Minimization of Non-Deterministic Automata with Large Alphabets ⋆

Parosh Aziz Abdulla, Johann Deneux, Lisa Kaati, and Marcus Nilsson

Dept. of Information Technology, P.O. Box 337, S-751 05 Uppsala, Sweden
{parosh,johannd,kaati,marcusn}@it.uu.se

Abstract. There has been several attempts over the years to solve the bisimulation minimization problem for finite automata. One of the most famous algorithms is the one suggested by Paige and Tarjan. The algorithm has a complexity of $\mathcal{O}(m \log n)$ where m is the number of edges and n is the number of states in the automaton. A bottleneck in the application of the algorithm is often the number of labels which may appear on the edges of the automaton. In this paper we adapt the Paige-Tarjan algorithm to the case where the labels are symbolically represented using Binary Decision Diagrams (BDDs). We show that our algorithm has an overall complexity of $\mathcal{O}(\ell \cdot m \cdot \log n)$ where ℓ is the size of the alphabet. This means that our algorithm will have the same worst case behavior as other algorithms. However, as shown by our prototype implementation, we get a vast improvement in performance due to the compact representation provided by the BDDs.

1 Introduction

Several algorithms have been proposed in the literature for solving the *coarsest refinement problem*: given a finite state automaton and an initial partitioning of the set of states, find the coarsest stable refinement of the given partitioning. The problem is equivalent to the minimization of non-deterministic automata modulo bisimulation, and consequently also gives an algorithm for minimizing deterministic automata modulo language equivalence. Minimization is relevant in many areas of computer science such as concurrency theory, formal verification, set theory, etc. For instance, in formal verification, several existing tools use minimization with respect to bisimulation in order to reduce the size of the state space to be analyzed [Bou98,CS96,FGK^{+}96]. Also, bisimulation is of particular interest in *regular model checking*. This is a framework which has recently been extensively studied for verification of systems with infinite state spaces (see e.g.[AJNS04]). The idea of regular model checking is to represent the state space of a system using regular languages. Most regular model checking algorithms rely heavily on efficient methods for checking bisimulation [AJNd03].

There has been several attempts over the years to solve the coarsest refinement problem. In [Hop71] Hopcroft presents an algorithm for minimization of a

⋆ This work was supported by the the Swedish Research Council (http://www.vr.se/)

deterministic automaton in $\mathcal{O}(n \log n)$ time. The algorithm relies on a "negative strategy": start from the initial partitioning and perform a number of iterations. During each iteration choose a block (equivalence class) B, and split all the blocks which violate the stability condition with respect to B. The main ingredient is the choice of the blocks which are used in the splitting (the so called "process the smaller half strategy"). The paper [PTB85] solves the problem for the special case of deterministic and unlabeled automata in linear time, using a "positive strategy": start with blocks which are singletons, and perform a number of iterations, where one or more blocks are merged during each iteration. Paige and Tarjan [PT87] generalized the algorithm of Hopcroft to the case of non-deterministic automata. The key idea is to employ counters which give the number of edges from states to blocks. This makes it possible to avoid partitioning with respect to large blocks. The algorithm runs in $\mathcal{O}(m \log n)$ time where m is the number of edges and n is the number of states in the automaton.

Many applications give rise to automata with large alphabets. For instance, transition systems generated by verification tools such as SPIN [Hol91] usually have very large alphabets [DPP04]. Also, the bottle-neck in applications of regular model checking is often the size of the alphabet in the automata which arise during the analysis [AJNd03,AJNS04]. Therefore, this paper adapts the Paige-Tarjan algorithm [PT87] to consider automata which have large alphabets. To deal with the size of the alphabet, we use a symbolic representation of labels on the edges of the automaton. More precisely, for states q and r, we characterize the set of symbols on which the automaton can move from q to r. This characterization is given through a *Binary Decision Diagram (BDD)* [Bry86]. The main task then is to adapt each step of the Paige-Tarjan algorithm which operates on explicit representation of the transition relation into a symbolic one which operates on BDDs. To achieve that, we use *Algebraic Decision Diagrams (ADDs)* [BFG$^+$93] to give a compact representation of the counters. Also, we show that each BDD or ADD operation can be performed in $\mathcal{O}(\ell)$ time where ℓ is the size of the alphabet. We show that this implies an overall complexity of $\mathcal{O}(\ell \cdot m \cdot \log n)$ of our algorithm. In other words, the algorithm will have the same worst case behavior as other algorithms. However, as shown by our prototype implementation, we often get a great improvement in performance due to the compact representation provided by the BDDs and ADDs.

Related Work The algorithm of [PT87] operates on an explicit representation of the (unlabeled) automaton. For automata with large alphabets, we report a big improvement compared to [PT87] using our prototype (see Section 7).

Fernandez [Fer89] presents an algorithm with complexity $\mathcal{O}(m \log n)$ in the case of labeled automata, the algorithm operates on an explicit representation of the automaton, where each edge is labeled with one symbol. In our case, an edge is labeled with a BDD which characterizes a set of symbols. Therefore the worst case complexity of our algorithm is the same as the one reported in [Fer89]. More precisely, we can replace each edge, labeled with a BDD $\mathcal{B}$, by a set of edges each carrying one symbol whose encoding satisfies $\mathcal{B}$.

Bouali and De Simone [BdS92] present a symbolic approach to the problem. The whole automaton and the computed blocks are encoded using BDDs. Such a full symbolic representation is in contrast with our approach where we only encode the alphabet symbolically, while we maintain an explicit representation of the set of states and of the blocks. The authors of [BdS92] do not perform a complexity analysis. However, they mention that they do not gain a drastic improvement compared to the classical algorithm. This indicates that, at least in the case of a large alphabet, it is more efficient to avoid a fully symbolic representation.

The work in [DPP04] combines the negative and positive approaches to bisimulation (described above) in the non-symbolic case. The authors also propose a symbolic algorithm for unlabeled automata, where each block is represented as a BDD. They show that their algorithm performs $\mathcal{O}(n)$ symbolic steps. No experimental results are reported for the symbolic algorithm.

In [Kla99] Klarlund presents an algorithm where the whole automaton (rather than only the alphabet) is represented symbolically. However, this algorithm can only be applied in the case of deterministic automata.

In [FV99] Fisler and Vardi compare symbolic versions of the Paige-Tarjan algorithm and algorithms described in the two papers [BFH90] and [LY92]. The latter two papers aim at adapting minimization to the context of the on-the-fly model checking. The paper argues both theoretically and based on experimental data that the Paige-Tarjan algorithm performs better than both.

Outline In the next two Sections we give preliminaries on automata, equivalence relations, BDDs, and ADDs. In Section 4 we describe our algorithm which consists of performing a number of iterations; and analyze its correctness and complexity. In Section 5 we describe the data structures we use in the implementation of the algorithm. Section 6 describes the steps performed during each iteration. We report on the results we obtain through running our prototype in Section 7. Finally, we give some conclusions and directions for future research in Section 8.

2 Preliminaries

In this section, we give some preliminaries of automata and equivalence relations. Throughout this paper, we will work with a *non-deterministic automaton*, NFA, which is a triple $\langle \mathtt{Q}, \Sigma, \Delta \rangle$ where

- $\mathtt{Q}$ is a finite set of states, with $|\mathtt{Q}| = n$
- Σ is a finite set of symbols, with $|\Sigma| = \ell$.
- Δ is a function $\Delta : \mathtt{Q} \times \mathtt{Q} \rightarrow 2^{\Sigma}$. An *edge* is a pair $\langle q, r \rangle$ such that $\Delta(q, r) \neq \emptyset$. We say that q and r are respectively the *source* and the *target* of the edge $\langle q, r \rangle$. We let m be the number of edges.

In other words, we consider an automaton with n states and m edges. Each edge is labeled with a set of symbols from an alphabet of size ℓ. Without loss of generality, we assume that each state is the source of at least one edge; which implies $m \geq n$. The automaton can change state from q to r on the symbols $\Delta(q, r)$. We write $q \xrightarrow{a} r$ to denote that $a \in \Delta(q, r)$ and $q \longrightarrow r$ to denote that $\Delta(q, r)$ is not empty. We use $(q \longrightarrow r)$ to denote the set $\{a \,:\, a \in \Delta(q, r)\}$, and use $Pre(r)$ to denote the set $\{q \,:\, (q \longrightarrow r) \neq \emptyset\}$. An element of $Pre(r)$ is said to be a *predecessor* of r. For a state $q \in \mathtt{Q}$ and a set $\mathtt{R} \subseteq \mathtt{Q}$, we use $(q \longrightarrow \mathtt{R})$ to denote the set $\bigcup_{r \in \mathtt{R}}(q \longrightarrow r)$, and $Pre(\mathtt{R})$ to denote the set $\bigcup_{r \in \mathtt{R}} Pre(r)$.

We consider equivalence relations on $\mathtt{Q}$. For an equivalence relation $\simeq$, we let $(\mathtt{Q}/\simeq)$ be the set of equivalence classes, henceforth called *blocks* of $\simeq$. For $q \in \mathtt{Q}$, $a \in \Sigma$, and $B \in (\mathtt{Q}/\simeq)$, we define $count(q)(B)(a)$ to be the size of the set $\{r \,:\, r \in B \text{ and } q \xrightarrow{a} r\}$.

For two equivalence relations $\simeq$ and $\simeq'$, we say that $\simeq$ is *coarser* than $\simeq'$ if $\simeq' \subseteq \simeq$. Alternatively, we say that $\simeq'$ is a refinement of $\simeq$. Notice that each block of $\simeq$ is the union of a number of blocks of $\simeq'$.

An equivalence relation $\simeq$ is *stable* with respect to an equivalence relation $\simeq'$, if whenever $q \simeq r$ then $(q \longrightarrow B) = (r \longrightarrow B)$ for each $B \in (\mathtt{Q}/\simeq')$. Equivalently, if $q \simeq r$ and $q \xrightarrow{a} q_1$ then there is an r_1 such that $r \xrightarrow{a} r_1$ and $q_1 \simeq' r_1$. In other words, equivalent states in $\simeq$ make moves on the same set of symbols to blocks in $\simeq'$. We say that $\simeq$ is *stable* if it is stable with respect to itself. The *coarsest refinement problem* is defined as follows

Instance An equivalence relation $\simeq_{init}$.

Task Find the coarsest stable refinement of $\simeq_{init}$.

3 BDDs and ADDs

In this section we recall some preliminaries of BDDs and ADDs, and introduce concepts which we will use in our algorithm.

We assume familiarity with *Binary Decision Diagrams (BDDs)* (see e.g. [Bry86,And98,Som99]) *Algebraic Decision Diagrams (ADDs)* [BFG$^+$93] are extensions of BDDs in the sense that the leaves of an ADD are labeled with natural numbers (rather than only 0 and 1 as is the case with BDDs).

BDDs We encode each symbol of the alphabet Σ by a finite binary word. Furthermore, we encode sets of symbols of Σ by Boolean expressions which are represented by BDDs. To do that, we use a set V of *BDD variables* where $|V| = \lceil \log_2(|\Sigma|) \rceil$ (recall that $\ell = |\Sigma|$). The variable v_i represents the i^{th} position in the encoding of a word (see the example below). A Boolean expression over V represents the set of symbols whose encodings satisfy the expression. Consequently, each BDD characterizes a set of symbols. In fact, each path from the root to a leaf of a BDD, represents a set of symbols, namely the set of symbols satisfying the path. Sometimes, we identify a BDD with the set of symbols it represents. For instance, given a BDD $\mathcal{B}$ and a symbol $a \in \Sigma$, we use $a \in \mathcal{B}$ to denote that a belongs to the set characterized by $\mathcal{B}$. Also, we use $\mathcal{B}(a)$ to denote the truth value of the formula $a \in \mathcal{B}$. In our algorithm, we use BDDs to represent the function Δ in the definition of an automaton (see Section 2). More precisely, for each $q, r \in \mathtt{Q}$, we represent $\Delta(q, r)$ by a BDD $\mathcal{B}$ such that $a \in \mathcal{B}$ iff $a \in \Delta(q, r)$. We write $\Delta(q, r) = \mathcal{B}$ to denote that the set of symbols in $\Delta(q, r)$ is characterized by $\mathcal{B}$.

Operations on BDDs The classical algorithm for computing a binary operation such as conjunction and disjunction on two BDDs is of time complexity $\mathcal{O}(2^k)$ where k is the number of variables which appear in the two input BDDs (see e.g. [Bry86,And98,Som99] for a description of the algorithm). In our case the value of k is bounded by $|V|$. Since $|V| = \lceil \log_2 (|\Sigma|) \rceil$ it follows that these operations can performed in $\mathcal{O}(\ell)$ time.

ADDs In a similar fashion to BDDs, we use an ADD to encode a multiset of symbols in Σ. Also in the case of ADDs, a path from the root to a leaf characterizes a set of symbols. For an ADD $\mathcal{A}$, the paths from the root to the leaf labeled i, characterizes the set of symbols which occur i times in the multiset represented by $\mathcal{A}$. We use $\mathcal{A}(a)$ to denote the number of occurrences a in the multiset represented by $\mathcal{A}$. By the *symbol set* of $\mathcal{A}$ we mean the set $\{a : \mathcal{A}(a) > 0\}$. We perform the following operations on ADDs:

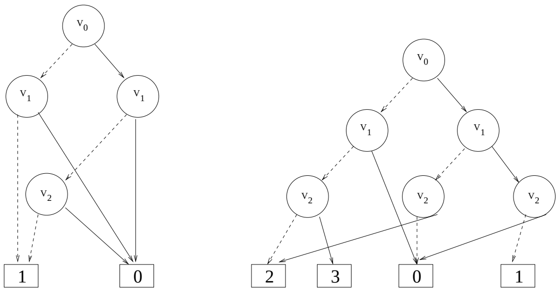
- *Addition*: $\mathcal{A}_1 + \mathcal{A}_2$ is an ADD $\mathcal{A}$ such that $\mathcal{A}(a) = \mathcal{A}_1(a) + \mathcal{A}_2(a)$ for each a. We define the *subtraction* $\mathcal{A}_1 - \mathcal{A}_2$ of two ADDs in a similar manner.
- *Comparison*: $\mathcal{A}_1 \oplus \mathcal{A}_2$ returns a BDD $\mathcal{B}$ such that $\mathcal{B}(a)$ is true iff $\mathcal{A}_1(a) = \mathcal{A}_2(a)$.
- *BDD conversion*: $\overline{\mathcal{A}}$ is a BDD which characterizes the symbol set of $\mathcal{A}$.

Using a similar reasoning to BDDs, all the above operations can be performed in time $\mathcal{O}(\ell)$. Sometimes, we mix BDDs and ADDs in the above operations. In such a case we interpret a BDD $\mathcal{B}$ as an ADD where $\mathcal{B}(a) = 1$ iff $a \in \mathcal{B}$. For instance, given an ADD $\mathcal{A}$ and a BDD $\mathcal{B}$ then $(\mathcal{A} + \mathcal{B})(a)$ is equal to $\mathcal{A}(a)$ in case $a \notin \mathcal{B}$, and is equal to $\mathcal{A}(a) + 1$ otherwise.

Example We consider the alphabet $\{a, b, c, d, e, f\}$. We use the encoding a: 000, b: 001, c: 010, d: 011, e: 100, f: 101. A dashed line in Figure 3 represents the false branch while the filled line represent the true branch of the BDD (ADD). Figure 3 a) shows a BDD characterizing the set $\{a, b, e\}$, while Figure 3 b) shows an ADD $\mathcal{A}$ with $\mathcal{A}(e) = 3$, $\mathcal{A}(a) = \mathcal{A}(f) = 2$, $\mathcal{A}(d) = 1$, and $\mathcal{A}(b) = \mathcal{A}(c) = 0$.

4 Algorithm

In this section, we describe our algorithm which consists of performing a number of iterations. Each step of the iteration is described in detail in Section 6. Given an initial equivalence $\simeq_{init}$, the iterations generate two sequences of equivalences of the forms $\simeq_0, \simeq_1, \simeq_2, \ldots, \simeq_t$ and $\cong_0, \cong_1, \cong_2, \ldots, \cong_t$ respectively. We define $\simeq_0$ to be $\simeq_{init}$ and $\cong_0$ to be $\mathtt{Q} \times \mathtt{Q}$. We derive $\simeq_{i+1}$ and $\cong_{i+1}$ from $\simeq_i$ and $\cong_i$ as



a) b)

Fig. 1. Example of a BDD and an ADD.

follows. Let $B_i \in (\mathtt{Q}/\simeq_i)$ and $S_i \in (\mathtt{Q}/\cong_i)$ be such that[1] $B_i \subset S_i$ and $|B_i| \leq \frac{|S_i|}{2}$. We define $\simeq_{i+1}$ such that $q \simeq_{i+1} r$ iff the following three conditions are satisfied:

- $q \simeq_i r$.
- $(q \longrightarrow B_i) \;=\; (r \longrightarrow B_i)$.
- $\begin{pmatrix} count(q)(B_i)(a) \\ = \\ count(q)(S_i)(a) \end{pmatrix}$ iff $\begin{pmatrix} count(r)(B_i)(a) \\ = \\ count(r)(S_i)(a) \end{pmatrix}$ for each $a \in \Sigma$.

We define $\cong_{i+1}$ such that $q \cong_{i+1} r$ iff the following two conditions are satisfied:

- $q \cong_i r$.
- $q \in B_i$ iff $r \in B_i$.

The iteration continues until we reach the termination point n at which we have $\simeq_t \;=\; \cong_t$. In the next Section, we describe the data structures which we use to represent the equivalences $\simeq_i$ and $\cong_i$; and in Section 6 we show how we can implement each step to maintain the above invariants.

Now, we proceed to prove some properties of the generated equivalences. The following lemma shows that $\simeq_i$ is a refinement of $\cong_i$. This implies that, up to the termination point, we will be able to pick $B_i \in (\mathtt{Q}/\simeq_i)$ and $S_i \in (\mathtt{Q}/\cong_i)$ such that $B_i \subset S_i$ and $|B_i| \leq \frac{|S_i|}{2}$.

Lemma 1. $\simeq_i \subseteq \cong_i$ *for all* i.

Next, we show partial correctness of the algorithm (Theorem 1). To do that, we show two auxiliary lemmas.

Lemma 2. $\simeq_i$ *is stable with respect to* $\cong_i$*, for all* i.

Lemma 3. *For any stable refinement* $\simeq'$ *of* $\simeq_{init}$*, it is the case that* $\simeq' \subseteq \simeq_i$ *for each* i.

[1] As we will show below (Lemma 1), $\simeq_i$ is a refinement of $\cong_i$ and therefore such B_i and S_i exist .

By definition we know that each $\simeq_i$ (and in particular $\simeq_t$) is a refinement of $\simeq_{init}$. From Lemma 2 and the fact that $\simeq_t = \cong_t$ we know that $\simeq_t$ is stable. This, together with Lemma 3 implies the following

Theorem 1. *$\simeq_t$ is the coarsest stable refinement of $\simeq_{init}$.*

Termination of the algorithm can be shown as follows: We know that, as long as the algorithm has not terminated we have $B_i \subset S_i$ and consequently $\cong_{i+1} \subset \cong_i$. By finiteness of $\mathtt{Q}$ it follows that after at most $t = |\mathtt{Q}| - 1$ steps we reach a point where there are no $B_t \in (\mathtt{Q}/\simeq_t)$ and $S_t \in (\mathtt{Q}/\cong_t)$ such that $B_t \subset S_t$ and $|B_t| \leq \frac{|S_t|}{2}$. This implies $\simeq_t = \cong_t$.

Theorem 2. *There is a $t \leq n - 1$ such that $\simeq_t = \cong_t$.*

Finally, we consider complexity of the algorithm.

Lemma 4. *For each $q \in \mathtt{Q}$ and $i < j$ if $q \in B_i \cap B_j$ then $|B_j| \leq \frac{|B_i|}{2}$.*

In Section 6 we will show that each iteration i can be performed in time

$$\mathcal{O}\left(\ell \cdot \left(|B_i| + \sum_{q \in B_i} |Pre(q)|\right)\right)$$

From this and Lemma 4, we get the following.

Theorem 3. *The algorithm has complexity $\mathcal{O}(\ell \cdot m \cdot \log n)$.*

5 Data Structures

In this section we describe the data structures used in the representation of the equivalences $\simeq_i$ and $\cong_i$ (see Section 4). Also, we use a number of auxiliary data structures which allow efficient implementation of each iteration in the algorithm.

Each state is represented by a record which we identify with the state itself. We maintain three lists of blocks:

- Q which corresponds to blocks in $\simeq_i$. Each state points to the block in Q containing it. Each block in Q is equipped with a natural number which indicates its size.
- X which corresponds to the blocks in $\cong_i$. A block of X is *simple* if contains a single block of Q, and is *compound* otherwise.
- C which is a sublist of X containing only the compound blocks in X.

The elements of the above lists are doubly linked. This allows deletion of elements in constant time. Each block in Q or X is represented by a record which we will identify with the block itself. Each block S in Q contains:

- a natural number which is equal to the size of the block.
- a pointer to a doubly linked list of its elements.

– a pointer to the block of X containing it.

Each block in X contains:

– a pointer to a doubly linked list of the blocks of Q contained in it.
– a pointer to a list of pairs: the first element of the pair is a state q such that q has an edge to a state in S; the second element is an ADD $\mathcal{A}_{(q,S)}$ which encodes $count(q)(S)$, i.e., $\mathcal{A}_{(q,S)}(a) = count(q)(S)(a)$ for each $a \in \Sigma$.

A state r has the following pointers to

– all pairs of the form $\langle q, \mathcal{B} \rangle$ where $\Delta(q,r) = \mathcal{B}$.
– the block in Q which it belongs to.

We shall also use a number of lists which will create and then destroy after each iteration step. These lists are implemented as hash tables, which means that searching for an element in the list can be assumed to take constant time.

6 Refinement Steps

In this section we describe how to implement each iteration of the algorithm of Section 4, so that an iteration takes $\mathcal{O}\left(\ell \cdot \left(|B_i| + \sum_{q \in B_i} |Pre(q)|\right)\right)$ time. An iteration consists of six steps as follows.

Step 1 This step chooses two blocks[2] B and S. Remove a block S from C. Examine the first two blocks in S. Let B be the smaller one. If they are equal in size, then B can be arbitrarily chosen to be anyone of them. This step can be performed in constant time.

Step 2 This step is to maintain the invariant that $q \cong_{i+1} r$ implies that $q \in B$ iff $r \in B$. Remove B from S and create a new block S' in X. The block S' is simple and contains B as its only block. If S is still compound, put it back into C. This step can be performed in constant time.

Step 3 Create a new list L, implemented as a hash table. Each element of L is a record containing a pointer to a state q and an ADD which we call $\mathcal{A}_{L(q)}$. The ADD $\mathcal{A}_{L(q)}$ characterizes $count(q)(B)$, i.e., it gives, for each $a \in \Sigma$, the number of edges from q which go to states in B and whose symbol sets include a. We create L by scanning the elements of B. For each $r \in B$ and each edge $\langle q, r \rangle$ we add q to L with $\mathcal{A}_{L(q)} = \mathcal{B}$ where $\mathcal{B} = \Delta(q,r)$. If q already is in L we modify the value of $\mathcal{A}_{L(q)}$ to be $\mathcal{A}_{L(q)} + \mathcal{B}$, i.e., we update $\mathcal{A}_{L(q)}$ according to the symbols in the set $\Delta(q,r)$.

Since L is a hash table, searching for a state q in L takes constant time. Performing addition on ADDs takes $\mathcal{O}(\ell)$ time (Section 2).

[2] These blocks correspond to B_i and S_i chosen during the i^{th} iteration (Section 4).

Step 4 We partition each block of Q with respect to B. This will maintain the invariant that $q \simeq_i r$ implies $(q \longrightarrow B) = (r \longrightarrow B)$. We create a new block $D_{\mathcal{B}}$ for each block D of Q and BDD $\mathcal{B}$ such that there is a state $q \in B$ with q in L and $\overline{\mathcal{A}_{L(q)}} = \mathcal{B}$. Intuitively, the block $D_{\mathcal{B}}$ will contain all states which originally belonged to D and which have edges to B on the same set of symbols (namely the set of symbols characterized by $\mathcal{B}$). To perform this operation, each block D in Q will maintain a list L_D, implemented as a hash table. Each element of L_D is a pair, where the first element is a BDD and the second element is a pointer to a block. We traverse the list L created in step 3 above. For each state q in L, we consider the block D in Q to which q currently belongs. We remove q from D. We find the entry in L_D with a BDD equal to $\overline{\mathcal{A}_{L(q)}}$. We insert q in the corresponding block.

In the second phase of step 4, we add the newly created blocks to Q. If a block D has become empty we remove it from Q. If the block in X which contains B or one of the newly created blocks has become compound, we insert it in C.

Computing $\overline{\mathcal{A}_{L(q)}}$ takes time $\mathcal{O}(\ell)$ (see Section 3). Since L_D is a hash table, searching the table takes constant time. Removing q from D and inserting it in the new block takes constant time. Moving and checking emptiness of block takes constant time.

Step 5 We partition each block of Q with respect to $S - B$. This will keep the invariant that $q \simeq_i r$ implies

$$\begin{pmatrix} \mathit{count}(q)(B_i)(a) \\ = \\ \mathit{count}(q)(S_i)(a) \end{pmatrix} \text{ iff } \begin{pmatrix} \mathit{count}(r)(B_i)(a) \\ = \\ \mathit{count}(r)(S_i)(a) \end{pmatrix}$$

This step is similar to Step 4 above. The only difference is the manner in which we insert a state in the list L_D. When considering a state q in L, belonging to (say) block D, we compute $\mathcal{B} = \mathcal{A}_{L(q)} \oplus \mathcal{A}_{S(q)}$. The position of q in L_D will be determined by the BDD $\mathcal{B}$ (rather by the BDD $\overline{\mathcal{A}_{L(q)}}$ as was the case in Step 4). Intuitively, the BDD $\mathcal{B}$ characterizes the set of symbols through which the state q moves to $S - B$. This means that, states which will end up in the same block will move to $S - B$ on the same set of symbols, and hence the above mentioned invariant will be maintained.

Step 6 Since B was removed from S, the value of $\mathit{count}(q)(S)(a)$ may have been reduced (in case q has an edge to B labeled with a). This step updates the value of $\mathit{count}(q)(S)(a)$ accordingly. Recall that L contains all states which have edges to B. We scan the list L, and for each state q, we replace the current value $\mathcal{A}$ of $\mathcal{A}_{S(q)}$ by $\mathcal{A} - \mathcal{A}_{L(q)}$ (takes $\mathcal{O}(\ell)$). If $\mathcal{A}$ becomes empty we discard the pair q and its associated ADD $\mathcal{A}_{S(q)}$ from the list pointed to by S. Finally, we make B point to the list L.

Observe that the time spent on an iteration is $\mathcal{O}(\ell)$ per scanned edge and state of B, which gives a total time of $\mathcal{O}\left(\ell \cdot \left(|B_i| + \sum_{q \in B_i} |\mathit{Pre}(q)|\right)\right)$.

7 Experiments

There is no official set of benchmarks for testing algorithms that compute bisimulation equivalence [DPP04]. Therefore, we have implemented a procedure for randomly generating non-deterministic automata. In the procedure, we can change a number of parameters which decide the shape of the generated automata. Such parameters include the number of states, the size of the alphabet, the density of edges between two states, the probability that a certain symbol is included in the symbol set between two states, and the size of such a set.

Symbols in alphabet	2^2	2^4	2^5	2^7	2^9	2^{10}	2^{11}	2^{12}	2^{40}	2^{80}	2^{100}	2^{120}
Non-symbolic	3.99	4.55	4.81	7.18	32.99	120.7	557.74	1955	–	–	–	–
Symbolic	0.05	0.08	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.11	0.12

Table 1. Comparing the symbolic and the non-symbolic versions of the algorithm on automata with 20 states. The execution time is measured in seconds. Larger numbers of states give similar behaviours.

In Table 1 we compare the execution times of our implementation of the algorithm and a non-symbolic version of the Paige-Tarjan algorithm. To make the comparison meaningful we have implemented both versions of the algorithm in the same code, using the same data structures and the same procedures. As evident from the table, the symbolic version is almost insensitive to the size of the alphabet, while the non-symbolic version exhibits an exponential increase in time until we reach a point where it takes too long time (more than 24 hours). The above experiments are conducted with the number of states being equal to 20. We get a similar behaviour pattern when increasing the number of states. We have tested our prototype on automata with up to 200 states.

We have also compared our implementation with The Concurrency WorkBench (CWB) [CPS93] and The Concurrency WorkBench of The New Century (CWB-NC) [CS96]. The results are presented in Table 2. CWB uses minimization techniques based on the Kanellakis and Smolka algorithm [KS90], while CWB-NC uses the Paige-Tarjan algorithm. Both tools show similar behaviour to the non-symbolic version of our code.

Symbols in alphabet	2^2	2^5	2^{10}	2^{11}	2^{12}	2^{25}	2^{40}	2^{50}	2^{80}	2^{100}	2^{115}
Symbolic	4.69	6.05	6.52	6.60	6.60	6.69	6.84	7.66	8.43	10.15	12.46
CWB	0.13	0.68	10.60	18.50	28.44	–	–	–	–	–	–
CWB-NC	0.31	0.32	–	–	–	–	–	–	–	–	–

Table 2. The execution time for our implementation of the algorithm and minimization in CWB and CWB-NC. The automata have 150 states and 250 transitions. Execution time is measured in seconds.

In Figure 2 we keep the size of alphabet intact while we increase the probability that a symbol is included in the symbol set of an edge. We observe that while our algorithm copes well with large alphabets, its efficiency decreases with symbol density.

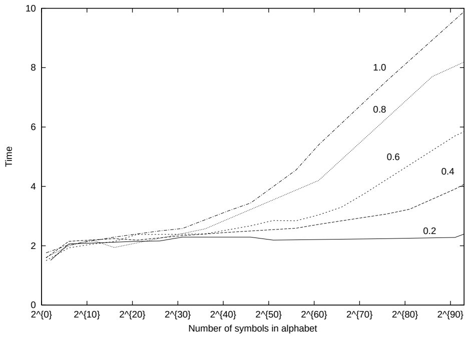


Fig. 2. Increasing the symbol density, while keeping the size of the alphabet fixed. The automata have 150 states and 250 transitions.

8 Conclusions and Future Work

We have presented a version of the Paige-Tarjan algorithm where the edge relation for labeled automata is represented symbolically using BDDs. For automata with large alphabets, our experiments indicate that the algorithm behaves better than algorithms which operate on an explicit representation of the automaton.

One direction for future research, is to consider Boolean encodings of the alphabet which are not canonical (as is the case with BDDs) and then use SAT solvers to perform the necessary operations on the symbolic encoding. It is well-known that SAT solvers outperform BDDs in certain applications, and it could be interesting to find out whether this is the case for minimization of automata. Also, we intend to consider similar algorithms for checking *simulation* relations. This is relevant, for instance, in the context of regular model checking, where several classes of acceleration techniques rely on computing simulations [AJNS04].

9 Example

Consider the transition system shown in Figure 9. Compute the list L_S where each element of L_S is a record containing a pointer to a state q and an ADD which we call $L_S(q)$. In this example we represent $L_S(q)$ with a list of pairs where each pair contains a symbol and the leaf value for the path of the encoding of the symbol.

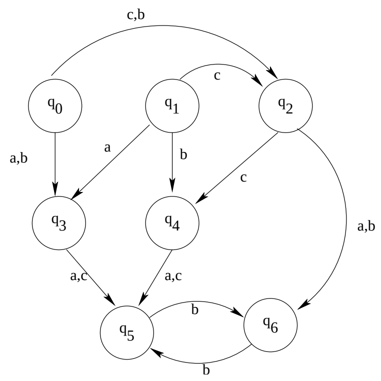


Fig. 3. Transition system before minimization.

$L_S(q_0)$= $((a,1)$,$(b,2)$,$(c,1))$
$L_S(q_1)$= $((a,1)$,$(b,1)$,$(c,1))$
$L_S(q_2)$= $((a,1)$,$(b,1)$,$(c,1))$
$L_S(q_3)$= $((a,1)$,$(c,1))$
$L_S(q_4)$= $((a,1)$,$(c,1))$
$L_S(q_5)$= $((b,1))$
$L_S(q_6)$= $((b,1))$

Create an inital partition $Q = (\{q_0, q_1, q_2\}, \{q_3, q_4\}, \{q_5, q_6\})$. Create a compound block S containing all blocks of Q and add S to C and to X.

First iteration

Remove the compund block $S = \{\{q_0, q_1, q_2\}, \{q_3, q_4\}, \{q_5, q_6\}\}$ from C.
Examine the first two blocks of S and choose the smallest one $B = \{q_3, q_4\}$.

Compute the list L_B.

$L_B(q_0) = ((a, 1),\ (b, 1))$
$L_B(q_1) = ((a, 1),\ (b, 1))$
$L_B(q_2) = ((c, 1))$

Remove q_0, q_1 and q_2 from the block they are contained in. Insert q_0 and q_1 in a block determined by the symbols $\{a, b\}$. Insert q_2 in a block determined by the symbol $\{c\}$. Since the block that contained q_0, q_1 and q_2 is empty remove it from Q.
$Q = (\{q_0, q_1\}, \{q_2\}, \{q_3, q_4\}, \{q_5, q_6\})$
$X = (\{\{q_0, q_1\}, \{q_2\}, \{q_5, q_6\}\}, \{q_3, q_4\})$
$C = (\{\{q_0, q_1\}, \{q_2\}, \{q_5, q_6\}\})$

Refine with respect to $S - B$

For all q in L_B, determine the set of symbols where the pairs in $L_B(q)$ and $L_S(q)$ are equal. Move each state to a new block depending on the set of symbols. Move q_0 to a block determined by $\{a\}$, move q_1 to a block determined by $\{a, b\}$. Since the old blocks containing the states are empty, delete them.
$Q = (\{q_0\}, \{q_1\}, \{q_2\}, \{q_3, q_4\}, \{q_5, q_6\})$ $X = (\{\{q_0\}, \{q_1\}, \{q_2\}, \{q_5, q_6\}\}, \{q_3, q_4\})$
$C = (\{\{q_0\}, \{q_1\}, \{q_2\}, \{q_5, q_6\}\})$

Update the list L_S to L_{S-B}

$L_S(q_0)$= $((b, 1)$,$(c, 1))$
$L_S(q_1)$= $((c, 1))$
$L_S(q_2)$= $((a, 1)$,$(b, 1))$
$L_S(q_3)$= $((a, 1)$,$(c, 1))$
$L_S(q_4)$= $((a, 1)$,$(c, 1))$
$L_S(q_5)$= $((b, 1))$
$L_S(q_6)$= $((b, 1))$

Second iteration

Refine with respect to $B = \{q_0\}$, $S = \{\{q_0\}, \{q_1\}, \{q_2\}, \{q_5, q_6\}\}$
S is still compound after removing B and put back in C. The partition is not modifided and
$Q = (\{q_0\}, \{q_1\}, \{q_2\}, \{q_3, q_4\}, \{q_5, q_6\})$
$X = (\{q_0\}, \{\{q_1\}, \{q_2\}, \{q_5, q_6\}\}, \{q_3, q_4\})$
$C = (\{\{q_1\}, \{q_2\}, \{q_5, q_6\}\})$

Third iteration

Refine with respect to $B = \{q_1\}$, $S = \{\{q_1\}, \{q_2\}, \{q_5, q_6\}\}$
S is still compound after removing B and put back in C. The partition is not modifided and
$Q = (\{q_0\}, \{q_1\}, \{q_2\}, \{q_3, q_4\}, \{q_5, q_6\})$
$X = (\{q_0\}, \{q_1\}, \{\{q_2\}, \{q_5, q_6\}\}, \{q_3, q_4\})$
$C = (\{\{q_2\}, \{q_5, q_6\}\})$

Forth iteration

Refine with respect to $B = \{q_2\}$, $S = \{\{q_2\}, \{q_5, q_6\}\}$
S is not compound after removing B and therefore not added to C.
$L_B(q_0)$ $((b, 1)$,$(c, 1))$
$L_B(q_1)$ $((c, 1))$
Move q_0 and q_1 since their original blocks are empty, delete the blocks.
Refine with respect to $S - B$

Move q_0 and q_1 to new blocks determined by $\{b, c\}$ and $\{c\}$.

$Q = (\{q_0\}, \{q_1\}, \{q_2\}, \{q_3, q_4\}, \{q_5, q_6\})$
$X = (\{q_0\}, \{q_1\}, \{q_2\}, \{q_5, q_6\}, \{q_3, q_4\})$
$C = (\ )$

Since C is empty, we have reached a stable partition. The resulting automata is shown in Figure 4.

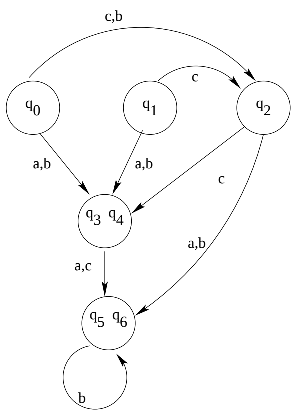


Fig. 4. Transition system after minimization.

References

[ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Proc. 11^{th} Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.

[AJNd03] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Algorithmic improvements in regular model checking. In *Proc. 15^{th} Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 236–248, 2003.

[AJNS04] P.A. Abdulla, B. Jonsson, Marcus Nilsson, and M. Saksena. A survey of regular model checking. In *Proc. CONCUR 2004, 15^{th} Int. Conf. on Concurrency Theory*, pages 348–360, 2004.

[And98] H. R. Andersen. An introduction to binary decision diagrams. Technical Report DK-2800, Department of Information Technology, Technical University of Denmark, 1998.

[BdS92] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Proc. Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 96–108, 1992.

[BFG+93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 188–191. IEEE Computer Society Press, 1993.

[BFH90] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. Minimal model generation, May 1990. Manuscript.

[BLW03] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large. In *Proc. 15^{th} Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235, 2003.

[Bou98] Amar Bouali. Xeve, an esterel verification environment. In *Proc. 10^{th} Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 500–504. Springer Verlag, 1998.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Trans. on Programming Languages and Systems*, 15(1), Jan. 1993.

[CS96] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In *Proc. 8^{th} Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397. Springer Verlag, 1996.

[DPP04] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004.

[Fer89] J-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Sci. Comput. Program.*, 13(1):219–236, 1989.

[FGK+96] J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In *CAV'96*. LNCS 1102, 1996.

[FV99] Kathi Fisler and Moshe Y. Vardi. Bisimulation and model checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 338–341, 1999.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Hop71] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[Kla99] N. Klarlund. An $n \log n$ algorithm for online bdd refinement. *J. Algorithms*, 32(2):133–154, 1999.

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.

[KS90] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[LY92] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proc. 24^{th} ACM Symp. on Theory of Computing*, 1992.

[PT87] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[PTB85] R. Paige, R. Tarjan, and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theoretical Computer Science*, 40:67–84, 1985.

[Som99] F. Somenzi. Binary decision diagrams, 1999.