On Equivalence Checking of Nondeterministic Finite Automata

Chen Fu[1,3] ✉, Yuxin Deng[2], David N. Jansen[1], and Lijun Zhang[1,3]

[1] State Key Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing, China
fchen@ios.ac.cn
[2] Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
[3] University of Chinese Academy of Sciences, Beijing, China

Abstract. We provide a comparative study of some typical algorithms for language equivalence in nondeterministic finite automata and various combinations of optimization techniques. We find that their practical efficiency mostly depends on the density and the alphabet size of the automaton under consideration. Based on our experiments, we suggest to use HKC (Hopcroft and Karp's algorithm up to congruence) [4] if the density is large and the alphabet is small; otherwise, we recommend the antichain algorithm (Wulf, Doyen, Henzinger, Raskin) [6]. Bisimulation equivalence and memoisation both pay off in general. When comparing highly structured automata over a large alphabet, one should use symbolic algorithms.

1 Introduction

Checking whether two *nondeterministic finite automata (NFA)* accept the same language is important in many application domains such as compiler construction and model checking. Unfortunately, solving this problem is costly: it is PSPACE-complete [16].

However, the problem is much easier when restricted to *deterministic finite automata (DFA)* where nondeterminism is ruled out. Checking language equivalence for DFA can be done using either minimisation [10,12,18] or Hopcroft and Karp's algorithm (HK algorithm) [9]. The former searches for equivalent states in the whole state space of a finite automaton or the disjoint union of two automata. This works well in practice because for DFA, bisimulation equivalence, simulation equivalence and language equivalence coincide, and both bisimulation and simulation can be computed in polynomial time. The HK algorithm is more appropriate in the case where one only wants to know if two particular states are language equivalent because it is an "on-the-fly" algorithm that explores merely the part of state space that is really needed. It should be mentioned that the HK algorithm exploits a technique nowadays called *coinduction* [15].

A straightforward idea for checking the language equivalence of two NFA is to convert them into DFA through a standard powerset construction, and then

execute an equivalence checking algorithm for DFA. Since there are exponentially many state sets in the powerset, one would like to avoid constructing them as much as possible. In particular, if one only needs to decide if two specific sets of states in a nondeterministic finite automaton are equivalent, one can construct the state sets on-the-fly, and simultaneously try to build a bisimulation that relates these sets. With this approach, the number of constructed sets is usually much smaller than the exponential worst-case bound. In terms of implementation, it is easy to adapt a naive version of the HK algorithm from DFA to NFA: The algorithm maintains two sets *todo* and R. The set *todo* contains the pairs (X, Y) to be checked, where X and Y are two sets of NFA states. The set R contains the checked (equivalent) pairs. It is a bisimulation relation when the the algorithm terminates successfully. There are several optimizations for this algorithm:

1. Reduce the set R by constructing a relation that is not a bisimuation but is a bisimulation up to equivalence, e. g. the HK algorithm [9], or up to congruence, e. g. the HKC algorithm [4].
2. Often *todo* is implemented as a list that may contain repeated elements. Avoid these repetitions by some memoisation techniques [13].
3. Represent the automata symbolically rather than explicitly by using *binary decision diagrams (BDD)* [19,13].
4. Saturate the given automata with respect to bisimulation equivalence or simulation preorder.

An alternative approach to checking NFA equivalence is to use *antichain* algorithms [6,2]. The basic idea is to check language inclusion in both directions. This approach also exploits the *coinduction* technique: in order to check whether the language of a set of NFA states X is a subset of the language of a set of NFA states Y, it simultaneously tries to build a simulation relation, relating each state $x \in X$ (as a state in the NFA) to Y (as a state in the DFA). This algorithm can also be optimized by reducing the list *todo* by memoisation or by reducing the list R with antichains. The antichain algorithm can be enhanced by exploiting any preorder contained in language inclusion [2]. For example, the simulation preorder can be used for this purpose.

In this paper we investigate the mentioned algorithms and their combinations with optimizations to achieve the best time efficiency. We find that in most cases the antichain algorithm is stable and often outperforms other algorithms. In contrast, the performance of HK and HKC algorithms may vary a lot, depending on the size of the alphabet and the density of transitions in the automata under consideration. When the size of the alphabet is small (e. g. 2) and the density is large (e. g. 1.25 or 1.5), HKC is the best choice. Otherwise, computing congruence closures is very costly and renders HKC impractical to use. One should use memoisation because it mostly accelerates the algorithms. Further, if the considered automata are highly structured and over a large alphabet, one should try the symbolic algorithms because the BDDs are usually small in such situations. Finally we suggest to minimize the automata by bisimilarity instead of saturating the automata by similarity before performing the algorithms. Although the

latter is more powerful, the time efficiency of computing the bisimilarity makes the total time shorter.

The rest of this paper is organized as follows. In Section 2 we recall some basic concepts. In Section 3 we introduce the HK, HKC and antichain algorithms and relevant optimizations. In Section 4 we assess those techniques introduced previously by comparing their running times experimentally. We discuss related work in Section 5 and summarize our recommendations in the concluding Section 6.

2 Preliminaries

Finite Automata. A Nondeterministic Finite Automaton (NFA) A is a tuple $(\Sigma, Q, I, F, \delta)$ where: Σ is an alphabet, Q is a finite set of states, $I \subseteq Q$ is a non-empty set of initial states, $F \subseteq Q$ is a set of accepting states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. A word $u = u_1 u_2 \dots u_n$ is accepted by $q \in Q$ if there exists a sequence $q_0 u_1 q_1 u_2 \dots u_n q_n$ such that $q_0 = q$, $q_j \in \delta(q_{j-1}, u_j)$ for all $0 < j \leq n$ and $q_n \in F$. Define $L(q) = \{u \mid u \text{ is accepted by } q\}$ as the language of q and $L(A) = \bigcup_{q \in I} L(q)$ as the language of A. Two NFA A, B are said to be language equivalent iff $L(A) = L(B)$.

An NFA is called deterministic if $|I| = 1$ and $|\delta(q, a)| \leq 1$ for all $q \in Q$ and $a \in \Sigma$. For each NFA $A = (\Sigma, Q, I, F, \delta)$, we can use the standard powerset construction [3, Sect. 4.1] to transform it to a DFA $A^\sharp = (\Sigma, Q^\sharp, I^\sharp, F^\sharp, \delta^\sharp)$ with the same language.

The NFA equivalence checking problem is to decide whether two given NFA accept the same language.

Simulation and Bisimulation

Definition 1. *Let R, $R' \subseteq Q \times Q$ be two binary relations on states, we say that R* s-progresses to *R', denoted $R \rightarrowtail_s R'$, if $x\ R\ y$ implies:*

- *if x is accepting, then y is accepting;*
- *for any $a \in \Sigma$ and $x' \in \delta(x, a)$, there exists some $y' \in \delta(y, a)$ such that $x'\ R'\ y'$.*

A simulation is a relation R such that $R \rightarrowtail_s R$ and a bisimulation is a relation R such that $R \rightarrowtail_s R$ and $R^{-1} \rightarrowtail_s R^{-1}$, where R^{-1} is the inverse relation of R.

The largest simulation and bisimulation are called *similarity* and *bisimilarity,* denoted by $\precsim$ and $\sim$, respectively. For any NFA, if a bisimulation between two states can be found, then they are language equivalent. Similarly, for any NFA, if a simulation between two states can be found, for example $x \precsim y$, then $L(x) \subseteq L(y)$. The reverse direction of these two implications holds in general only for DFA. Computing similarity needs $O(|\delta| \cdot |Q|)$ time [7,14,1,8], while computing bisimilarity is faster, as it is in $O(|\delta| \cdot \log |Q|)$ [18]. Bisimulation is a sound proof technique for checking language equivalence of NFA and it is also complete for DFA. Simulation is a sound proof technique for checking language inclusion of NFA and it is also complete for DFA.

Binary Decision Diagrams. A standard technique [19,13] for working with automata over a large alphabet consists in using BDDs to represent the automata. A Binary Decision Diagram (BDD) over a set of variables $X_n = \{x_1, x_2, \ldots, x_n\}$ is a directed, acyclic graph having leaf nodes and internal nodes. There is exactly one root node in a BDD; each internal node is labelled with a variable and has two outgoing edges whose ends are other nodes. The leaf nodes are labelled with 0 or 1. After fixing the order of variables, any BDD can be transformed into a reduced one which has the fewest nodes [5]. In the sequel, we only work with reduced ordered BDDs, which we simply call BDDs.

BDDs can be used to represent functions of type $2^{X_n} \to \{0, 1\}$. Here, we use BDDs to represent NFA. The advantage is that one often does not need many variables. For example, if there are 2^k letters, one only needs k variables to encode (the characteristic function of) a set of letters.

3 Algorithms and Optimizations

3.1 A Naive Algorithm for Language Equivalence checking

A naive adaptation of Hopcroft and Karp's algorithm from DFA to NFA is shown in Algorithm 1. Starting with the two sets of initial states, we do the powerset construction on-the-fly for both NFA and simultaneously try to build a bisimulation relating these two sets. The sets of states of the NFA become the states of the DFA constructed by powerset construction. We use two sets: *todo* and R. We call a pair (X, Y) a bad pair if one of X and Y is accepting but the other is not. Whenever we pick a pair from the set *todo*, we check if it is a bad pair; if it isn't, we generate their successors and insert these successor pairs into *todo*. The set R is used to store the processed pairs: if a pair is in R, the

Algorithm 1. The `Naive Eq` algorithm for checking NFA equivalence

Input: two NFA $A = (\Sigma, Q_A, I_A, F_A, \delta_A)$ and $B = (\Sigma, Q_B, I_B, F_B, \delta_B)$
Output: "Yes" if $L(A) = L(B)$, otherwise "No"

```
 1: R := ∅, todo := {(I_A, I_B)}
 2: while todo ≠ ∅ do
 3:     Pick (X, Y) ∈ todo and remove it
 4:     if (X, Y) ∉ R then
 5:         if (X, Y) is a bad pair then
 6:             return "No, L(A) ≠ L(B)."
 7:         end if
 8:         for all a ∈ Σ do
 9:             todo := todo ∪ {(δ_A^♯(X, a), δ_B^♯(Y, a))}
10:         end for
11:         R := R ∪ {(X, Y)}
12:     end if
13: end while
14: return "Yes, L(A) = L(B)."
```

states are language equivalent if the pairs in *todo* are language equivalent. In a formula, $R \rightarrowtail_s R \cup todo$ and $R^{-1} \rightarrowtail_s R^{-1} \cup todo^{-1}$. If the algorithm terminates with the return value "yes", R is a bisimulation between $A^\sharp$ and $B^\sharp$.

When checking (X, Y), the algorithm eventually determinizes both parts corresponding to X and Y, that is, it compares X (regarded as state of $A^\sharp$) with Y (regarded as state of $B^\sharp$). Based on the naive algorithm, one can imagine several optimizations: One idea is to try to reduce the number of pairs in R; the algorithm proposed in [9] by Hopcroft and Karp (called the HK algorithm) does so. In [4] Bonchi and Pous extend the HK algorithm by exploiting the technique of bisimulation up to congruence and obtain the HKC algorithm, in which R contains even fewer pairs. Another idea is to reduce the number of pairs in *todo* by so-called memoisation. The observation is very simple: one does not need to insert the same pair into *todo* more than once. (In practice, the set *todo* is often implemented as a list, so it actually can "contain" an element multiple times.) Besides these, one can do some preprocessing: One can use bisimilarity or similarity to saturate the NFA before running the algorithms, in the hope to accelerate the main algorithm. In addition, we also test whether it is a good idea to use BDDs to represent transition functions and then perform symbolic algorithms.

3.2 Reducing R

We only need to know whether there *exists* a bisimulation relating two sets, and it is unnecessary to build the whole bisimulation. So we can reduce R to a relation that is contained in – and sufficient to infer – a bisimulation.

HK algorithm. Hopcroft and Karp [9] propose that if an encountered pair is not in R but in its reflexive, symmetric and transitive closure, we can also skip this pair. Ignoring the concrete data structure to store equivalence classes, the HK algorithm consists in simply replacing Line 4 in Algorithm 1 with

4: **if** $(X, Y) \notin rst(R)$ **then**

where *rst* is the function mapping each relation $R \subseteq \mathcal{P}(Q) \times \mathcal{P}(Q)$ into its reflexive, symmetric, and transitive closure. With this optimization, the number of pairs in R will be reduced. When the algorithm returns "yes", R is no longer a bisimulation, but is contained in a bisimulation [4] and one can infer a bisimulation from R.

HKC algorithm. Based on the simple observation that if $L(X_1) = L(Y_1)$ and $L(X_2) = L(Y_2)$ then $L(X_1 \cup X_2) = L(Y_1 \cup Y_2)$, Bonchi and Pous [4] improve the HK algorithm with congruence closure. One gets the HKC algorithm just by replacing Line 4 in Algorithm 1 with

4: **if** $(X, Y) \notin rstu(R \cup todo)$ **then**

where *rstu* is the reflexive, symmetric, and transitive closure extended with the following union of relations: $u(R)$ is the smallest relation containing R satisfying: if $X_1\ R\ Y_1$ and $X_2\ R\ Y_2$, then $(X_1 \cup X_2)\ u(R)\ (Y_1 \cup Y_2)$. Note that Bonchi and Pous use $R \cup \mathit{todo}$ rather than R because this helps to skip more pairs, and this is safe since all pairs in *todo* will eventually be processed [4].

When the HKC algorithm returns "yes", R is also contained in a bisimulation [4] and sufficient to infer one.

In order to check whether a pair is in the equivalence closure of a relation, Hopcroft and Karp use disjoint sets forests to represent equivalence classes, which allow to check $(X, Y) \notin \mathit{rst}(R)$ in almost constant amortised time. Unfortunately, this data structure cannot help one to do the checking for congruence closure *rstu*. Bonchi and Pous use a set rewriting approach to do this. However, this requires to scan the pairs in R one by one, which makes it slow. As we shall show in the experiments, this has a great impact on the performance of HKC.

3.3 Reducing *todo*

The pairs in *todo* are those to be processed. However, if this set is implemented as a list or similar data structure, there are often redundancies. We can remember that some element has already been inserted into *todo* earlier; this is called *memoisation.* In Line 9, we check whether we have inserted the same pair into *todo* earlier and only insert the pair if it never has been in *todo*. (Note that this also skips pairs that have in the meantime moved from *todo* to R.) We can use hash sets to check this condition in constant time.

3.4 BDD representation

Pous [13] proposed a symbolic algorithm for checking language equivalence of finite automata over large input alphabets. By processing internal nodes, the symbolic algorithm may insert fewer pairs into *todo*, which makes us want to know whether it can save time if we perform the symbolic algorithm instead of the explicit one.

The symbolic version of the HK and HKC algorithms and of memoisation can be easily constructed from the explicit ones. The only difference is that the pairs of sets of states become pairs of BDD nodes, including leaf nodes and internal nodes. If a pair of internal nodes is skipped, then all its successors are also skipped. This is why the symbolic algorithm may have fewer pairs in *todo*.

3.5 Preprocessing operations

Bonchi and Pous [4] extend the HKC algorithm to exploit the similarity preorder. It suffices to notice that for any similarity pair $x \precsim y$ (in the NFA), we have $\{x, y\} \sim \{y\}$ (in the DFA). So to check whether $(X, Y) \in \mathit{rstu}(R \cup \mathit{todo})$, it suffices to compute the congruence closure of X and Y w. r. t. the pairs from $R \cup \mathit{todo} \cup \{(\{x, y\}, \{y\}) | x \precsim y\}$. This may allow to skip more pairs. However, the time required to compute similarity may be expensive.

Since bisimilarity can be computed in less time than similarity, it may be advantageous to replace similarity with bisimilarity. So we can replace the similarity with bisimilarity to get another algorithm, i. e. computing the congruence closure of X and Y w. r. t. the pairs from $R \cup \mathit{todo} \cup \{(\{x\}, \{y\}) | x \sim y\}$.

As a matter of fact, we can use this technique as a preprocessing operation. For similarity, we saturate the original NFA w. r. t. the similarity preorder before running the algorithms. For bisimilarity, we choose another approach, that is taking a quotient according to bisimilarity. This amounts to saturating the NFA w. r. t. bisimilarity, but it is more efficient. Note that we do not take a quotient according to simulation equivalence, because it is less powerful than saturating the NFA w. r. t. similarity. Although taking a quotient according to bisimilarity may be less powerful than saturating by similarity, using bisimilarity makes the total time shorter in many cases, which is shown in Section 4.

3.6 Algorithms for Language Inclusion Checking

Instead of directly checking language equivalence for NFA, it is possible to check two underlying language inclusions: for any pair (X, Y), we check whether $L(X) \subseteq L(Y)$ and $L(Y) \subseteq L(X)$. If both of them hold, we have $L(X) = L(Y)$. So it is enough to solve the problem of checking $L(X) \subseteq L(Y)$. The naive algorithm is quite similar to the one for checking equivalence. Here, we call a pair (x, Y) a bad pair if x is accepting but Y not. The idea of the algorithm is still: Whenever we pick a pair from *todo*, we check whether it is a bad pair; if not, we insert the successor pairs into *todo*.

When checking (X, Y), the algorithm eventually determinizes the part corresponding to Y and remains nondeterministic for X, that is, it compares $x \in X$ from A and Y from $B^\sharp$. The sets *todo* and R will therefore be subsets of $Q_A \times \mathcal{P}(Q_B)$. Again, the naive algorithm allows for several optimizations. Memoisation can be used without modifications. The antichain algorithm proposed in [6] aims to reduce the number of pairs in R, and it can be enhanced by exploiting similarity [2].

Given a partial order $(X, \sqsubseteq)$, an *antichain* is a subset $Y \subseteq X$ containing only incomparable elements. The antichain algorithm exploits antichains over the set $Q_A \times \mathcal{P}(Q_B)$, equipped with the partial order $(x_1, Y_1) \sqsubseteq (x_2, Y_2)$ iff $x_1 = x_2$ and $Y_1 \subseteq Y_2$.

In order to check $L(X) \subseteq L(Y)$ for two sets of states X, Y, the antichain algorithm ensures that R is an antichain of pairs (x', Y'). If one of these pairs p is larger than a previously encountered pair $p' \in R$ (i. e. $p' \sqsubseteq p$) then the language inclusion corresponding to p is subsumed by p' so that p can be skipped. Conversely, if there are some pairs $p_1, \ldots, p_n \in R$ which are all larger than p (i. e. $p \sqsubseteq p_i$ for all $1 \leq i \leq n$), one can safely remove them: they are subsumed by p and, by doing so, the set R remains an antichain. We denote the antichain algorithm as "AC".

Abdulla et al. [2] propose to accelerate the antichain algorithm by exploiting similarity. The idea is that when processing a pair (x, Y), if there is a previously encountered pair (x', Y') such that $x \precsim x'$ and $Y' \precsim Y$ (which means $\forall y' \in$

$Y', \exists y \in Y : y' \precsim y$), then (x, Y) can be skipped because it is subsumed by (x', Y'). For the same reason as in Sect. 3.5, we can also use bisimilarity instead of similarity.

Here, we can still take a quotient according to bisimilarity before performing the algorithms. However, we can not saturate the NFA like in Section 3.5 because the algorithms need to maintain an antichain and if we saturate the NFA, there would be lots of pairs which can be actually skipped. So we can only exploit similarity while running the algorithms, which sometimes slows them down.

4 Experiments

In this section, we describe some experiments to compare the performance of all the algorithms.

We implemented all the algorithms mentioned above in Java. For the symbolic algorithms, we use the JavaBDD library [20]. We implemented the algorithm in [18] to compute bisimilarity and the algorithm in [8] to compute similarity, which, as far as we know, are the two fastest algorithms to compute bisimilarity and similarity, respectively. All the implementation details are available at `https://github.com/fuchen1991/EBEC`.

We conducted the experiments on random automata and automata obtained from model-checking problems. All the experiments were performed on a machine with an Intel i7-2600 3.40 GHz CPU and 8 GB RAM.

Random automata: We generate different random NFA by changing three parameters: the number of states ($|Q|$), the number of letters ($|\Sigma|$), and the density (d), which is the average out-degree of each state with respect to each letter. Although Tabakov and Vardi [17] empirically showed that one statistically gets more challenging NFA with $d = 1.25$, we find that with different densities, the performance of algorithms varies a lot and the densities of many NFA from model checking are much smaller than this value, so we test more values for this parameter. For each setting, we generated 100 NFA. To make sure that all the algorithms meet their worst cases, there are no accepting states in the NFA (So we have to skip the operation of removing non-coaccessible states, otherwise this reduces each NFA to the empty one). The two initial state sets are two distinct singleton sets.

Automata from model checking: Bonchi et al. [4] use the automata provided by L. Holík, which come from the model checking of various programs (the bakery algorithm, bubble sort, and a producer-consumer system). We also use these automata. The difference between our work and Bonchi et al. is that they only show the performance of HKC and AC after preprocessing with similarity, while we compare more algorithms.

We record the running time of each algorithm on the above NFAs, measured as the average over four executions. We depict all the data by boxplots – a method for graphically depicting groups of numerical data through their quartiles. In a boxplot, the box denotes the values between the lower quartile

and the upper quartile, and the horizontal line in the box denotes the median value. Numbers which are outside 1.5 times the interquartile range above the upper quartile or below the lower quartile are regarded as outliers and shown as individual points. Finally, the two end points of the vertical line outside the box denote the minimal and maximal values that are not considered to be outliers.

4.1 Memoisation accelerates the algorithms

In most situations, memoisation saves time because it reduces the pairs in *todo*. But the effects for the three algorithms are different, that is, it saves more time to optimize HKC with memoisation than HK and AC. For HKC, repeated pairs are skipped because they are in the congruence closure of R (Algorithm 1, Line 4). This check costs much more time than the corresponding checks of HK and AC. Besides, memoisation needs less time than all the three methods to remove repeated pairs. So as we can see in Figure 1, the huge difference of the number of pairs in *todo* leads to the huge difference of time for HKC, but leads to small difference for HK and AC.

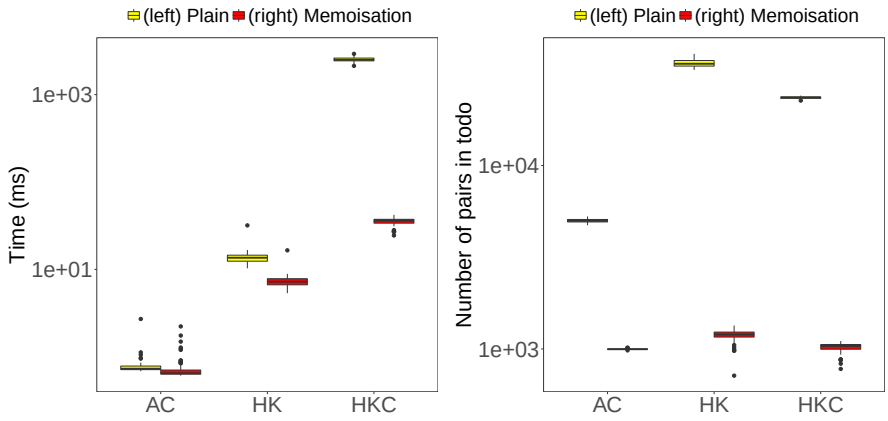


Fig. 1. Comparison of algorithms with and without memoisation ($|Q| = 500$, $|\Sigma| = 50$, $d = 0.1$)

When memoisation only reduces a few pairs in *todo*, it just costs a little more time because memoisation requires only constant time. So it always pays off to use memoisation. In the follwoing, we always use memoisation for HK, HKC, and AC.

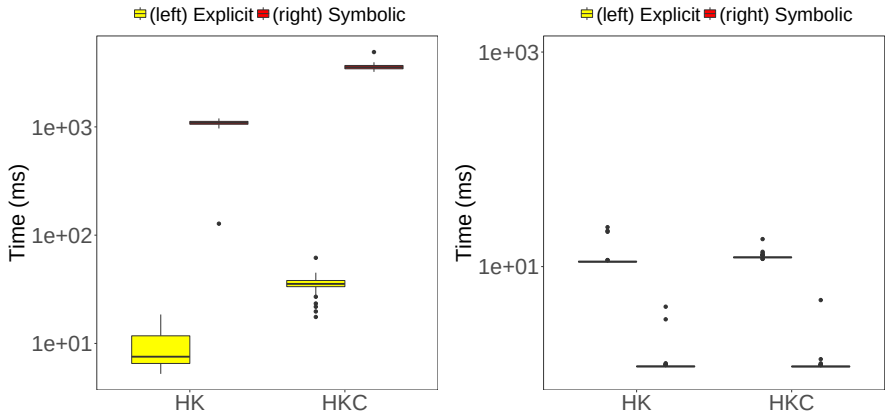
4.2 BDDs are only suitable for highly structured NFA

As discussed in Section 3.4, storing NFA with BDDs and performing symbolic algorithms can reduce the pairs in *todo*. However, we find that this variant slows

down the algorithms on random NFA for most settings, as shown in Figure 2(a). This is because it is hard to generate highly structured random NFA.

The size of the BDD highly depends on the structure of the automaton. If an automaton has many symmetries, there are fewer nodes in the BDD; this makes the symbolic algorithms run faster. Also, the performance of explicit algorithms on automata over large alphabet is bad, while BDDs can represent large alphabets with few variables, so symbolic algorithms are preferable for this kind of automata.

In order to show this, we let $|Q| = 4$, $|\Sigma| = 2^{17} = 131072$, and $d = 4.0$, which may be impractical, but a clear example to exhibit the advantage of symbolic algorithms. In this NFA, each state has a transition to all states on all input symbols, so the algorithms only need to process two pairs, namely the pair of initial state sets and the pair of sets containing all states. An explicit algorithm needs to scan every symbol in Σ, while there is only one node in the BDD, which makes symbolic algorithm faster. The result is shown in Figure 2(b).



(a) $|Q| = 500,\ |\Sigma| = 50,\ d = 0.1$ (b) $|Q| = 4,\ |\Sigma| = 131072,\ d = 4.0$

Fig. 2. Comparison of explicit and symbolic algorithms

4.3 Comparison of HK, HKC and AC

In addition to $|Q|$, $|\Sigma|$ and d, there is another parameter that should be considered: the number of transitions, $|\delta| = |Q| \cdot |\Sigma| \cdot d$. Here, we compare the performance of HK, HKC and AC under different settings of these parameters.

First, let us fix $|\Sigma| = 2$ and $d = 1.25$ and vary the state space size $|Q|$. The result is shown in Figure 3(a). When increasing $|Q|$ (and also $|\delta|$), the time required by all the three algorithms increases, and HK increases much faster

than the others. HKC performs best, and AC has some bad outliers, which can be more than 100 times slower than HKC.

Then, we fix $|Q| = 150$ and $d = 1.25$ and vary the alphabet size $|\Sigma|$. The result is shown in Figure 3(b): Upon increasing $|\Sigma|$ (and also $|\delta|$), the time required by all the three algorithms increases, and HK increases much faster than the others. The average performance of AC is best overall; however, if $|\Sigma| = 2$, there are some very slow outliers, so HKC may be preferable, as its performance is not much worse than the average of AC.

Next, we fix $|Q| = 300$ and $|\Sigma| = 10$ and vary the density d. The measurements are shown in Figure 3(c). Basically, the time first increases then decreases as the density grows. But the peak values appear at different densities for the three algorithms. The peak value of HKC appears to be near $d = 1$, but for HK and AC at a much larger density. We do not expect very large densities in practice; if the density is maximal ($d = |Q|$), then it can be found very quickly that all states are language equivalent. Still, HK always performs worst. When the density is between 0.1 and 1.25, AC performs 10 to 100 times faster than HKC, but when the density is 1.5, HKC can be several times faster than AC.

Until now, we have seen that HK always performs worse than at least one of the other two algorithms, and no one of HKC and AC always performs best. In the following, we only compare the performance of HKC and AC to find out in which situation HKC performs better and in which situation AC performs better.

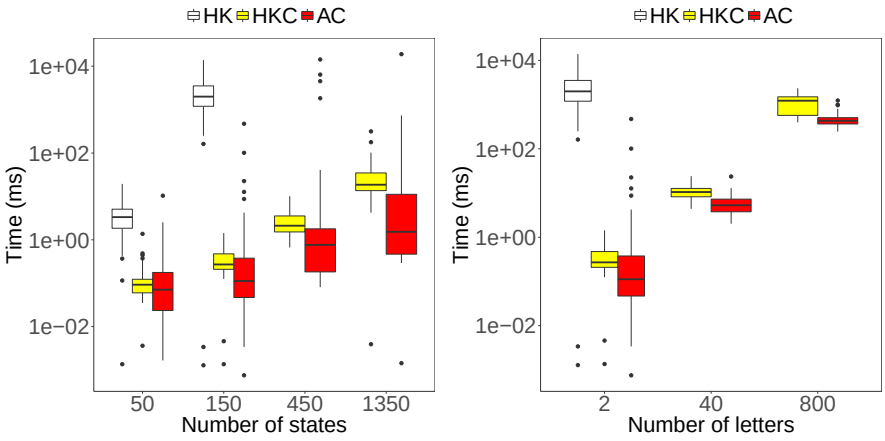
Let us fix $|Q| = 500$ and $|\delta| = 2500$ and let $|\Sigma|$ and d vary. The result is shown in Figure 3(d). The running time remains approximately the same as we fix $|Q|$ and $|\delta|$, but HKC performs better at $|\Sigma| = 2, d = 1.25$ and worse in other settings. Then we fix $|\Sigma| = 2$ and $|\delta| = 2500$. The result is shown in Figure 3(e). We can see that HKC is always worse except $d = 1.25$. Finally, we fix $|\Sigma| = 50$ and $|\delta| = 25000$. The result is shown in Figure 3(f). We can see that HKC is always slower than AC when $|\Sigma| = 50$. The smaller the density is, the larger difference between the performance of HKC and AC is.

In conclusion, HKC performs better when d is large and $|\Sigma|$ is small. In this setting, the maximal value of AC is always very large, which can be 10 to 100 times slower than HKC. Moreover, there are always some bad outliers for AC. So HKC is a better choice in this setting. But If d is small or $|\Sigma|$ is large, AC can even be more than 100 times faster than HKC.

4.4 Automata from model checking

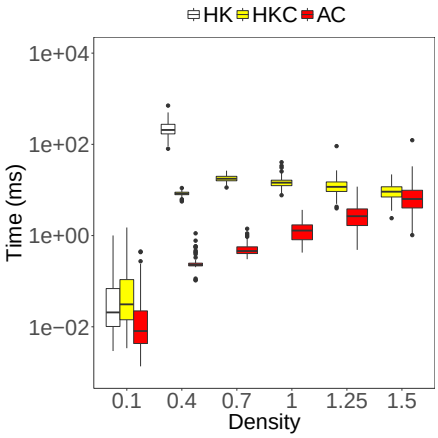
Now we compare the performance of HK, HKC, AC and all these three algorithms with preprocessing with similarity and bisimilarity on the automata from model checking. Bonchi and Pous state that HKC can mimic AC even for language inclusion problem. Here, we also use these algorithms to check language inclusion for the automata.

We perform all the three algorithms without any preprocessing ("Plain"), with saturating w. r. t. similarity ("Sim"), and minimizing w. r. t. bisimilarity

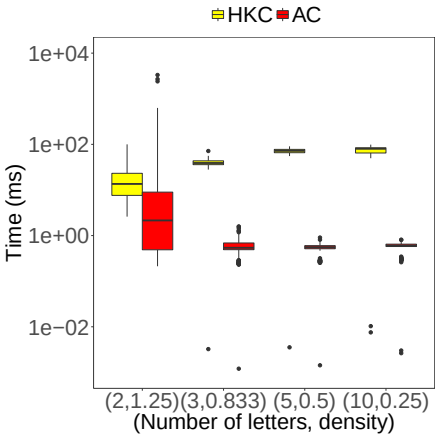


(a) $|\Sigma| = 2,\ d = 1.25$

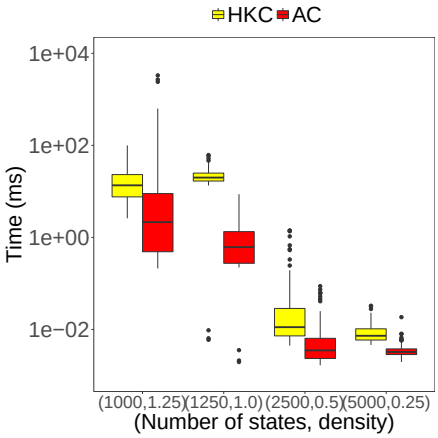
(b) $|Q| = 150,\ d = 1.25$



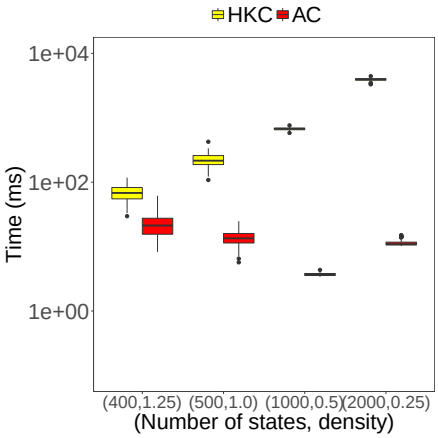
(c) $|Q| = 300,\ |\Sigma| = 10$



(d) $|Q| = 1000,\ |\delta| = 2500$



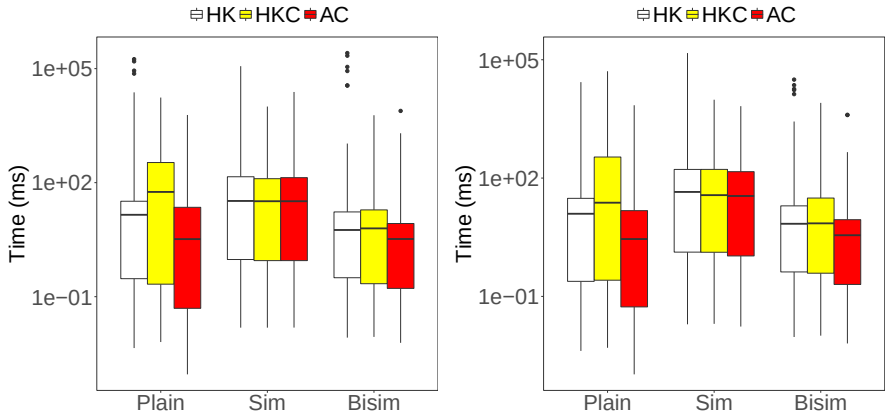
(e) $|\Sigma| = 2,\ |\delta| = 2500$



(f) $|\Sigma| = 50,\ |\delta| = 25000$

Fig. 3. Comparison of HK, HKC and AC

("Bisim"), respectively. We separate the results into those for which the inclusion holds and those for which the inclusion does not hold. Figure 4 shows the total running time of each algorithm. First, we find that the densities of these automata are all between 0.05 and 0.49, and over half of them are smaller than 0.19. Their alphabet sizes are between 7 and 36, and over half of them are smaller than 20. We also observe that the performance difference of the algorithms is similair to Figure 3(c) with $d = 0.1$. This is approximately in agreement with our conclusion for random automata. Second, as we can see the total running time of preprocesssing with bisimilarity is much shorter than similarity because computing bisimilarity is often much faster than similarity. Preprocesssing with similarity may even be slower than no preprocessing at all, but this does not happen for bisimilarity.



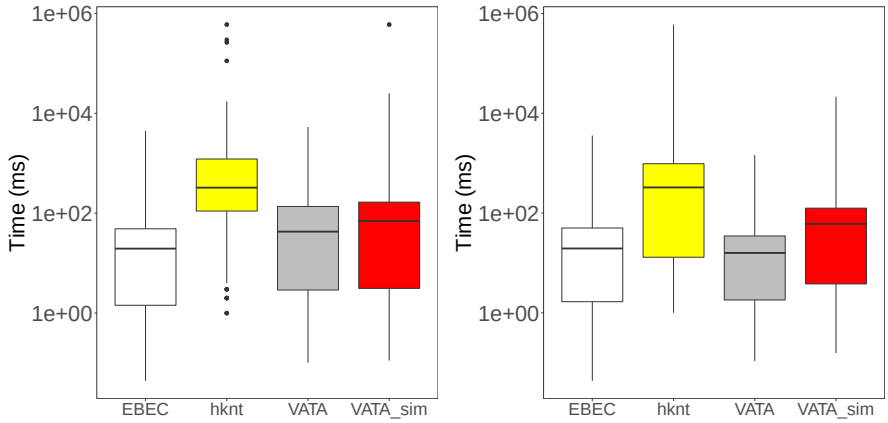
(a) Inclusion holds (b) Inclusion does not hold

Fig. 4. Total time of algorithms with different preprocessing operations

4.5 Tools: EBEC, hknt, and VATA

We also conducted experiments with other tools. Abdulla et al. [2] implemented their algorithm in their tool "VATA" [11] in C++, and Bonchi and Pous [4] implemented the HKC algorithm in their tool "hknt" in OCaml.

We again run experiments on the same automata sets used in Section 4.4. The result is shown in Figure 5. In this figure, "EBEC" denotes our tool in which we choose the antichain algorithm with memoisation and preprocessing with bisimilarity, since it is the optimal combination according our previous experiments. "hknt" denotes Bonchi and Pous's tool running their HKC algorithm and "VATA, VATA_sim" the tool of Abdulla et al. running the basic antichain algorithm and antichain algorithm with preprocessing with similarity respectively.



(a) Inclusion holds (b) Inclusion does not hold

Fig. 5. Comparison of the tools

We choose 10 minutes as the timeout and find that for a few tests, hknt and VATA_sim does not terminate in this time, while EBEC and VATA both terminate. We also see that our tool performs about 10 times faster than hknt in both situations. When the inclusion holds, EBEC is 2–3 times faster than VATA and VATA_sim. When the inclusion does not hold, our tool EBEC has the same performance as VATA on most automata and is up to 2.5 times slower on some automata. However, one should note that if the inclusion does not hold, the performance highly depends on the order of visiting the states.

5 Related Work

To our knowledge, VATA implemented by Abdulla et al. [2] is the most efficient implementation currently available for the antichain algorithms. They compare their algorithm with the naive algorithm and the basic antichain algorithm [6], but not with other algorithms.

Bonchi and Pous propose that HKC can be optimized by similarity [4], but their implementation of the algorithm to compute similarity is slow, because it is based on the algorithm proposed by Henzinger et al. [7], which is not fast enough nowadays. Bonchi and Pous compare HKC with HK and AC only on random automata, and they also compare HKC and AC after preprocessing with similarity. However, they do not show the total time of these two algorithms. In our work, we find that preprocessing with similarity is not that good, even makes the total time longer sometimes. So we propose preprocessing with bisimilarity, and the experiments show that it is preferable.

Pous [13] proposes a symbolic algorithm for checking language equivalence of finite automata over large alphabets and applies it to Kleene algebra with tests.

We have implemented the symbolic version of HK and HKC, and show that the symbolic algorithm is suitable for highly structured automata, especially those over large alphabets.

6 Conclusion

We have reviewed various algorithms and optimization techniques for checking language equivalence of NFA, and compared their performance by experiments.

We find that their practical efficiencies depend very much on the automata under consideration. But according to the automata to be checked, one can choose the appropriate algorithm: If the density is large and the alphabet size is small, then one should choose HKC (Hopcroft and Karp's algorithm up to congruence) [4], otherwise the antichain algorithm (Wulf, Doyen, Henzinger, Raskin) [6]. Moreover, one should always use memoisation and minimize the automata by bisimilarity before performing the algorithm. One may choose to use a symbolic algorithm when working with highly structured automata over a large alphabet. Finally, we compared the performance of our tool "EBEC" with "VATA" and "hknt" and showed that EBEC can perform better on most automata we tested.

Acknowledgements

This work has been supported by the National Natural Science Foundation of China (Grants 61532019, 61472473, 61672229), the CAS/SAFEA International Partnership Program for Creative Research Teams, the Sino-German CDZ project CAP (GZ 1023) and Shanghai Municipal Natural Science Foundation (16ZR1409100).

References

1. Abdulla, P.A., Bouajjani, A., Holík, L., Kaati, L., Vojnar, T.: Computing simulations over tree automata: Efficient techniques for reducing tree automata. In: Tools and Algorithms for the Construction and Analysis of Systems: TACAS. LNCS, vol. 4963, pp. 93–108. Springer (2008)
2. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains: on checking language inclusion of nondeterministic finite (tree) automata. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems: TACAS. LNCS, vol. 6015, pp. 158–174. Springer, Berlin (2010)
3. Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press, Cambridge, Mass. (2008)
4. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: POPL'13: Principles of Programming Languages. pp. 457–468. ACM, New York (2013)

5. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers 35(8), 677–691 (1986), https://doi.org/10.1109/TC.1986.1676819
6. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification: CAV. LNCS, vol. 4144, pp. 17–30. Springer, Berlin (2006)
7. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Symposium on Foundations of Computer Science, Milwaukee: FOCS. pp. 453–462. IEEE Computer Society (1995)
8. Holík, L., Šimáček, J.: Optimizing an LTS-simulation algorithm. Computing and Informatics 29(6+), 1337–1348 (2010), http://www.cai.sk/ojs/index.php/cai/article/view/147
9. Hopcroft, J.E., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Tech. Rep. 71–114, Cornell University, Ithaca, NY (1971), http://hdl.handle.net/1813/5958
10. Hopcroft, J.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computations. pp. 189–196. Academic Press, New York (1971)
11. Lengál, O., Simácek, J., Vojnar, T.: VATA: A library for efficient manipulation of non-deterministic tree automata. In: Tools and Algorithms for the Construction and Analysis of Systems: TACAS. LNCS, vol. 7214, pp. 79–94. Springer (2012)
12. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
13. Pous, D.: Symbolic algorithms for language equivalence and Kleene algebra with tests. In: POPL'15: Principles of Programming Languages. pp. 357–368. ACM, New York (2015)
14. Ranzato, F., Tapparo, F.: A new efficient simulation equivalence algorithm. In: Symposium on Logic in Computer Science: LICS. pp. 171–180. IEEE Computer Society (2007)
15. Rutten, J.: Automata and coinduction (an exercise in coalgebra). In: Sangiorgi, D., de Simone, R. (eds.) CONCUR'98 Concurrency Theory. LNCS, vol. 1466, pp. 194–218. Springer, Berlin (1998)
16. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time. In: Proceedings of the 5th Annual ACM Symposium on Theory of Computing: STOC. pp. 1–9 (1973)
17. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Sutcliffe, G., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning: LPAR. LNCS, vol. 3835, pp. 396–411. Springer, Berlin (2005)
18. Valmari, A.: Bisimilarity minimization in $O(m \log n)$ time. In: Franceschinis, G., Wolf, K. (eds.) Applications and Theory of Petri Nets: PETRI NETS. LNCS, vol. 5606, pp. 123–142. Springer, Berlin (2009)
19. Veanes, M.: Applications of symbolic finite automata. In: Konstantinidis, S. (ed.) Implementation and Application of Automata: CIAA. LNCS, vol. 7982, pp. 16–23. Springer, Heidelberg (2013)
20. Whaley, J.: Javabdd. http://javabdd.sourceforge.net/, accessed June 13, 2017