the Equivalence of Regular Languages [1]

Marco Almeida[2], Nelma Moreira, Rogério Reis
DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
e-mail: {mfa, nam, rvr}@ncc.up.pt

ABSTRACT

The minimal deterministic finite automaton is generally used to determine regular languages equality. Using Brzozowski's notion of derivative, 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 this algorithm's best-case running time, present an extension to non-deterministic finite automata, and establish a relationship with the one proposed in Almeida *et al.*, for which we also exhibit an exponential lower bound. We also present some experimental comparative results.

Keywords: regular languages, minimal automata, regular expressions, derivatives

1. Introduction

The minimal deterministic finite automaton is generally used for determining regular languages equality. Whether the languages are represented by deterministic finite automata, non-deterministic finite automata, or regular expressions, the usual procedure uses the equivalent (unique) minimal finite automaton to decide equivalence. The best known algorithm, in terms of worst-case complexity analysis, for finite automata minimisation is log-linear [11], and the equivalence problem is PSPACE-complete for both non-deterministic finite automata and regular expressions.

Based on the algebraic properties of regular expressions and the notion of derivative, Antimirov and Mosses proposed a terminating and complete rewrite system for deciding their equivalence [7]. In a paper about testing the equivalence of regular expressions, Almeida *et al.* [3] 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 [12] presented, in 1971, an almost linear algorithm for testing the equivalence of two deterministic finite automata that avoids their minimisation.

[1] This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI, and by project ASA (PTDC/MAT/65481/2006).
[2] Marco Almeida is funded by FCT grant SFRH/BD/27726/2006.

Considering the merge of the two automata 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 finite automaton is the coarsest relation in that condition.

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

All these algorithms are also closely related with the recent co-algebraic approach to automata developed by Rutten [20], where the notion of *bisimulation* corresponds to a 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 [16].

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.* [13] and Kozen [18].

An *alphabet* Σ is a nonempty set of letters. 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* (r.e.) 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 unnecessary 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 regular expressions 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 [21, 17]. Let ACI denote the associativity, commutativity and idempotence of $+$.

A *non-deterministic 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 $(q, a, q') \in \delta$. We consider the *size* of an NFA to be its number of states. 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 if $R \cap F_N \neq \emptyset$, and for all $a \in \Sigma$, $\delta_D(R, a) = \bigcup_{q \in R} \delta_N(q, a)$. This construction can be optimized by omitting states $R \in Q_D$ that are unreachable from the initial state. A DFA such that all states are accessible from the initial state is called *initially connected* (ICDFA).

Given a finite automaton $(Q, \Sigma, \delta, q_0, F)$, for $q \in Q$ 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, $S/_\equiv = \{[s]_\equiv \mid s \in S\}$. 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. In terms of worst-case complexity analysis, the best known algorithm for this procedure [11] runs in $O(kn \log n)$ time for a DFA with n states over an alphabet of k symbols.

Hopcroft and Karp [12] 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 [15, 22] showed that the running-time of this algorithm is actually $O(m \log^\star n)$ for $m \geq n$ FIND operations intermixed with $n - 1$ UNION[3] 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 [10], thus achieving a running time complexity $O(m\alpha(n))$

[3]Referred to as MERGE by Hopcroft and Ullman [14].

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 as 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 $p \in Q_i$, $i \in \{1, 2\}$.

First, we present the original algorithm by Hopcroft and Karp [1, 12] for testing the equivalence of two DFAs as Algorithm 1. It does not involve any minimisation process and is 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.

A very important subtlety of the UNION operation is that the two combined sets are destroyed in the end. This assures that after a function call such as UNION(p, q, q'), $|S_{q'}| = |S_p| + |S_q|$ 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 running time of the algorithm is $O(k(n + m))$, where $k = |\Sigma|$.

```
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 ∀S_i ∀p, q ∈ S_i   ε(p) = ε(q): return True
13     else: return False
```

Algorithm 1 The original **HK** algorithm.

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 into 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 **while** loop is executed once for each symbol in the alphabet, the total running 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 $2^{2^{2^{16}}}$), $\alpha(j) \leq 4$. 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 as 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 is it 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 increasingly high number of minimal ICDFAs observed by Almeida *et al.* [2], 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 (**HK**), 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 *Algorithm 1 (**HK**) and Algorithm 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 underlying idea is to embed the powerset construction into the algorithm, but this must be done with some caution. As any DFA is a particular case of an NFA, all the experimental results presented on Section 5 use this generalized approach, 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 $p \in Q_i$, $i \in \{1, 2\}$.

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 to define 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, to which we call **HKe**, is presented in Algorithm 3.

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

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

Lemma 2 *Algorithm 3 (**HKe**) is an extension of Algorithm 2 (**HKi**) 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 Aho *et. al* [1], 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

In this Section we present an algorithm to test the equivalence of regular expressions without converting them to the equivalent minimal automata, and relate it with the

algorithms presented in the previous section.

4.1. Antimirov and Mosses's algorithm

The *derivative* [9] of a regular expression 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 in the following way:

$$\epsilon^{-1}(\alpha) = \alpha;$$
$$w^{-1}(\alpha) = (u \cdot a)^{-1}(\alpha) = a^{-1}(u^{-1}(\alpha)).$$

We also have the following properties:

$$L(a^{-1}(\alpha)) = \{w \mid aw \in L(\alpha)\},$$

$$\alpha \sim \varepsilon(\alpha) + \sum_{a \in \Sigma} a \cdot a^{-1}(\alpha).$$

Considering regular expressions modulo the ACI axioms, Brzozowski [9] proved that, the set of derivatives of a regular expression α, $\mathcal{D}(\alpha)$, is finite. This result leads to the definition of *Brzozowski's automaton* which is equivalent to a given regular expression 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 [7] proposed[4] a rewrite system for deciding the equivalence of two extended regular expressions (with intersection), based on a complete axiomatization. This is a refutation method such that testing the equivalence of two regular expressions corresponds to an iterated process of testing the equivalence of their derivatives.

Not considering extended regular expressions, Algorithm 4 (**AM**) presents a version of Antimirov and Mosses method, which is, essentially, the one proposed by Almeida *et al.* [3]. 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]The idea of testing the equivalence of two regular expressions using the notion of *derivative* was already present in Brzozowski's PhD thesis

```
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 the r.e. equivalence test (**AM**).

4.2. A naïve HK algorithm

We now present, as Algorithm 5, a naïve version of the **HK** algorithm. It will be useful to prove its correctness and to establish a relationship to the **AM** method.

Definition 1 *Let R be defined as follows:*

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

Consider Algorithm 5 and 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. To prove its correctness we will show that the algorithm collects in H the pairs of states from the relation R.

```
1   def HKn(α, β):
2       S = {(p_0, q_0)}
3       H = ∅
4       while (p, q) = POP(S):
5           PUSH(H, (p, q))
6           for a ∈ Σ:
7               p′ = δ_1(p, a)
8               q′ = δ_2(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 of Algorithm 5 (**HKn**), $(p, q) \notin H$ 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 S and H are disjoint and 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 *Algorithm 5 (**HKn**) is terminating with time complexity $O(knm)$.*

Proof. The elements of S 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 (**HKn**), 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 $k < k'$. □

Lemma 6 *For all $(p, q) \in Q_1 \times Q_2$, $(p, q) \in S$ at some step of Algorithm 5 (**HKn**), 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 follows by induction on the length 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 of Algorithm 5 (**HKn**), 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 follows 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 one has that $(p, q) \in R$. □

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

Theorem 2 *Let A and B be DFAs. $A \sim B$ 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 $Q_1 \cup Q_2$ 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 $\forall (p, q) \in R^\star$, $\varepsilon(p) = \varepsilon(q)$.*

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 (**HKn**) and Algorithm 1 (**HK**) are equivalent.*

4.3. Equivalence of the two methods

It is possible to use Algorithm 4 (**AM**) to obtain 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 and 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).

Q_α and Q_β are the *Brzozowski's automata* of the regular expressions α and β, respectively.

In order to prove the equivalence of Algorithm 1 (**HK**) and Algorithm 4 (**AM**), we will first apply Theorem 1 to Algorithm 5 (**HKn**), transforming it into a refutation procedure. This modified version is presented as Algorithm 6. We will then show that the regular expressions equivalence test **AM** actually embeds Hopcroft and Karp's method while constructing the equivalent DFAs.

```
1  def HK(A,B)t:
2      S = {(p_0, q_0)}
3      H = ∅
4      while (p,q) = POP(S):
5          if ε(p) ≠ ε(q): return False
6          PUSH(H, (p,q))
7          for a ∈ Σ:
8              p' = δ_1(p,a)
9              q' = δ_2(q,a)
10             if (p',q') ∉ H:
11                 PUSH(S, (p',q'))
12     return True
```

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

Lemma 9 *Algorithm 4 (**AM**) embeds Algorithm 5 (**HKn**) while constructing the Brzozowski's DFAs.*

Proof. By Theorem 1, Algorithm 6 is equivalent to Algorithm 5, but including a refutation step. To verify that Algorithm 4 actually embeds Algorithm 6 while constructing the Brzozowski's DFAs, 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 $\varepsilon(\alpha) = \epsilon$, 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 1 (**HK**) applied to the Brzozowski's automata of the two regular expressions.*

Proof. By Corollary 2, Algorithm 5 and Algorithm 1 are equivalent. Applying Lemma 9 we have that Algorithm 4 embeds Algorithm 5 while constructing the Brzozowski's DFAs. Thus, applying Algorithm 4 to two regular expressions α and β is equivalent to the application of Algorithm 1 to the Brzozowski's automata of α and β. □

*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 this subsection we exhibit a family of regular expressions for which a non-deterministic version of the **AM** comparison method is exponential on the number of letters $|\alpha|_\Sigma$ of the regular expression α.

We proceed by showing that the NFA N of a regular expression α, obtained using *partial derivatives* [6] with the method described by Almeida *et al.* [5], is such that $|N| \in O(|\alpha|_\Sigma)$ and the number of states of the smallest equivalent DFA is exponential on $|N|$.

Partial derivatives are related to NFAs in the same natural way as derivatives are related to DFAs. Let α be a regular expression and $\alpha' = a^{-1}(\alpha)$. The partial derivatives of α w.r.t. the letter $a \in \Sigma$, denoted by $\partial_a(\alpha)$, can be seen as the set of the operands of the disjunction α'. The following recursive definition computes the set of partial derivatives of an arbitrary regular expression w.r.t a letter a.

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

This definition can trivially be extended to words and clearly $L(a^{-1}(\alpha)) = L(\sum \partial_a(\alpha))$. The set of all partial derivatives is finite [6] and denoted by $PD(\alpha)$.

Figure 1 presents a classical example of a bad behaved case (with $n+1$ states) of the powerset construction, by Hopcroft *et al.* [13]. Although this example does not reach the 2^{n+1} states bound, the smallest equivalent DFA has exactly 2^n states.

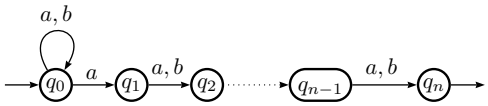

Figure 1: NFA which has no equivalent DFA with less than 2^n states.

Consider the regular expression family $\alpha_\ell = (a+b)^\star(a(a+b)^\ell)$, where $|\alpha_\ell|_\Sigma = 3+2\ell$. It is easy to see that the NFA in Figure 1 is obtained directly from the application of the non-deterministic **AM** method to α_ℓ, with the corresponding partial derivatives presented on Figure 2.

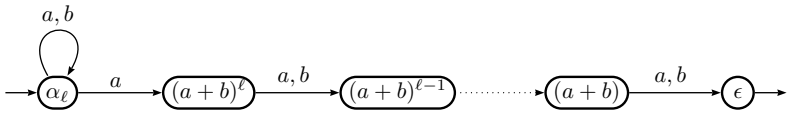

Figure 2: NFA obtained from the r.e. α_ℓ using the **AM** method.

The set of the partial derivatives $PD(\alpha_\ell) = \{\alpha_\ell, (a+b)^\ell, \ldots, (a+b), \epsilon\}$ has $\ell+2 = n+1$ elements, which corresponds to the size of the obtained NFA. The equivalent minimal DFA has $2^n = 2^{\ell+1}$ states.

5. Experimental results

In this section we present some experimental results of the previously discussed algorithms applied to DFAs, NFAs, and regular expressions We also include the same results of the tests using Hopcroft's (**Hop**) [11] and Brzozowski's (**Brz**) [8] automata minimization algorithms. Due to space constraints, the full set of experimental results can be found in Appendix A.

The random DFAs were generated using publicly available tools[5] [2]. The NFAs dataset was obtained with a set of tools described by Almeida *et al.* [4]. As the ICDFAs datasets were obtained with a uniform random generator, the size of each sample (10 000 elements) is sufficient to ensure a 95% confidence level within a 1% error margin. It is calculated with the formula $n = (\frac{z}{2\epsilon})^2$, where z is obtained from the normal distribution table such that $P(-z < Z < z)) = \gamma$, ϵ is the error margin, and γ is the desired confidence level.

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.

Table 1 shows the results of experimental tests with 10000 pairs of complete ICDFAs. We 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. Column *Eff.* shows the effective time spent by the algorithm itself while column *Total* presents the total time spent, including overheads, such as making

[5] http://www.ncc.up.pt/FAdo/

	$n = 5$						$n = 50$					
	$k = 2$			$k = 50$			$k = 2$			$k = 50$		
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 10 000 uniformly generated ICDFAs.

a DFA complete, initializing auxiliary data structures, etc. All times are expressed in seconds, and the algorithms that were not finished within a 10 hours time span are accordingly signaled. Brzozowski's algorithm, **Brz**, is by far the slowest. Hopcroft's 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.* [2, 19]. 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.

	$n = 5$						$n = 50$					
	$k = 2$			$k = 20$			$k = 2$			$k = 20$		
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.

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 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	AMo	Equiv	EquivP	HKe	EquivUF
$k = 2$	10	21.025	19.06	26.27	7.78	5.51	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 regular expressions and their comparison with the algorithms presented by Almeida *et al.* [3], where **equiv** and **equivP** are the functional variants of the original algorithm by Antimirov and Mosses (**AMo**). The NFAs were computed with Glushkov's algorithm, as described by Yu [23]. **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 regular expressions 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 without resorting to minimisation and extended it to NFAs. This approach provides much more time-efficient methods for checking the equivalence of automata and regular expressions. 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. Using Brzozowski's automata, we showed that a modified version of Antimirov and Mosses method translates directly to Hopcroft and Karp's algorithm.

References

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

[2] M. Almeida, N. Moreira, R. Reis, Enumeration and Generation with a String Automata Representation. *Theoret. Comput. Sci.* **387** (2007) 2, 93–102.

[3] M. Almeida, N. Moreira, R. Reis, Antimirov and Mosses's rewrite system revisited. In: O. Ibarra, B. Ravikumar (eds.), *CIAA 2008: 13th International Conference on Implementation and Application of Automata*. Number 5448 in LNCS, Springer, 2008, 46–56.

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

[5] M. Almeida, N. Moreira, R. Reis, Antimirov and Mosses's rewrite system revisited. *International Journal of Foundations of Computer Science* **20** (2009) 04, 669 – 684.

[6] V. M. Antimirov, Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theoret. Comput. Sci.* **155** (1996) 2, 291–319.

[7] V. M. Antimirov, P. D. Mosses, Rewriting Extended Regular Expressions. In: G. Rozenberg, A. Salomaa (eds.), *Developments in Language Theory*. World Scientific, 1994, 195 – 209.

[8] J. A. Brzozowski, Canonical Regular Expressions and Minimal State Graphs for Definite Events. In: J. Fox (ed.), *Proc. of the Sym. on Math. Theory of Automata*. MRI Symposia Series 12, NY, 1963, 529–561.

[9] J. A. Brzozowski, Derivatives of Regular Expressions. *JACM* **11** (1964) 4, 481–494.

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

[11] J. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton. In: *Proc. Inter. Symp. on the Theory of Machines and Computations*. Academic Press, Haifa, Israel, 1971, 189–196.

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

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

[14] J. Hopcroft, J. D. Ullman, *A linear list merging algorithm*. Technical report, Princeton University, 1971.

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

[16] 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, 2008.

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

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

[19] R. Reis, N. Moreira, M. Almeida, On the Representation of Finite Automata. In: C. Mereghetti, B. Palano, G. Pighizzini, D.Wotschke (eds.), *Proc. of DCFS'05*. Rap. Tec. DICO, Univ. di Studi Milano, IFIP, Como, Italy, 2005, 269–276.

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

[21] A. Salomaa, Two Complete Axiom Systems for the Algebra of Regular Events. *Journal of the Association for Computing Machinery* **13** (1966) 1, 158–169.

[22] R. E. Tarjan, Efficiency of a Good But Not Linear Set Union Algorithm. *JACM* **22** (1975) 2, 215 – 225.

[23] S. Yu, Regular languages. In: G. Rozenberg, A. Salomaa (eds.), *Handbook of Formal Languages*. 1, Springer, 1997.

A. Experimental results

	$n = 5$								
	$k = 2$			$k = 20$			$k = 50$		
Alg.	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**
	Eff.	Total	Avg.	Eff.	Total	Avg.	Eff.	Total	Avg.
Hop	4.8	6.0	-	34.1	36.4	-	84.3	87.8	-
Brz	17.8	19.2	-	356.6	358.9	-	874.8	878.3	-
HK	2.2	3.8	8.9	11.6	13.6	9.0	24.2	28.5	9.0
HKe	0.8	2.2	2.4	2.4	4.4	2.4	4.9	8.1	2.4
HKs	0.5	1.2	2.4	1.4	2.5	2.4	3.0	4.8	2.4
HKn	0.7	1.9	3.0	7.5	9.7	11.3	42.4	46.0	25.7
	$n = 10$								
	$k = 2$			$k = 20$			$k = 50$		
Alg.	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**
	Eff.	Total	Avg.	Eff.	Total	Avg.	Eff.	Total	Avg.
Hop	16.3	17.9	-	161.7	165.1	-	425.0	431.8	-
Brz	276.5	278.2	-	30625.3	30614.6	-	-	-	-
HK	4.9	6.7	18.9	22.5	24.0	19.0	51.8	56.2	19.0
HKe	0.9	2.7	2.5	2.8	6.3	2.6	5.7	11.5	2.6
HKs	0.5	1.3	2.5	2.0	3.6	2.6	3.6	6.7	2.6
HKn	0.6	2.4	3.2	9.0	12.1	12.2	47.9	53.2	27.0
	$n = 20$								
	$k = 2$			$k = 20$			$k = 50$		
Alg.	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**
	Eff.	Total	Avg.	Eff.	Total	Avg.	Eff.	Total	Avg.
Hop	61.6	64.3	-	742.6	748.1	-	1875.6	1886.1	-
Brz	-	-	-	-	-	-	-	-	-
HK	9.3	12.1	38.9	44.9	48.9	39.0	118.4	126.7	39.0
HKe	1.2	3.7	2.7	3.7	8.9	3.0	7.0	17.1	2.9
HKs	0.6	1.6	2.7	2.5	5.6	3.0	5.2	11.1	2.9
HKn	0.9	3.3	3.6	9.4	14.6	12.3	55.2	65.3	31.3
	$n = 50$								
	$k = 2$			$k = 20$			$k = 50$		
Alg.	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**
	Eff.	Total	Avg.	Eff.	Total	Avg.	Eff.	Total	Avg.
Hop	510.8	516.1	-	6300.6	6312.4	-	16652.0	16676.3	-
Brz	-	-	-	-	-	-	-	-	-
HK	23.9	29.2	98.9	124.5	135.0	99.0	298.5	318.3	99.0
HKe	1.3	7.0	2.7	5.3	18.5	3.4	10.6	36.0	3.4
HKs	0.7	2.2	2.7	4.3	10.7	3.4	8.8	23.5	3.4
HKn	1.1	6.3	3.7	14.0	26.7	17.4	28.1	53.2	15.4

Table 4: Running times for tests with 10 000 uniformly generated ICDFAs.

	$n = 5$						$n = 10$					
	$k = 2$			$k = 20$			$k = 2$			$k = 20$		
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	-	133.2	136.0	-	-	-	-
Brz	8.4	10.6	-	866.6	876.2	-	117.4	120.3	-	-	-	-
HKe	0.8	2.9	2.2	8.4	19	4	1.4	4.1	2.6	77.3	98.1	22.3
Transition Density $d = 0.5$												
Hop	17.9	19.8	-	2759.4	2767.5	-	37.4	40.8	-	5617.9	5636.0	-
Brz	14.4	16	-	2189.3	2191.6	-	32.0	35.0	-	768.8	793.1	-
HKe	2.6	4.3	4.9	36.3	47.3	10.3	2.1	5.3	3.2	224.4	250.2	38.5
Transition Density $d = 0.8$												
Hop	12.5	14.3	-	376.9	385.5	-	45.2	49.0	-	2348.0	2371.8	-
Brz	14	15.8	-	177	179.6	-	957.5	41.3	-	451.2	482.8	-
HKe	1.4	3.2	2.7	39	49.9	10.7	1.8	5.3	2.4	64.4	97.1	11.2

	$n = 50$					
	$k = 2$			$k = 20$		
Alg.	**Time (s)**		**Iter.**	**Time (s)**		**Iter.**
Transition Density $d = 0.1$						
Hop	660.1	672.9	-	-	-	-
Brz	264.5	278.4	-	-	-	-
HKe	24.4	37.8	10.2	-	-	-
Transition Density $d = 0.5$						
Hop	538.7	572.6	-	-	-	-
Brz	614.9	655.7	-	-	-	-
HKe	6.8	48.9	2.5	294.6	702.3	11.5
Transition Density $d = 0.8$						
Hop	1087.3	1134.2	-	-	-	-
Brz	957.5	1014.3	-	-	-	-
HKe	7.3	64.8	2.5	440.5	986.6	11.5

Table 5: Running times for tests with 10 000 random NFAs.