Bisimilarity Minimization in $O(m \log n)$ Time

Antti Valmari

Tampere University of Technology, Department of Software Systems
PO Box 553, FI-33101 Tampere, Finland
Antti.Valmari@tut.fi

Abstract. A new algorithm for bisimilarity minimization of labelled directed graphs is presented. Its time consumption is $O(m \log n)$, where n is the number of states and m is the number of transitions. Unlike earlier algorithms, it meets this bound even if the number of different labels of transitions is not fixed. It is based on refining a partition on states with respect to the labelled transitions. A splitter is a pair consisting of a set in the partition and a label. Earlier algorithms consume lots of time in scanning splitters that have no corresponding relevant transitions. The new algorithm avoids this by maintaining the sets of the corresponding transitions. To facilitate this, a refinable partition data structure with amortized constant time operations is introduced. Detailed pseudocode and correctness proof are presented, as well as some measurements.

Keywords: Analysis of reachability graphs, verification of systems.

1 Introduction

Bisimilarity (also known as *strong bisimilarity*) has an important role in the analysis and verification of the behaviours of concurrent systems. For instance, two finite systems satisfy the same CTL or CTL* formulae if and only if they are bisimilar [2], and two process-algebraic systems are observationally equivalent in the sense of [10] if and only if their so-called saturated versions are bisimilar [8]. Bisimilarity abstracts away precisely the part of information stored by the state of the system that does not have any effect on subsequent behaviour of the system (this only holds in the absence of the notion of "invisible action"). Unlike isomorphism, bisimilarity can unite different states. These properties make it also a useful mathematical tool for dealing with other concepts, like symmetries.

Bisimilarity is an equivalence relation for comparing the vertices of a directed graph whose vertices or edges or both have labels. We will use the words *state* and *transition* instead of "vertex" and "edge" from now on, because the vertices represent states of the concurrent system and edges represent (semantic) transitions between states. Absence of state labels is equivalent to every state having the same label, and similarly with transitions. So we may simplify the discussion by assuming that both states and transitions have labels.

If two states s_1 and s_2 are bisimilar, then they have the same label, and they can simulate each other's transitions in the following sense. If s_1 can make a

G. Franceschinis and K. Wolf (Eds.): PETRI NETS 2009, LNCS 5606, pp. 123–142, 2009.

transition with label a to some state s'_1, then there is a state s'_2 that is bisimilar with s'_1 such that s_2 can make an a-transition to s'_2. In the same fashion, whatever transition s_2 can make, s_1 can simulate it.

The description of bisimilarity given above cannot be used as a definition, because it is circular: to explain what it means that s_1 and s_2 are bisimilar, it appeals to the bisimilarity of s'_1 and s'_2. Many different relations satisfy the description. Therefore, the precise definition uses the auxiliary concept of *bisimulation*. A binary relation between states is a bisimulation if and only if it meets the description of bisimilarity given above. The empty relation and the identity relation are trivially bisimulations. It is not difficult to check that the union of any set of bisimulations over the same graph is a bisimulation. It is the largest bisimulation, and it is an equivalence. It is the bisimilarity relation.

Bisimilarity can be applied to reachability graphs of Petri nets. The label of a semantic transition could be the name of the Petri net transition whose occurrence created the semantic transition, or it could be something more abstract, like a common name of several Petri net transitions. The label of a state could be a collection of Boolean variables that describe some properties of the state, such as there is a token in a certain subset of places. Using the marking as the label of the state as such would make bisimilarity useless, because then each state would be bisimilar only with itself.

Two systems can be compared by taking their disjoint union and checking that their initial states are bisimilar in it. If the systems have several initial states, then each initial state of each system must have a bisimilar initial state in the other system. For each system, there is a unique smallest system that is bisimilar to it. It can be found by removing the states that are not reachable from any initial state, dividing the set of remaining states to equivalence classes according to bisimilarity, and fusing each equivalence class into a single state. The label and output transitions of the fused state are copied from one of its states, where the end state of the copy transition is the fused state that contains the end state of the copied transition. Thanks to the properties of bisimilarity, it does not matter which state in the fused state is used in the copying. The fused state is made an initial state if and only if it contains an original initial state.

In this paper we concentrate on finding the bisimilarity equivalence classes. We present a new algorithm whose asymptotic time consumption is better than that of earlier algorithms. The consumption does not depend on the number of different labels of transitions. This is nice when there are many different labels, like "send$\langle x \rangle$" and "receive$\langle x \rangle$" where x may assume many different values.

In Section 2 we describe the problem in more detail and discuss earlier work. The new algorithm uses three copies of a special refinable partition data structure that is described in Section 3. The new algorithm is presented in Section 4. Detailed proofs of its correctness and performance are deferred to Section 5, because they cannot be followed before having seen the algorithm as a whole. Some measurements that were made with a prototype implementation are shown in Section 6. Section 7 contains the conclusions.

2 Background

The bisimilarity minimization problem is the following. A *partition* $\mathcal{S}$ of a set S is a collection of non-empty, mutually disjoint sets $S_1, S_2, \ldots, S_k$ such that $S_1 \cup S_2 \cup \cdots \cup S_k = S$. A *refinement* of $\mathcal{S}$ is any partition $\{Z_1, Z_2, \ldots, Z_h\}$ such that $Z_1 \cup \cdots \cup Z_h = S$ and each Z_i is a subset of some S_j. Given is a labelled directed graph (S, L, Δ), where $\Delta \subseteq S \times L \times S$, together with an initial partition $\mathcal{I}$ of S. By $s \mathrel{-a\!\!\rightarrow} s'$ we mean that $(s, a, s') \in \Delta$. A partition $\mathcal{S}$ is *compatible* with Δ, if and only if for every $S \in \mathcal{S}$, $S' \in \mathcal{S}$, $s_1 \in S$, $s_2 \in S$, $s'_1 \in S'$, and $a \in L$ such that $s_1 \mathrel{-a\!\!\rightarrow} s'_1$, there is an $s'_2 \in S'$ such that $s_2 \mathrel{-a\!\!\rightarrow} s'_2$. It has been proven that there is a unique partition that is a refinement of $\mathcal{I}$, compatible with Δ, and contains as few sets as possible. The task is to find that partition.

In this formulation of the problem, the state labels of Section 1 have been replaced by the initial partition. This is not an important modification, because the only thing done with state labels in the definition of bisimilarity is checking whether the labels of two states are the same or not. The initial partition can be constructed quickly enough by sorting the states according to their labels.

We denote the numbers of states, transitions, and labels by $n = |S|$, $m = |\Delta|$, and $\alpha = |L|$. The number of sets in the initial partition is denoted with k. To avoid getting into troublesome technicalities with complexity formulae, we assume that $n \leq 2m$. This is not a significant restriction, because to violate it the directed graph must be rather pathological. If every state of a graph has at least one input or output transition, then it meets the assumption.

In some applications of the bisimilarity minimization problem, only those states are relevant that are reachable from an initial state. Furthermore, it may be that a state is irrelevant also if no final state is reachable from it and it is not initial. It is well known that irrelevant states can be removed in $O(m + n)$ time by basic graph traversal algorithms.

The bisimilarity minimization problem can be solved by starting with the initial partition, and splitting sets of the partition as long as necessary. In this context, the sets of the partition are traditionally called *blocks*. If s_1 and s_2 are in the same block B and $s_1 \mathrel{-a\!\!\rightarrow} s'_1$, where s'_1 is in block B', but there is no $s'_2 \in B'$ such that $s_2 \mathrel{-a\!\!\rightarrow} s'_2$, then B must be split so that s_1 and s_2 go into different halves. This splitting may make further splitting necessary. There may be s''_1, s''_2, s'' in some block B'', and b such that $s''_1 \mathrel{-b\!\!\rightarrow} s_1$, $s''_2 \mathrel{-b\!\!\rightarrow} s_2$, $s'' \mathrel{-b\!\!\rightarrow} s_1$, $s'' \mathrel{-b\!\!\rightarrow} s_2$, and they do not have other b-labelled output transitions. Then the separation of s_1 and s_2 into different blocks makes it necessary to separate s'', s''_1, and s''_2 into three different blocks. We will call this *three-way splitting*.

Hopcroft's famous deterministic finite automaton (DFA) minimization algorithm [7] contains an early sub-quadratic algorithm for an important subproblem of the bisimilarity minimization problem. DFA minimization consists of removing irrelevant states and solving a restricted version of the bisimilarity minimization problem. In this version, $k \leq 2$ (the final states and the other states), and the graph is *deterministic*, that is, for each s and a, there is at most one s' such that $s \mathrel{-a\!\!\rightarrow} s'$. Thus $m \leq \alpha n$, while in general $m \leq \alpha n^2$.

Hopcroft's algorithm runs in $O(\alpha n \log n)$ time (see [5] or [9]). It uses *splitters*. Precise meaning varies in the literature, but let us define a splitter as a block–label pair (B, a). It is used for splitting each block according to whether its states do or do not have an outgoing a-labelled transition whose end state is in B. In Hopcroft's algorithm, these transitions are traversed backwards, and their start states are moved to tentative new blocks. This is much better than scanning a block and checking which of its states have an a-transition to some state in B, because the latter approach may involve costly scanning of numerous states that lack such a transition.

Because DFAs are deterministic, three-way splitting is never necessary. As a consequence, if (a, B) has been used for splitting and then B splits to B_1 and B_2, it is not necessary to use both (a, B_1) and (a, B_2) for further splitting. Hopcroft's algorithm chooses the (in some sense) "smaller" of them. This guarantees that each time when a transition is used for splitting, the size of some set is at most half of its size in the previous time. Therefore, the same transition is used at most a logarithmic number of times. This made it possible to reach $O(\alpha n \log n)$ time complexity instead of $O(\alpha n^2)$.

An $O(m \log n)$ time algorithm for the sub-problem of bisimilarity minimization where $\alpha = 1$ (or, equivalently, transitions have no labels) was presented by Paige and Tarjan [11]. Now the graph needs not be deterministic and three-way splitting is necessary. To facilitate the use of the "half the size" trick, the algorithm uses *compound blocks*. A compound block is a collection of blocks that once constituted together a single block that has been used for splitting. The use of the largest block in the compound block may be avoided in further splitting. A counter-based technique was used to find out whether the start state of the current transition has an output transition also to elsewhere in the current compound block, in addition to the current block.

Generalizing the Paige–Tarjan algorithm to $\alpha \geq 1$ while maintaining its good complexity is not trivial. The algorithm in [4, p. 229] does not meet the challenge. The paper does not give its time complexity, but there is certainly an αn term, because the algorithm scans the set of labels for each block that it uses in a splitter. Furthermore, it relies on the restrictive assumption that there is a global upper bound to the number of output transitions of any state and label (p. 228).

In [3, p. 242], the label of each transition is represented by adding a new state in the middle of the transition and initially partitioning these states according to the labels they represent. Then the Paige–Tarjan algorithm can be used as such. The time complexity is $O(m \log m)$, which is slightly worse than $O(m \log n)$. The approach also consumes more memory by a constant factor than the algorithm presented in Section 4.

When α is not fixed, an $O(m \log n)$ algorithm even for the deterministic case had not been published until 2008 [12]. The problem has been the time spent in scanning empty splitters, that is, splitters whose block does not have incoming transitions with the label. Nothing needs to be done for them, but they are so numerous that simply looking at each of them separately takes too much time. In [12], this is avoided by maintaining the non-empty sets of transitions that

correspond to splitters, and using these sets instead of the splitters to organize the work. The sets constitute a partition of the set of transitions that can be maintained similarly to the blocks. Therefore, [12] presents a refinable partition data structure, one instance of which is used for the blocks and another for the transitions.

In this paper we apply the above idea to the Paige–Tarjan algorithm, to design an $O(m \log n)$ algorithm for the bisimilarity minimization problem. To implement three-way splitting of blocks and to mimic the compound blocks, new features are added to the refinable partition data structure. The counter-based technique in [11] is replaced by a third instance of the structure.

3 A Refinable Partition Data Structure

In this section, a refinable partition data structure is presented. It is an extension of the structure presented in [12]. It maintains a partition $\{A_1, A_2, \ldots, A_{sets}\}$ of the set $\{1, 2, \ldots, items\}$ for some integer constant *items*. Later in this paper three instances of it will be used, one where the elements are states and $items = n$, and two where the elements are transitions and $items = m$.

The partition is refinable, meaning that it is possible to replace any A_i by two or three sets, provided that they are non-empty and disjoint and their union is A_i. This operation is called *splitting*. One part inherits the index i from A_i, while the other parts each get a brand new index.

To indicate which elements go into which subset of A_i, elements are *1-marked* and perhaps also *2-marked* before splitting A_i. There is one splitting operation that divides A_i to its 1-marked states and the remaining states, so that the set of 1-marked states gets a new index, and the remaining states retain the old index i. If either subset would be empty, then the operation does not divide A_i. There is also another splitting operation that does the same for 2-marked states. Each operation returns the index of the new subset, or zero to indicate that A_i did not split. The reason for having these two splitting operations instead of one three-way splitting operation is that returning the index or zero would be clumsier with the three-way operation. To discuss marking and splitting, let A_i^1 and A_i^2 denote the sets of 1-marked and 2-marked elements of A_i, respectively. Initially all elements of A_i are unmarked, that is, both A_i^1 and A_i^2 are empty.

One of the three instances of the data structure uses *bunches*. The bunches are a partition of the set $\{A_1, \ldots, A_{sets}\}$. Therefore, a bunch U_u is a set of sets $\{A_{u_1}, A_{u_2}, \ldots, A_{u_g}\}$. A bunch cannot contain just any subset of $\{A_1, \ldots, A_{sets}\}$. Instead, a bunch starts its life containing precisely one set A_i. When any set in a bunch is split, the bunch inherits all of its parts. There is also an operation that extracts a set from a non-singleton bunch and makes a new bunch of it.

Furthermore, there are services for scanning a set or a bunch, and for other duties. All services provided by the data structure are listed below.

Size(s) Returns the number of elements in the set with index s, that is, $|A_s|$.
Set(e) Returns the index of the set that element e belongs to, that is, the s such that $e \in A_s$.

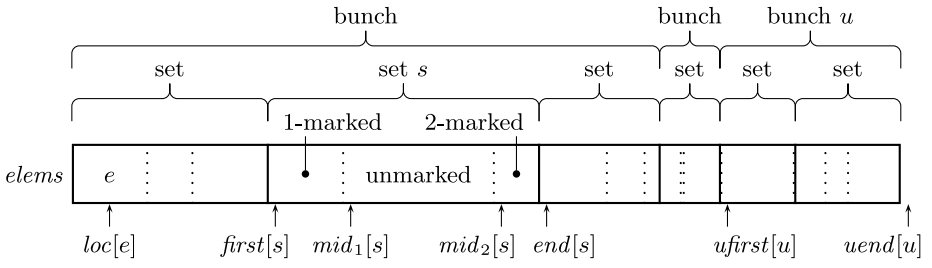

Fig. 1. Illustrating the refinable partition data structure

*Mark*1(e) **and** *Mark*2(e) Mark the element e for splitting the set A_s that contains e. *Mark*1 adds e to A_s^1 and *Mark*2 to A_s^2, unless it is already in $A_s^1 \cup A_s^2$.

*Split*1(s) **and** *Split*2(s) If $A_s^1 = \emptyset$ or $A_s = A_s^1 \cup A_s^2$, then *Split*1(s) unmarks all 1-marked elements in A_s and returns zero. Otherwise, it updates $A_s := A_s - A_s^1$, adds a new set $A_z := A_s^1$ to the partition, puts it into the same bunch with A_s, and returns z. In the end, $A_z^1 = A_z^2 = A_s^1 = \emptyset$, but A_s^2 has not changed. *Split*2 works similarly on 2-marked elements.

No_marks(s) Returns True if and only if $A_s^1 = A_s^2 = \emptyset$.

First(s) **and** *Next*(e) The elements of A_s can be scanned by first executing $e := First(s)$ and then **while** $e \neq 0$ **do** $e := Next(e)$. Each element will be returned exactly once, but the order in which they are returned is unspecified. While scanning a set, *Mark*1, *Mark*2, *Split*1, and *Split*2 must not be executed. These operations are provided to promote data abstraction. Instead of using them, it would be slightly more efficient to scan A_s directly from the arrays that implement the data structure.

Bunch(s) Returns the index of the bunch that set s belongs to.

Bunch_first(u) **and** *Bunch_next*(e) Let $U_u = \{A_{u_1}, A_{u_2}, \ldots, A_{u_g}\}$ be a bunch. With these operations, the elements of $A_{u_1} \cup A_{u_2} \cup \cdots \cup A_{u_g}$ can be scanned, similarly to how *First*(s) and *Next*(e) scan a set in the partition.

Has_many(u) Returns False if and only if bunch U_u consists of precisely one set.

Extract_set(u) Let $U_u = \{A_{u_1}, A_{u_2}, \ldots, A_{u_g}\}$ be a bunch. If $g = 1$, then this operation returns zero without changing anything. Otherwise, it selects some i, introduces a new bunch $\{A_{u_i}\}$, removes A_{u_i} from U_u, and returns u_i. The chosen i is such that if U_u has a unique biggest set, then it is not A_{u_i}.

Left_neighbour(e) **and** *Right_neighbour*(e) If the partition consists of one set, then both of these return zero. Otherwise, at least one of them returns an element that is not currently in the same set as e, but was in the same set until the most recent splitting of the set. The other one may return zero or an element. The motivation for these operations is explained in Section 4.

The implementation of the services is illustrated in Figure 1, and shown in Figures 2 and 3. We will soon discuss the implementation of the most complicated operations. The variables *sets* and *bunches* tell the numbers of sets and bunches. The implementation uses them and the following arrays:

Size(s)
return $end[s] - first[s]$

Set(e)
return $sidx[e]$

First(s)
return $elems[first[s]]$ /* Certainly exists, because the A_i are non-empty */

Next(e)
if $loc[e] + 1 \geq end[sidx[e]]$ **then return** 0 **else return** $elems[loc[e] + 1]$

*Mark*1(e)
$s := sidx[e]$; $\ell := loc[e]$; $m := mid_1[s]$
if $m \leq \ell < mid_2[s]$ **then**
  $mid_1[s] := m + 1$
  $elems[\ell] := elems[m]$; $loc[elems[\ell]] := \ell$; $elems[m] := e$; $loc[e] := m$

*Mark*2(e)
$s := sidx[e]$; $\ell := loc[e]$; $m := mid_2[s] - 1$
if $mid_1[s] \leq \ell \leq m$ **then**
  $mid_2[s] := m$
  $elems[\ell] := elems[m]$; $loc[elems[\ell]] := \ell$; $elems[m] := e$; $loc[e] := m$

*Split*1(s)
if $mid_1[s] = mid_2[s]$ **then** $mid_1[s] := first[s]$
if $mid_1[s] = first[s]$ **then return** 0
else
  $sets := sets + 1$; $uidx[sets] := uidx[s]$
  $first[sets] := first[s]$; $end[sets] := mid_1[s]$; $first[s] := mid_1[s]$
  $mid_1[sets] := first[sets]$; $mid_2[sets] := end[sets]$
  for $\ell := first[sets]$ **to** $end[sets] - 1$ **do** $sidx[elems[\ell]] := sets$
  return $sets$

*Split*2(s)
if $mid_1[s] = mid_2[s]$ **then** $mid_2[s] := end[s]$
if $mid_2[s] = end[s]$ **then return** 0
else
  $sets := sets + 1$; $uidx[sets] := uidx[s]$
  $first[sets] := mid_2[s]$; $end[sets] := end[s]$; $end[s] := mid_2[s]$
  $mid_1[sets] := first[sets]$; $mid_2[sets] := end[sets]$
  for $\ell := first[sets]$ **to** $end[sets] - 1$ **do** $sidx[elems[\ell]] := sets$
  return $sets$

No_marks(s)
if $mid_1[s] = first[s] \wedge mid_2[s] = end[s]$ **then return** True **else return** False

Fig. 2. Main features of the refinable partition data structure

elems Contains $1, 2, \ldots,$ *items* in such an order that elements that belong to the same set are one after another. It is also the case that the sets that belong to the same bunch are one after another in *elems*.

Bunch(s)
return $uidx[s]$

Bunch_first(u)
return $elems[ufirst[u]]$

Bunch_next(e)
if $loc[e]+1 \geq uend[uidx[sidx[e]]]$ **then return** 0 **else return** $elems[loc[e]+1]$

Has_many(u)
if $end[sidx[elems[ufirst[u]]]] \neq uend[u]$ **then return** True **else return** False

Extract_set(u)
$s_1 := sidx[elems[ufirst[u]]]$; $s_2 := sidx[elems[uend[u]-1]]$
if $s_1 = s_2$ **then return** 0
else
  $bunches := bunches + 1$
  if $Size(s_1) \leq Size(s_2)$ **then** $ufirst[u] := end[s_1]$ **else** $uend[u] := first[s_2]$; $s_1 := s_2$
  $ufirst[bunches] := first[s_1]$; $uend[bunches] := end[s_1]$; $uidx[s_1] := bunches$
  return s_1

Left_neighbour(e)
$\ell := first[sidx[e]]$; **if** $\ell > 1$ **then return** $elems[\ell-1]$ **else return** 0

Right_neighbour(e)
$\ell := end[sidx[e]]$; **if** $\ell \leq items$ **then return** $elems[\ell]$ **else return** 0

Fig. 3. Bunch- and neighbour-features of the refinable partition data structure

first **and** *end* Indicate the segment in *elems* where the elements of a set are stored. That is, $A_s = \{\, elems[f],\ elems[f+1],\ \ldots,\ elems[\ell-1] \,\}$, where $f = first[s]$ and $\ell = end[s]$.

mid_1 **and** mid_2 Let f and ℓ be as above, and let $m_1 = mid_1[s]$ and $m_2 = mid_2[s]$. Then $A_s^1 = \{\, elems[f], \ldots, elems[m_1-1] \,\}$, the unmarked elements are $elems[m_1], \ldots, elems[m_2-1]$, and $A_s^2 = \{\, elems[m_2], \ldots, elems[\ell-1] \,\}$.

loc Tells the location of each element in *elems*, that is, $elems[loc[e]] = e$.

sidx The index of the set that e belongs to is $sidx[e]$. That is, $e \in A_{sidx[e]}$.

uidx The index of the bunch that A_s belongs to is $uidx[s]$. That is, $A_s \in U_{uidx[s]}$.

ufirst **and** *uend* The union of the sets in bunch U_u is $\{\, elems[f],\ elems[f+1], \ldots, elems[\ell-1] \,\}$, where $f = ufirst[u]$ and $\ell = uend[u]$.

To avoid marking the same element more than once, *Mark*1(e) first tests that the element e is in the segment for unmarked elements of the set that contains e. If it is, then *Mark*1 swaps the element with the first unmarked element, and moves the borderline between 1-marked and unmarked elements one location forward. The *loc* array is updated according to the new locations of the swapped elements. *Mark*2 works similarly, but with the last unmarked element.

If set s does not contain 1-marked elements, *Split*1(s) returns zero and terminates on its second line. If it does not contain unmarked elements, the first line unmarks all 1-marked elements, leading to termination on the second line. Otherwise *Split*1 adjusts the number of sets and the set boundaries so that the

1-marked elements become a new set whose all elements are unmarked. The second statement on line 4 makes the new set a member of the same bunch as the original set. The **for**-loop updates the set index of the 1-marked elements to refer to the new set. *Split*2 works similarly with 2-marked elements.

Has_many(u) finds the end location of the set that the first element of bunch u belongs to, and tests if it is different from the end location of the bunch.

Extract_set(u) first finds the indices of the first and last set in bunch u. If they are the same, the bunch consists of only one set, and *Extract_set* returns zero. Otherwise *Extract_set* chooses the smaller of the two sets, removes it from the bunch, and makes a new bunch of it.

Left_neighbour(e) returns the element that is immediately before the set that contains e in *elems*, or zero, if the set is the first set in *elems*. *Right_neighbour* works similarly at the opposite end of the set.

The time consumption of initializing the data structure with the partition that consists of one set is linear in *items*. The operations obviously run in constant time, except *Split*1 and *Split*2. Fortunately amortized analysis reveals that they, too, can be treated as constant time in the analysis of the algorithm as a whole. Their running times are proportional to the number of *Mark*1- and *Mark*2-operations executed after the previous splitting. Thus the total cost does not change, even if the split operations are only charged constant cost.

4 The Algorithm

In this section the new bisimilarity minimization algorithm is described. Its operation and asymptotic time consumption are discussed at an informal level. Detailed correctness and performance proofs are deferred to Section 5.

It is assumed that states and labels are represented with numbers. That is, $S = \{1, 2, \ldots, n\}$ and $L = \{1, 2, \ldots, \alpha\}$. We have $\Delta \subseteq S \times L \times S$ and $m = |\Delta|$, that is, there are m transitions. Let $\mathcal{I} = \{S_1, S_2, \ldots, S_k\}$ denote the initial partition of S. The input to the algorithm consists of n, α, Δ, and $S_2, \ldots, S_k$. The set S_1 need not be given, because it is $S - (S_2 \cup \cdots \cup S_k)$.

Let $\Delta_{a,B} = \Delta \cap (S \times \{a\} \times B)$ and $\Delta_{s,a,B} = \Delta \cap (\{s\} \times \{a\} \times B)$. That is, $\Delta_{a,B}$ is the set of those transitions whose label is a and whose end state is in B. Adding the requirement that the start state must be s converts $\Delta_{a,B}$ to $\Delta_{s,a,B}$.

It is assumed that transitions are represented via three arrays *tail*, *label*, and *head*. Each transition (s, a, s') has an index t in the range $1, \ldots, m$ such that $tail[t] = s$, $label[t] = a$, and $head[t] = s'$. It is also assumed that the indices of the transitions that share the same head state s are available via $In\_transitions[s]$ in some unspecified order. It may be implemented similarly to *elems*, *first*, and *end*, and initialized in $\Theta(m + n)$ time with counting sort. For convenience, confusing the transitions with their indices will be allowed in formulae, like in $In\_transitions[s] = \{\, (s_1, a, s_2) \in \Delta \mid s_2 = s \,\}$.

For keeping track of work to be done, the algorithm uses four *worksets*. In the prototype implementation, stacks were used as the worksets. However, they need not be stacks. It suffices that they provide three constant time operations:

Add(e) that adds the element e to the workset without checking whether it already is there, *Remove* that removes any element of the workset and returns the removed element, and *Empty* that returns True if and only if the workset is empty. The *capacity* of a workset is the maximum number of elements it can store.

The algorithm uses the following data structures:

Blocks. This is a refinable partition data structure on $\{1, \ldots, n\}$. Its sets are the blocks. The index of the set in *Blocks* is used as the index of the block also elsewhere in the algorithm.

Splitters. This is a refinable partition data structure on $\{1, \ldots, m\}$. Each set in it consists of the indices of the a-labelled input transitions of some block B, for some label a. That is, $\mathit{Splitters} = \{\, \Delta_{a,B} \mid a \in L \wedge B \in \mathit{Blocks} \wedge \Delta_{a,B} \neq \emptyset \,\}$. That this property remains valid is not obvious from the code, so it will be proven later as Lemma 1 (1). The bunch feature of *Splitters* will be used. When saying that a transition is in a bunch it is meant that the bunch contains a splitter that contains the transition.

Outsets. This, too, is a refinable partition data structure on $\{1, \ldots, m\}$, but it stores a finer partition than *Splitters*. Transitions that are in the same set of *Outsets* also share their start state. That is, $\mathit{Outsets} = \{\, \Delta_{s,a,B} \mid s \in S \wedge a \in L \wedge B \in \mathit{Blocks} \wedge \Delta_{s,a,B} \neq \emptyset \,\}$. This will be proven as Lemma 1 (2).

Unready_Bunches. This is an initially empty workset of capacity $\lfloor m/2 \rfloor$. It contains the indices of the bunches of *Splitters* that consist of two or more splitters. This will be proven as Lemma 1 (3).

Touched_Blocks. This is an initially empty workset of capacity n. It contains the indices of the blocks that have been affected when using a splitter, but have not yet been split. In other words, precisely those blocks contain marked states.

Touched_Splitters **and** *Touched_Outsets*. These are initially empty worksets of capacity m. They contain the indices of the sets in *Splitters* and *Outsets*, respectively, that must be updated, because the block where their transitions end has been split. In other words, precisely those splitters or outsets contain marked transitions.

Before discussing the main algorithm, it is useful to introduce the *Update* subroutine that is shown in Figure 4. Each time a block has been split, it is necessary to split sets in *Splitters* and *Outsets* accordingly, to keep them consistent with *Blocks* in the above-mentioned sense. The updating of *Splitters* may make it necessary to update *Unready_Bunches*, to maintain its above-mentioned relation to *Splitters*. These duties are taken care of by *Update*.

When *Update* is called, b and b' contain the indices of the halves of the block that has just been split. Both halves must be non-empty.

To understand *Update*, let us first discuss what happens to a single set in *Outsets* and temporarily ignore the rest. *Update* chooses one of the halves of the block that has just been split, and marks those transitions of the outset that end in the chosen half. (The implementation of this will be discussed soon.) If there

```
    Update(b, b')
1   if Blocks.Size(b) ≤ Blocks.Size(b') then s := Blocks.First(b)
2   else s := Blocks.First(b')
3   while s ≠ 0 do
4       for t ∈ In_transitions[s] do
5           p := Splitters.Set(t);  o := Outsets.Set(t)
6           if Splitters.No_marks(p) then Touched_Splitters.Add(p)
7           if Outsets.No_marks(o) then Touched_Outsets.Add(o)
8           Splitters.Mark1(t);  Outsets.Mark1(t)
9       s := Blocks.Next(s)
10  while ¬Touched_Splitters.Empty do
11      p := Touched_Splitters.Remove
12      u := Splitters.Bunch(p);  if Has_many(u) then u := 0
13      p' := Splitters.Split1(p);  if u ≠ 0 ∧ p' ≠ 0 then Unready_Bunches.Add(u)
14  while ¬Touched_Outsets.Empty do
15      o := Touched_Outsets.Remove;  o' := Outsets.Split1(o)
```

Fig. 4. The *Update* subroutine

are no such transitions, then nothing happens to the outset. Otherwise, after the marking phase, *Update* calls the split operation on the outset. If all transitions of the outset were marked, then the split operation just unmarks them. Otherwise, it divides the outset to two halves, those transitions that were marked and those that were not, and unmarks all of its transitions. As a consequence, if all transitions of the outset end in the same half-block, then *Update* does not modify the outset; otherwise, it divides it to two outsets according to in which half-block its transitions end.

To obtain good performance, *Update* updates all outsets in one batch. On lines 1 and 2, *Update* finds the first state in the smaller half-block. As will be discussed later, the performance of the algorithm depends on choosing the smaller half. On lines 3 to 9, s scans through the states in the chosen half, and t scans the input transitions of s on lines 4 to 8. Transitions in the outsets are marked on line 8. The indices of the outsets whose transitions are marked are collected into *Touched_Outsets*. The test on line 7 ensures that the index is added to *Touched_Outsets* only once. The test works, because it is executed before the transition is marked. Lines 14 and 15 pick each outset that contains marked transitions one at a time, and splits it.

The processing of *Splitters* is otherwise similar to *Outsets*, but it contains an additional step on lines 12 and 13. Line 13 may increase the number of splitters in the bunch that contains p. If it is increased from one to two, then the bunch index must be added to *Unready_Bunches*, to maintain the property that it contains the indices of precisely those bunches that consist of more than one splitter. If line 13 does not actually split p, the test $p' \neq 0$ fails and *Unready_Bunches* is not changed. If the bunch is already in *Unready_Bunches*, then u becomes zero on line 12 and *Unready_Bunches* is not changed. The code is a bit complicated, because *Has_many* must be executed before p is split.

```
     Main_part
16   initialize Blocks to {S} and Splitters to { Δ_{a,S} | a ∈ L ∧ Δ_{a,S} ≠ ∅ }
17   make every set of Splitters a singleton bunch
18   initialize Outsets to { Δ_{s,a,S} | s ∈ S ∧ a ∈ L ∧ Δ_{s,a,S} ≠ ∅ }
19   for i := 2 to k do
20       for s ∈ S_i do Blocks.Mark1(s)
21       b := Blocks.Split1(1);  Update(1, b)
22   for u := 1 to Splitters.bunches do
23       t := Splitters.Bunch_first(u)
24       while t ≠ 0 do
25           s := tail[t];  b := Blocks.Set(s)
26           if Blocks.No_marks(b) then Touched_Blocks.Add(b)
27           Blocks.Mark1(s);  t := Splitters.Bunch_next(t)
28       while ¬Touched_Blocks.Empty do
29           b := Touched_Blocks.Remove
30           b' := Blocks.Split1(b);  if b' ≠ 0 then Update(b, b')
31   while ¬Unready_Bunches.Empty do
32       u := Unready_Bunches.Remove;  p := Splitters.Extract_set(u)
33       if Splitters.Has_many(u) then Unready_Bunches.Add(u)
34       t := Splitters.First(p)
35       while t ≠ 0 do
36           if t = Outsets.First( Outsets.Set(t) ) then
37               s := tail[t];  b := Blocks.Set(s)
38               if Blocks.No_marks(b) then Touched_Blocks.Add(b)
39               t1 := Outsets.Left_neighbour(t);  t2 := Outsets.Right_neighbour(t)
40               if     t1 > 0 ∧ tail[t1] = s ∧ Splitters.Bunch(Splitters.Set(t1)) = u
41                   ∨ t2 > 0 ∧ tail[t2] = s ∧ Splitters.Bunch(Splitters.Set(t2)) = u
42               then Blocks.Mark1(s) else Blocks.Mark2(s)
43           t := Splitters.Next(t)
44       while ¬Touched_Blocks.Empty do
45           b := Touched_Blocks.Remove
46           b' := Blocks.Split1(b);  if b' ≠ 0 then Update(b, b')
47           b' := Blocks.Split2(b);  if b' ≠ 0 then Update(b, b')
```

Fig. 5. Main part of the bisimilarity minimization algorithm

Touched_Splitters and *Touched_Outsets* are always empty when *Update* is started, because they are initially empty, not used elsewhere, and *Update* leaves them empty. The number of *Remove*-operations on them is thus the same as the number of *Add*-operations. The cost of each split-operation is linear in the number of the corresponding mark-operations. Other individual operations in *Update* take constant time. Therefore, its execution time is dominated by lines 3 to 9. It is thus linear in the sum of the number of states in the scanned half-block and the total number of their input transitions.

The main algorithm is shown in Figure 5. It starts by initializing *Blocks*, *Splitters*, and *Outsets* on lines 16 to 18 according to the situation where there is only one block, and each splitter constitutes alone a bunch. *Unready_Bunches* is initially empty. This is consistent with the effect of line 17.

The time consumption of the initialization is not otherwise a problem, but the initialization of *Splitters* and *Outsets* requires putting the transitions in a suitable order in *Splitters.elems* and *Outsets.elems*. Sorting the transitions with heapsort would take $\Theta(m \log m)$ time in the worst case, which is more than is allowed. Sorting them with counting sort using the label as the key would take $\Theta(m + \alpha)$ time and memory. That is too much, when $\alpha = \omega(m \log n)$. There is a trick with which the transitions can be classified according to their labels in $\Theta(m)$ time and $\Theta(m + \alpha)$ memory. It is based on [1, Exercise 2.12] and was applied to the present purpose in [12]. The resulting order is suitable for *Splitters*. Counting sorting the transitions with the start state as the key before using the trick makes the resulting order suitable also for *Outsets*, and only takes $\Theta(n + m) = \Theta(m)$ extra time and memory.

Lines 19 to 21 split the original block according to the initial partition given in the input, and update the other data structures accordingly. In the beginning of line 21, block 1 is $S_1 \cup S_i \cup S_{i+1} \cup \cdots \cup S_k$. The split operation extracts S_i and makes it block b. Because the S_i constitute a partition, none of them is empty. Therefore, the half-blocks 1 and b are both non-empty when *Update* is called.

Now blocks have to be split until the partition is compatible. As has been discussed above, the sets in *Splitters* correspond precisely to the non-empty $\Delta_{a,B}$, where a is a label and B is a block. Therefore, the splitting obligation introduced by a and B can be met by marking the start states of the transitions in the corresponding splitter and then splitting the blocks that contain marked states.

To keep track of pending splitting obligations, the algorithm uses the bunches of *Splitters* together with *Unready_Bunches*. The bunches are used largely in the same way as compound blocks were used in [11].

The algorithm first establishes and then maintains the property that *if two states are in the same block, then, for each bunch in Splitters, either none or both of them have an output transition in the bunch.* This invariant is crucial for performance, as it allows to skip a largest splitter in the bunch when splitting.

The test on line 31 implies that when the algorithm terminates, each bunch consists of a single splitter. Therefore, upon termination, if two states are in the same block, then, for each splitter, either none or both of them have an output transition in the splitter. This means that all splitting obligations have been satisfied. In other words, for all states s_1, s_2, and s'_1, labels a, and blocks B and B', if $s_1 -a\rightarrow s'_1$ and $s_1 \in B$ and $s_2 \in B$ and $s'_1 \in B'$, then s_1 has an output transition in the splitter that corresponds to $\Delta_{a,B'}$, so also s_2 has, implying that there is some $s'_2 \in B'$ such that $s_2 -a\rightarrow s'_2$. That is, the partition is compatible.

The **for**-loop on lines 22 to 30 establishes the above-mentioned property. For each bunch, it splits the blocks so that the property starts to hold. Each cycle around the **for**-loop processes *Blocks* similarly to the processing of *Outsets* in *Update*. On lines 23 to 27, the start states of the transitions in the bunch are marked, and the indices of the blocks that contain marked states are collected into *Touched_Blocks*. *Touched_Blocks* is discharged on lines 28 to 30 by splitting

the touched blocks. For each splitting, if both halves are non-empty, then *Update* is called, to keep the other data structures consistent with *Blocks*.

Lines 22 to 30 actually separate the states according to the labels of their output transitions. This is because bunches have not yet been divided after their initialization, although splitters in them may have. Each bunch thus contains precisely the a-labelled transitions for some label a. Therefore, lines 22 to 30 separate two states if and only if, for some label a, one of them has and the other does not have an a-labelled output transition.

Lines 31 to 47 discharge *Unready_Bunches* while maintaining the above-mentioned property. On line 32, one bunch is chosen for processing, and a splitter is extracted from it. If the remaining part of the bunch still contains more than one splitter, line 33 puts it back to *Unready_Bunches*, in accordance with the main property of *Unready_Bunches*.

To re-establish the above-mentioned property, it may be necessary to separate two states because only one of them has an output transition in the remaining bunch u, or because only one of them has an output transition in the extracted splitter p (which is now a bunch on its own). This means that states of each block have to be separated into three groups: those that have output transitions only in p, only in u, or in both. Lines 34 to 47 do that according to a pattern that we have seen twice before. We now discuss what is new on those lines.

The first novelty is the test on line 36. All transitions in the same set of *Outsets* have the same start state. They also have the same left neighbour and the same right neighbour in the sense of line 39. So they all have the same effect on line 42. Therefore, it suffices to investigate only one of them on lines 37 to 42. This is what the test on line 36 achieves. The test is an optimization that affects neither correctness nor asymptotic time consumption, but improves practical time consumption.

The second novelty is the splitting of blocks into three parts, and the test on lines 40 and 41 that controls the splitting. We claim that a state s that has an output transition in p or u is 1-marked, is 2-marked, or remains unmarked, if it has an output transition in both p and u, only in p, or only in u, respectively. Because t scans p, s is marked in some way if and only if it has an output transition in p. If s is 1-marked, then t_1 or t_2 is its output transition in u. It remains to be shown that if a marked s has an output transition in u, then t_1 or t_2 is such a transition. For each s and a, the order of the $\Delta_{s,a,B}$ in *Outsets.elems* is the same as the order of the $\Delta_{a,B}$ in *Splitters.elems*, because the algorithm updates *Splitters* and *Outsets* in a similar way. As a consequence, the output transitions of s that are in p or u are contiguously in *Outsets.elems*. Thus *Left_neighbour* or *Right_neighbour* or both find an output transition in u, if any exists.

Thanks to three issues, the algorithm runs in $O(m \log n)$ time. The first is Hopcroft's trick [7]: because *Extract_set* tries two splitters and extracts the smaller of them, each time when a transition is used for splitting blocks, it belongs to a splitter whose size is at most half the size in the previous time. All transitions in a splitter have the same label, so there can be at most n^2 of them.

Thus each transition can be used at most $\log_2 n^2 = 2 \log_2 n$ times for splitting. Bunches were needed to make it legal to skip the largest splitter.

The second is Knuutila's trick [9]: because *Update* chooses the smaller half-block, each time when a state is used for updating splitters and outsets, it belongs to a block whose size is at most half the size in the previous time. Thus each state can be used at most $\log_2 n$ times for updating.

The third issue is from [12]. It is the organisation of the work in such a way that the set of labels is never scanned. Instead, subsets of transitions are scanned so that if there are no transitions for some label, then no work is done for that label. Failure to obey this principle would easily introduce an $\Omega(n\alpha)$ term to time consumption.

5 Detailed Proofs

The previous section explained the principle of the algorithm. In this section, detailed proofs of its correctness and performance are presented.

As is obvious from earlier discussion, it is important that *Splitters* and *Outsets* are consistent with *Blocks*, and *Unready_Bunches* is consistent with *Splitters*. The duty of the *Update* subroutine is to re-establish consistency each time *Blocks* has changed. Let us state this precisely, and check that also the main algorithm maintains consistency where necessary.

Lemma 1. *The following hold everywhere after line 18, except (1) and (2) on lines 21, 30, 46, and 47 and within Update, and (3) on lines 32, 33, and 13.*

(1) For any two transitions, they are in the same set in Splitters if and only if they have the same label and they end in the same set in Blocks.
(2) For any two transitions, they are in the same set in Outsets if and only if they have the same start state and the same label, and they end in the same set in Blocks. This is equivalent to that they have the same start state and belong to the same set in Splitters.
(3) Unready_Bunches contains the indices of precisely those bunches of splitters that contain two or more splitters.

Proof. The main algorithm starts by initializing *Blocks*, *Splitters*, and *Outsets* on lines 16 to 18 according to the situation where there is only one block. This makes (1) and (2) hold. It also makes (3) hold, because *Unready_Bunches* is initially empty and line 17 makes each bunch of splitters to consist of a single splitter. From then on, each time a block has been split (lines 21, 30, 46, and 47) resulting in two non-empty sub-blocks, *Update* is called with the two halves as the parameters. It splits *Splitters* and *Outsets* so that (1) and (2) are re-established.

The number of splitters in a bunch grows only on line 13. It has already been discussed. The number of splitters in a bunch decreases only on line 32. It removes the bunch from *Unready_Bunches* and removes a splitter from the bunch. Line 33 checks whether the bunch should have remained in *Unready_Bunches*, and puts it back there if necessary. The removed splitter becomes a bunch of its own. It is a singleton bunch, so it is not added to *Unready_Bunches*. □

The test on lines 40 to 42 is tricky enough to deserve a lemma of its own.

Lemma 2. *If line 42 1-marks state s, then s has an outgoing transition in bunch u. If line 42 2-marks s, then s does not have an outgoing transition in u.*

Proof. If line 42 1-marks s, then t_1 or t_2 is clearly such a transition.

Assume now that such a transition t' exists. The state s has been found on line 37 via some t. Immediately before the extract operation on line 32 both t and t' were in u. So, by lines 16 and 17, they have the same label, say a. Let B and B' be the blocks where t and t' end. So $B \neq B'$, $t \in \Delta_{s,a,B}$, and $t' \in \Delta_{s,a,B'}$.

Let $X \sqsubseteq Y$ denote that the elements of set X occupy at most as big indices in the *elems* array in question as the elements of set Y.

Assume first that $\Delta_{s,a,B} \sqsubseteq \Delta_{s,a,B'}$. Thanks to line 18, if $\Delta_{s,a,B} \sqsubseteq \Delta_{s'',a'',B''} \sqsubseteq \Delta_{s,a,B'}$, then $s'' = s$ and $a'' = a$. Therefore, $t_2 \neq 0$ and t_2 is in some $\Delta_{s,a,B''}$, where $B'' \neq B$. The operation of *Update* implies that $\Delta_{a,B} \sqsubseteq \Delta_{a,B''} \sqsubseteq \Delta_{a,B'}$ in *Splitters.elems*. Because *Extract_set* extracted $\Delta_{a,B}$ from one end of u while $\Delta_{a,B'}$ stayed in u, also $\Delta_{a,B''}$ was and stayed in u. As a consequence, t_2 is in u and passes the test on line 41. So s is 1-marked.

The case $\Delta_{s,a,B'} \sqsubseteq \Delta_{s,a,B}$ is symmetric with t_1 replacing t_2. □

The next lemma says that the algorithm does not do any splitting that it should not.

Lemma 3. *If the algorithm puts two states in different blocks, then those states belong to different blocks in each partition that is a refinement of $\mathcal{I}$ and compatible with Δ.*

Proof. If two states go into different blocks on line 21, then they are in different blocks in $\mathcal{I}$.

When lines 22 to 30 are executed, the bunches of splitters still contain the same transitions as originally, although they may have been divided into many splitters. Thus each execution of lines 23 to 27 scans some $\Delta_{a,S}$. Therefore, if line 30 separates two states, then one of them has and the other does not have an outgoing a-transition.

The case remains where line 46 or 47 puts the states into different blocks. Assume that s_1 is moved to a new block on line 46 and s_2 on line 47, while s_0 stays in the original block. By Lemma 1 (1), there is some $B \in \mathit{Blocks}$ and $a \in L$ such that s_1 and s_2 have but s_0 does not have an a-transition to B. (B is the block and a is the label that correspond to the splitter that is scanned on lines 34 to 43.) It is thus necessary to separate s_0 from s_1 and s_2 to obtain a compatible partition.

By Lemma 2, s_1 has and s_2 does not have an output transition that belongs to u. Let a be the label and B' the end block of that transition of s_1. Then s_1 has and s_2 does not have an a-transition that ends in B'. So it is correct to put s_1 and s_2 into different blocks. □

The next lemma says that the algorithm does all the splitting that it should.

Lemma 4. *When the algorithm terminates, Blocks is a refinement of $\mathcal{I}$ and compatible with Δ.*

Proof. Lines 19 to 21 ensure that *Blocks* will be a refinement of $\mathcal{I}$. The rest of the proof is based on the following Gries-style [6] invariant.

> On line 31, for every states s_1 and s_2 that are in the same block, transition t_1 that starts at s_1, and bunch of splitters u that contains t_1, there is a transition t_2 that starts at s_2 and is in u.

Lines 22 to 30 make the invariant hold by separating s_1 to a different block from s_2, if t_2 does not exist.

The constituents of the invariant may change only when a block is split or the set of transitions in a bunch is modified. Splitting a block is not a threat to the invariant (merging blocks would be, but the algorithm does not do that). Only *Extract_set* modifies the set of transitions in a bunch, and the only place where it is executed is line 32. There a bunch is divided to a new singleton bunch that consists of the splitter p, and u that contains the rest of the original bunch.

The purpose of lines 34 to 47 is to split blocks into up to three parts according to the existence of an output transition in p but not in u, in u but not in p, and in both. If s does not have an output transition in p, then it stays in its block. In the remaining two cases, by Lemma 2 it is put in a different block on line 46 or 47 depending on which case holds. So the invariant is re-established.

Lemma 1 (3) implies that when the algorithm terminates, every bunch consists of precisely one splitter. Then the invariant actually says that for every states s_1 and s_2 that are in the same block, transition t_1 that starts at s_1, and splitter p that contains t_1, there is a transition t_2 that starts at s_2 and is in p. By Lemma 1 (1), this is equivalent to that *Blocks* is compatible with Δ. □

The efficiency of the algorithm is stated in the next lemma.

Lemma 5. *The algorithm runs in $O(m \log n)$ time and $O(m + \alpha)$ memory (assuming that $n \leq 2m$).*

Proof. All data structures consume $O(n)$, $O(m)$, or $O(\alpha)$ memory. The running time of lines 16 to 18 was discussed in Section 4. Excluding the time spent in the loops within *Update*, lines 19 to 21 are obviously $O(n)$ and lines 22 to 30 $\Theta(m)$.

Because *Extract_set* avoids choosing the largest set, each splitter that is used as the p on lines 32 to 47 inherits at most half of the transitions of the bunch from which it is extracted. The splitter becomes a new bunch. As a consequence, when any transition is used anew for splitting a block, it belongs to a splitter whose size is at most half the size in the previous time. Initially a splitter contains at most n^2 transitions. So the same transition can be used at most $2 \log_2 n$ times. Therefore, lines 36 to 43 and 45 to 47 are executed at most $2m \log_2 n$ times. Because splitters are not empty, lines 32 to 34 are executed at most the same number of times as line 36, and lines 35 and 44 at most twice that many times. Line 31 is executed once more than line 32.

By now the act of calling *Update* has been taken into account in the analysis, but the execution of *Update* has not. Lines 1 and 2 determine whether b or b' is scanned. If b' is scanned, then each scanned state was marked on line 20, 27, or 42. Otherwise other states are scanned, but their number is at most the same. So the n, m, or $2m \log_2 n$ bound applies to line 9. Whenever lines 5 to 8 are executed anew for some t, the test on lines 1 and 2 guarantees that $head[t]$ belongs to a block whose size is at most half of the size in the previous time. This implies an $m \log_2 n$ upper bound. Lines 11 to 13 and 15 are executed at most as often as line 5. So every line of the algorithm meets the $O(m \log n)$ bound. □

(It is indeed the case that the reasons why lines 8 and 9 meet the time bound are different. Each of them may execute more often than the other, as $In\_transitions[s]$ may be empty for many s. A similar issue was discussed in [9].)

Corollary 1. *The algorithm solves the bisimilarity minimization problem in* $O(m \log n)$ *time and* $O(m + \alpha)$ *memory (assuming that* $n \leq 2m$*).*

6 Experience with a Prototype Implementation

For the purpose of testing the new algorithm and getting an idea of its performance, the present author implemented it in C++. No reference implementation was available for the general problem, but, thanks to [12], two comparable programs were available for the special case of DFA minimization. Therefore, a pre-processing stage was added that removes unreachable states, and non-initial states from which no state in S_1 is reachable. This made the new program applicable as such both to DFA minimization and to bisimilarity minimization with at most two initial blocks.

The author tested the correctness of his implementation first by giving an extensive set of randomly generated DFAs to the new and the reference program, and checking that the outputs were isomorphic. Then he tested the new program with more than 300 randomly generated nondeterministic graphs of various sizes and densities. Unfortunately, in the nondeterministic case there is no reference program, no straightforward way to check isomorphism, nor other simple way of fully checking the output. Therefore, each nondeterministic graph was given to the program in four different versions, and it was checked that the four outputs had the same number of states and the same number of transitions. Two of the versions were obtained by randomly permuting the numbering of states in the original version, and the first output was used as the fourth input.

Timing measurements were conducted by Petri Lehtinen and executed on a PC with Linux and 1 gigabyte of memory. A sample of results with randomly generated nondeterministic graphs is shown in Table 1. Each entry shows the fastest and slowest of three measurements, made with $|S_1| = n/2 + d$ and $|S_2| = n/2 - d$, where $d \in \{-1, 0, 1\}$. The times are shown in seconds. The clock was started when the input file had been read, and stopped when the program was ready to start writing the output file. No attempt was made to optimize the implementation to the extreme. In particular, all instances of the refinable

Table 1. Running time with nondeterministic input containing two initial blocks

n	α	$m = 20\,000$		$m = 50\,000$		$m = 100\,000$		$m = 200\,000$		$m = 500\,000$		1 000 000	
1 000	10	0.017	0.017	0.043	0.044	0.119	0.123	0.092	0.365	0.257	0.260	0.539	0.547
1 000	100	0.021	0.022	0.084	0.088	0.251	0.255	0.501	0.505	1.016	1.025	2.330	2.402
10 000	10	0.005	0.005	0.027	0.027	0.074	0.079	0.505	0.512	1.404	1.456	2.926	2.983

partition data structure contained also the arrays and functionality (like the bunches) that the particular instance does not need.

Because it is difficult to generate a precise number of transitions according to the uniform distribution, sometimes the generated number was slightly smaller than the desired number. Running time depends also on the size of the result: the smaller it is, the less splitting of blocks. It is also very difficult to get full control of all other activity that is going on in a modern computer. As a consequence, the measurements contain some noise. The results should be considered as typical, not as the absolute truth.

When m is big enough compared to $n\alpha$, each state is likely to have an output transition with every label to both a state in S_1 and in S_2, causing the graph to minimize to 2 states and 4α transitions, while with a smaller m the graph does not reduce much. This explains the anomaly with $n = 1\,000$, $\alpha = 10$, and $m = 200\,000$. The row $n = 10\,000$ is subject to another, smoother phenomenon that unduly reduces execution time: when m is small, many states are removed in the pre-processing stage as unreachable from the initial state or as unable to reach any final state. Altogether, the issue of precise running time is complicated.

7 Conclusions

The algorithm in this paper looks complicated. To some extent it is because it was presented in great detail. A significant part of its implementation could be obtained by copying the pseudocode in the figures and the definitions of arrays in the main text, and converting them to the programming language in question. This is how the author implemented the prototype. Algorithm descriptions in research papers (like [11]) are often so sketchy that they are very hard to implement. The author wanted this not to be the case with the present paper.

Only one of the instances of the refinable partition data structure used by the algorithm uses *Mark*2 and *Split*2, and only one uses the bunch feature. These features are supported by additional arrays. Leaving them out from the instances that do not use them would improve the performance of the prototype.

The neighbour trick on lines 39 to 41 and Lemma 2 is ugly, because it breaks the otherwise clean abstract interface of the data structure. It also made it necessary to introduce *Outsets* and *Touched_Outsets*. In [11], a similar problem was solved by keeping track of how many transitions to each compound block each state has. Finding the appropriate counter quickly enough is not trivial, so the technique is somewhat complicated. It is plausible that something similar could have been done in the new algorithm. We leave it for the future to find out if it would work and be better than the chosen approach.

In verification of concurrent systems, it is common to use equivalence notions that abstract away from invisible actions. Bisimilarity does not do that. However, it preserves all commonly used equivalences. Therefore, the new algorithm can be used as a preprocessing stage that makes the graph smaller before it is given to a reduction or minimization algorithm of the equivalence in question. Because the new algorithm is cheap compared to most algorithms for other equivalences, this kind of preprocessing may save a lot of time in practice.

When minimizing with respect to observation equivalence by saturating the graph and then running bisimilarity minimization [8], the growth in the number of transitions caused by saturation is a problem. A natural, but apparently difficult, topic for future research is whether saturation could be replaced by adding suitable graph traversal to the new algorithm, without losing too much of its good performance.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing Finite Kripke Structures in Propositional Temporal Logic. Theoret. Comput. Sci. 59, 115–131 (1988)
3. Dovier, A., Piazza, C., Policriti, A.: An Efficient Algorithm for Computing Bisimulation Equivalence. Theoret. Comput. Sci. 311, 221–256 (2004)
4. Fernandez, J.-C.: An Implementation of an Efficient Algorithm for Bisimulation Equivalence. Science of Computer Programming 13, 219–236 (1989/1990)
5. Gries, D.: Describing an Algorithm by Hopcroft. Acta Inform. 2, 97–109 (1973)
6. Gries, D.: The Science of Programming. Springer, Heidelberg (1981)
7. Hopcroft, J.: An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. Technical Report CS-190, Stanford University (1970)
8. Kanellakis, P., Smolka, S.: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In: 2nd ACM Symposium on Principles of Distributed Computing, pp. 228–240 (1983)
9. Knuutila, T.: Re-describing an Algorithm by Hopcroft. Theoret. Comput. Sci. 250, 333–363 (2001)
10. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
11. Paige, R., Tarjan, R.: Three Partition Refinement Algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
12. Valmari, A., Lehtinen, P.: Efficient Minimization of DFAs with Partial Transition Functions. In: Albers, S., Weil, P. (eds.) STACS 2008, Symposium on Theoretical Aspects of Computer Science, Bordeaux, France, pp. 645–656 (2008), `http://drops.dagstuhl.de/volltexte/2008/1328/`