Fundamenta Informaticae 105 (2010) 319–339
DOI 10.3233/FI-2010-369
IOS Press

Simple 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 of states with respect to the labelled transitions. A splitter is a pair consisting of a label and a set in the partition. A transition belongs to a splitter, if and only if it has the same label and its end state is in the set. Earlier algorithms consume lots of time in scanning splitters that have no transitions. The new algorithm avoids this by maintaining also a partition of transitions. To facilitate this, a refinable partition data structure with amortized constant time operations is used. Detailed pseudocode of all nontrivial details is presented. Novel simplifications are applied that both reduce the practical memory consumption and shorten the program code.

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

*Address for correspondence: Tampere University of Technology, Department of Software Systems, PO Box 553, FI-33101 Tampere, Finland

"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 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. The union of all bisimulations is the largest bisimulation. 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 listing of the truth values 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 (up to isomorphism) smallest system that is bisimilar with 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 is the fused state that contains the end state of the original 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.

This article concentrates on finding the bisimilarity equivalence classes. It has developed from [12], where a new algorithm was presented whose asymptotic time consumption is better than that of earlier algorithms. The asymptotic time consumption does not depend on the number of different labels of transitions. This is an advantage when there are many different labels, like "send$\langle x \rangle$" and "receive$\langle x \rangle$" where x may assume many different values.

When bigger and bigger inputs were given to the implementation discussed in [12], memory rather than time consumption became a limiting factor. Furthermore, new ideas have been found since writing [12]. As a consequence, the algorithm has been significantly re-designed to minimize practical memory consumption while not sacrificing the good asymptotic time consumption. The most impor-

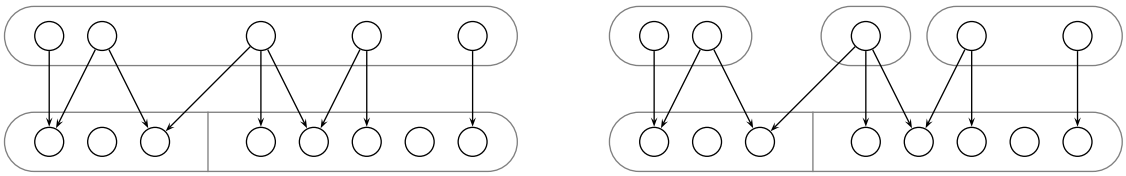

Figure 1. Illustrating three-way splitting. All transitions have the same label.

tant changes are listed in Section 2. This article also presents some unchanged issues more thoroughly than [12].

In Section 2 we describe the problem in more detail and discuss earlier work. The new algorithm uses two copies of a special refinable partition data structure that is described in Section 3. The new algorithm is presented, its correctness justified, and its performance analysed in Section 4. Section 5 contains some timing measurements that were made with a prototype implementation, and the conclusions.

2. Background

A *partition* $\mathcal{S}$ of a set S is a collection of nonempty, pairwise 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 $\{Z_1, Z_2, \ldots, Z_h\}$ of S such that each Z_i is a subset of some S_j. The algorithm takes as an input a labelled directed graph (S, L, Δ), where $\Delta \subseteq S \times L \times S$, together with an initial partition $\mathcal{I}$ of S. The initial partition replaces the state labels in Section 1: two states belong to the same set of $\mathcal{I}$ if and only if they have the same label. By $s\,{-a\!\rightarrow}\,s'$ we mean that $(s, a, s') \in \Delta$. The transition $s\,{-a\!\rightarrow}\,s'$ is an *input transition* of s' and *output transition* of s. 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 $s_1\,{-a\!\rightarrow}\,s'_1$, there is an $s'_2 \in S'$ such that $s_2\,{-a\!\rightarrow}\,s'_2$.

It is easy to see that an equivalence relation on S is a bisimulation if and only if its equivalence classes constitute a partition that is a refinement of $\mathcal{I}$ and compatible with Δ. In particular, the equivalence classes of the coarsest bisimulation—that is, bisimilarity—constitute the coarsest partition that is a refinement of $\mathcal{I}$ and compatible with Δ. The task of the bisimilarity minimization problem is to find it.

We denote the numbers of states, transitions, and labels by $n = |S|$, $m = |\Delta|$, and $\alpha = |L|$. To avoid getting into troublesome technicalities with complexity formulae, we assume that $n \leq 2m$ when discussing complexity. 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. An assumption to the same effect was also made by most other authors, although it was not always made clear. Indeed, otherwise time complexity would have to be at least $O(n + m \log n)$, because reading $\mathcal{I}$ takes $\Theta(n)$ time in the worst case.

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

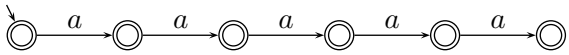

Figure 2. Demonstrating that splitting by traversing transitions forward may be slow.

called *blocks*. If s_1 and s_2 are in the same block B and $s_1 -a\rightarrow s'_1$, where s'_1 is in block B', but there is no $s'_2 \in B'$ such that $s_2 -a\rightarrow s'_2$, then B must be split so that s_1 and s_2 go into different parts. This splitting may make further splitting necessary. Figure 1 illustrates this. The block in the bottom row has just been divided to two parts. The block in the top row must be divided to three parts, because some states in it have transitions to only the left or only the right block in the bottom row, while one state has transitions to both. We will call this *three-way splitting*.

Hopcroft's famous deterministic finite automaton (DFA) minimization algorithm [7] is an early subquadratic algorithm for an important subproblem of the bisimilarity minimization problem. DFA minimization consists of removing irrelevant states (which [7] skipped) and solving a restricted version of the bisimilarity minimization problem. In that version, $|\mathcal{I}| \leq 2$ (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 $s -a\rightarrow s'$. 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 [6] or [9]). It uses *splitters*. Precise meaning varies in the literature, but let us define a splitter as a label–block pair (a, B). It is used for splitting each block according to whether its states do or do not have an a-labelled output transition whose end state is in B.

It might seem natural to split each block by scanning its states and checking which of them have an a-transition to some state in B. However, Figure 2 illustrates that this approach may yield $\Theta(n^2)$ running time when $\alpha = 1$. First all six states would be scanned and the rightmost one would be separated from the others, because it does not have an output a-transition. Next the five leftmost states would be scanned and the rightmost one of them would be separated, because its a-transition leads to a different block from the a-transitions of the other four states. Then the same happens with the four leftmost states, and so on.

In Hopcroft's algorithm, the a-transitions that end in B are traversed backwards, and their start states are moved to tentative new blocks. 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 subproblem 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 union of ordinary blocks. When a block is used for splitting, it is subtracted from its compound block and becomes a compound block of its own. As a consequence, the splitting effect of each compound block has been obtained. So the use of a 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 [5, p. 229] does not meet the challenge. The article 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 [4, p. 242], the label of each transition was 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. We will see in Section 5 that our algorithm consumes $12m + 8n + \max\{m, n\} + |\mathcal{I}| + O(1)$ words of memory, while the middle states approach consumes $19m + 10n + |\mathcal{I}| + O(1)$ words. In this comparison, our algorithm, adapted to the absence of labels, is used in the middle states approach instead of the algorithm in [11], because the latter consumes more memory.

When α is not fixed, an $O(m \log n)$ time algorithm even for the deterministic case had not been published until 2008 [14]. The problem has been the time spent in scanning empty splitters, that is, splitters whose block does not have input 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 [14], this was avoided by maintaining the nonempty 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, [14] presented a refinable partition data structure, one instance of which was used for the blocks and another for the transitions. In this article, the sets of this partition of transitions are called *clusters*.

The above idea was applied to the Paige–Tarjan algorithm in [12], to design an $O(m \log n)$ time algorithm for the bisimilarity minimization problem. The counter technique of Paige and Tarjan was replaced by a third instance of the refinable partition data structure. To implement three-way splitting of blocks and to mimic the compound blocks, new features were added to the data structure.

This article has developed from [12], but the algorithm has changed significantly, making it much simpler and reducing its practical memory consumption a lot. At the level of the O-notation, neither time nor memory consumption has changed. Execution time measured in seconds is so complicated a phenomenon in modern computers that it is impossible to say whether it has changed. The third instance of the refinable partition data structure has been rejected in favour of the counter technique of Paige and Tarjan. There is no representation of (mimicked) compound blocks other than the counters. Using a simple but apparently novel idea, the main data structure used for keeping track of work to be done in the future has been optimized away. These changes made it also possible to eliminate two auxiliary data structures. Because of the changes, proofs of correctness and speed had to be thoroughly rewritten.

3. A Refinable Partition Data Structure

In this section, a refinable partition data structure is presented. It is a modification of the structure presented in [14]. 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 article two instances of it will be used, one where the elements are states and $items = n$, and another where the elements are transitions and $items = m$.

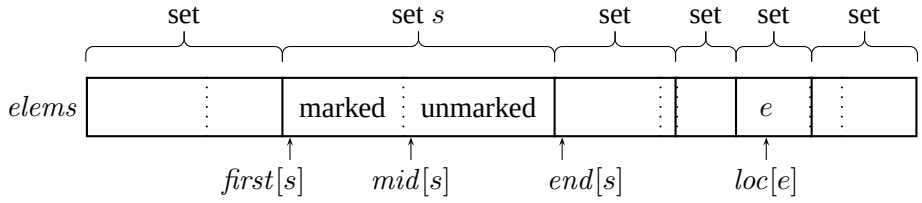

Figure 3. Illustrating the refinable partition data structure.

The partition is refinable, meaning that it is possible to replace any A_s by two sets, provided that they are nonempty and pairwise disjoint, and their union is A_s. This operation is called *splitting*. To indicate which elements go into which subset of A_s, elements may be *marked* or left *unmarked* before splitting A_s. To discuss this, let $\breve{A}_s$ denote the set of marked elements of A_s. Initially all elements of A_s are unmarked, that is, $\breve{A}_s = \emptyset$. It is also necessary to be able to scan a set. These services are specified below, after which their implementation is presented and discussed.

$Mark(e)$ is the marking operation. Let A_s be the set that contains the element e. $Mark(e)$ must not be called if e is already marked, that is, $e \in \breve{A}_s$. If e is not already marked, then $Mark(e)$ adds e to $\breve{A}_s$. If all states in A_s were unmarked before the operation, then $Mark(e)$ returns the number s of the set. Otherwise it returns 0. $Mark(e)$ runs in constant time.

$Split(s)$ unmarks all elements in the set A_s. It also splits A_s to $\breve{A}_s$ and $A_s \setminus \breve{A}_s$, except if one of them is empty. One part inherits the old index s, and the other part gets a brand new index. If one part is bigger than the other, then it inherits s. The time consumption of $Split(s)$ is at most proportional to $|\breve{A}_s|$.

Scanning a set accesses each element exactly once, but the order in which they are met is unspecified. While scanning a set, $Mark$ and $Split$ affecting that set must not be executed.

The implementation of the services is illustrated in Figure 3, and shown in Figure 4. The variable $sets$ tells the number of sets. The implementation uses it and the following arrays:

$elems$ contains 1, 2, ..., $items$ in such an order that elements that belong to the same set are adjacent.

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 divides the segment of a set to the subsegments of marked and unmarked elements. Let f and ℓ be as above. Then $\breve{A}_s = \{\, elems[f], \ldots, elems[\, mid[s] - 1 \,] \,\}$ and the unmarked elements are $elems[\, mid[s] \,], \ldots, elems[\ell - 1]$.

loc tells the locations of elements in $elems$. That is, $elems[\, loc[e] \,] = e$ and $loc[\, elems[i] \,] = i$.

$sidx$ maps elements to the indices of the sets that the elements belong to. That is, $e \in A_{sidx[e]}$.

It is easy to initialize the data structure with the partition that consists of one set, using linear time in $items$.

```
     Mark(e)
1    s := sidx[e] ; ℓ := loc[e] ; m := mid[s] ; mid[s] := m + 1
2    elems[ℓ] := elems[m] ; loc[ elems[ℓ] ] := ℓ
3    elems[m] := e ; loc[e] := m
4    if m = first[s] then return s else return 0

     Split(s)
1    if mid[s] = end[s] then mid[s] := first[s]
2    if mid[s] = first[s] then return
3    sets := sets + 1
4    if mid[s] − first[s] < end[s] − mid[s] then
5        first[sets] := first[s] ; end[sets] := mid[s] ; first[s] := mid[s]
6    else
7        end[sets] := end[s] ; first[sets] := mid[s]
8        end[s] := mid[s] ; mid[s] := first[s]
9    mid[sets] := first[sets]
10   for ℓ ∈ { first[sets], ..., end[sets] − 1 } do sidx[ elems[ℓ] ] := sets

     Scanning a set
1    for ℓ := first[s] to end[s] − 1 do
2        process elems[ℓ]
```

Figure 4. Implementation of the services of the refinable partition data structure.

$Mark(e)$ swaps the element e with the first unmarked element, and moves the borderline between marked and unmarked elements one location forward. The loc array is updated according to the new locations of the swapped elements. If $mid[s]$ was in its initial position before the operation, then $Mark(e)$ returns the number of the set, and otherwise it returns 0. The running time is obviously $O(1)$.

In Section 4, the numbers of the sets with marked elements will be collected into a workset for later use. Because the marking operation returns the number of the set only if the set did not already contain marked elements, each number will be returned only once until splitting. Therefore, it is trivial to ensure that the same number will not be stored twice to the workset. In [12, 14], the marking operations did not return a value. Instead, there were operations with which the number of the set of the element can be obtained and the presence of marked elements can be tested. The present solution is simpler.

The marking operations of [12, 14] contained a test that made them terminate immediately if e is already marked. The test has been removed in this article because the main algorithm never tries to mark the same element more than once before calling the splitting operation.

If all elements of the set s are marked, then line 1 of $Split$ unmarks them, leading to termination on line 2. If no element is marked, then $Split$ just terminates on line 2. On line 3, a new set number is taken into use. If the part consisting of the marked elements is smaller than the other part, then lines 5, 9, and 10 make a new set of it. The new set gets the new set number. The set indices of its elements are updated on line 10. In the opposite case, the unmarked elements become the new set on lines 7 to 10.

The running time of $Split$ is proportional to the size of the smaller part. This is at most the same as the size of the marked part, that is, $|\breve{A}_s|$. Because that much time has been spent in the calls of the

marking operations since the previous call of $Split$, $Split$ can be treated as constant time in the analysis of the time consumption of any algorithm that uses it. In other words, $Split$ runs in amortized constant time.

The implementation of $Split$ is more complicated than the splitting operations in [12,14]. The reason is that the present implementation always makes a biggest part inherit the old set number. We will see in Subsection 4.4 that this convention facilitates a nice memory-saving simplification to the main algorithm. In [12], there were two different marking and splitting operations. Following [11], the present article has only one of each.

4. The Algorithm

In this section the new bisimilarity minimization algorithm is described. When discussing complexity, but not elsewhere, we assume that $n \leq 2m$.

4.1. Input and Additional Notation

Please recall from Section 2 that the algorithm works on a labelled directed graph (S, L, Δ), where $\Delta \subseteq S \times L \times S$, together with an initial partition $\mathcal{I}$ of S. 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 $m = |\Delta|$, that is, there are m transitions. Let $\mathcal{I} = \{S_1, S_2, \ldots, S_k\}$. The input to the algorithm consists of n, α, Δ, and S_2, S_3, …, S_k. The set S_1 need not be given, because it is $S \setminus (S_2 \cup S_3 \cup \cdots \cup S_k)$.

Let $a \in L$ and $B \subseteq S$. We define $\Delta_{a,B} = \Delta \cap (S \times \{a\} \times B)$ 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 $\Delta_{s,a,B}$.

Where possible, we will systematically use s, s', etc., for variables in the pseudocode that store state numbers, t for transitions, a for labels, b for blocks, c for clusters, and ℓ and i for locations that do not clearly correspond to any particular entity.

4.2. Data Structures

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, 2, …, 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 in Section 3. As a matter of fact, either $first$ or end is not needed, because transitions can be sorted so that $end[s] = first[s+1]$, when $1 \leq s < n$. This data structure can be initialized in $\Theta(m+n) = \Theta(m)$ time with counting sort using $head[t]$ as the key.

For convenience, confusing the transitions with their indices will be allowed in formulae, like in

$$In\_transitions[s] \;=\; \{\, (s_1, a, s_2) \in \Delta \mid s_2 = s \,\} \;\; .$$

The algorithm uses the following two refinable partition data structures.

$Blocks$ is a refinable partition data structure on $\{1, 2, \ldots, n\}$. 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. It would be

conceptionally simplest to initialize $Blocks$ to $\{S\}$ towards the beginning of the algorithm. However, as will be discussed in detail in Subsection 4.3, $Blocks$ is initialized in a rather late stage and directly according to the initial partition. Until then, $Blocks.sidx$ stores information on the initial partition, and the other arrays of $Blocks$ have not yet been allocated memory. This reduces the peak memory consumption of the algorithm. Other data structures are initialized as if $Blocks$ were $\{S\}$.

$Clusters$ is a refinable partition data structure on $\{1, 2, \ldots, m\}$. When $Clusters$ has been fully updated, then each set in it consists of the indices of the a-labelled input transitions of some block B, for some label a. That is, $Clusters = \{\, \Delta_{a,B} \mid a \in L \wedge B \in Blocks \wedge \Delta_{a,B} \neq \emptyset \,\}$. However, the updating of $Clusters$ may lag behind the splitting of blocks, so that each cluster is of the form $\Delta_{a,B}$ where B is the union of some blocks. We will discuss this in detail in Subsection 4.4. $Clusters$ is initialized to the value $\{\, \Delta_{a,S} \mid a \in L \wedge \Delta_{a,S} \neq \emptyset \,\}$. This is nontrivial and will be discussed in Subsection 4.3.

For keeping track of work to be done, the algorithm uses one *workset* called $Touched\_items$. Its capacity, that is, the maximum number of elements it can store, is $\max\{n, m\}$. It is initially empty. In the prototype implementation, a stack was used as the workset. However, it need not be a stack. It suffices that it provides 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. (Three worksets were used in [14], four in [12], and two in [11]. The last one also had at least two other arrays or lists for temporary storage that do not have direct counterparts in [12, 14] and this article.)

Finally, the algorithm uses two arrays of counters and one array of links to counters. They are adapted from [11], and are used to implement a tricky detail in three-way splitting. One array of counters, s_count, is indexed by state numbers, and is thus of size n. It is only used for temporary storage. The other two arrays store long-term information. In Section 4.4 we will introduce the notion of "outset". The full definition will be given there, but let us tell now that each outset is a nonempty set of the form $\Delta_{s,a,U}$, where $s \in S$, $a \in L$, and $U \subseteq S$. Initially $U = S$. Each transition belongs to precisely one outset. For each outset $\Delta_{s,a,U}$, there is a counter in l_count that stores the value $|\Delta_{s,a,U}|$. Clearly at most m such counters are needed. To access them, if t is a transition such that $tail[t] = s$, $name[t] = a$, and $head[t] \in U$, then $link[t]$ is the index of the counter that stores $|\Delta_{s,a,U}|$. Again, how this property is initialized and maintained will be discussed in later subsections.

4.3. Initialization

The initialization of $Blocks$ is shown in Figure 5, assuming $n > 0$. Line 1 represents the reading of the initial partition from the input. Because S_1 is not given in the input, $Blocks.sidx$ must be initialized to all 1 before the reading. Lines 2 and 3 sort the states so that the states in the same initial block are adjacent. The sorting takes $O(n \log n)$ time, which is not more than the promised $O(m \log n)$ time consumption of the algorithm as a whole. Line 4 initializes $Blocks.loc$ in $\Theta(n)$ time. Lines 5 to 13 initialize the remaining arrays in $\Theta(n)$ time by scanning the states, recognizing the places where a block changes, and setting the block borders accordingly. Line 7 uses **for** ℓ := 2 **to** ... instead of **for** $\ell \in \ldots$ to emphasize that the order of scanning is important. After executing $Initialize\_blocks$, the partition in $Blocks$ is the initial partition. We have shown the following.

Lemma 4.1. $Initialize\_blocks$ makes $Blocks = \mathcal{I}$ and runs in $O(n \log n)$ time.

```
     Initialize_blocks
 1   for s ∈ S do Blocks.sidx[s] := the number of the initial block of state s
 2   for s ∈ S do Blocks.elems[s] := s
 3   sort Blocks.elems in O(n log n) time using Blocks.sidx[ Blocks.elems[ℓ] ] as the key
 4   for ℓ ∈ {1, 2, ..., n} do Blocks.loc[ Blocks.elems[ℓ] ] := ℓ
 5   Blocks.first[1] := 1 ; Blocks.mid[1] := 1 ; Blocks.sets := 1
 6   set_1 := 1
 7   for ℓ := 2 to n do
 8       set_2 := Blocks.sidx[ Blocks.elems[ℓ] ]
 9       if set_1 ≠ set_2 then
10           Blocks.end[Blocks.sets] := ℓ ; Blocks.sets := Blocks.sets + 1
11           Blocks.first[Blocks.sets] := ℓ ; Blocks.mid[Blocks.sets] := ℓ
12           set_1 := set_2
13   Blocks.end[Blocks.sets] := n + 1
```

Figure 5. Initialization of $Blocks$.

We will see later in this subsection that it may be advantageous to postpone the execution of lines 2 to 13 to a much later stage than line 1.

$Clusters$ must be initialized as follows:

$$Clusters \quad := \quad \{\, \Delta_{a,S} \mid a \in L \wedge \Delta_{a,S} \neq \emptyset \,\} \ .$$

To initialize $Clusters$, transitions with the same label must be put next to each other in its $elems$. Unfortunately, this is not trivial. The promised $O(m \log n)$ running time of the algorithm as a whole does not suffice for sorting the transitions according to their labels with such algorithms as heapsort or counting sort, because they take $\Theta(m \log m)$ or $\Theta(m + \alpha)$ time on the average. The known sorting algorithms that guarantee better performance make some special assumptions on the input, which do not necessarily hold in our case.

Our solution to this problem is based on the fact that transitions need not be *sorted*, it suffices that they are *grouped*. It is not necessary that the groups are in any particular order. Transitions can be grouped in $\Theta(m)$ time on the average by putting them in a hash table of size $\approx m$ using their labels as the keys, and then sorting the hash lists according to the labels. Because (at least in theory) all transitions may fall into the same hash list, the worst case running time is still $\Theta(m \log m)$. Therefore, although the use of a hash table is a practical solution, it only guarantees the promised running time on the average.

If an array of size α can be allocated, then transitions can be grouped in $\Theta(m)$ time by a method that will be discussed soon. We will call this array $index$. It need not be initialized, not even to all zeros. This is important, because initialization would consume $\Theta(\alpha)$ time, jeopardizing the promised performance. The asymptotic memory consumption of the algorithm as a whole becomes $\Theta(m+n+\alpha) = \Theta(m+\alpha)$.

The memory reserved by $index$ can be released after grouping, to be used later for other purposes. Although this does not affect the asymptotic measure, it reduces the maximal memory consumption measured in bytes. The less memory is reserved for other purposes before $index$ is released, the better. In our method $index$ can be released when memory has been only allocated for $tail$, $label$, $head$, $Clusters.elems$, $Clusters.end$, and also for $Blocks.sidx$, if the blocks of the initial partition come be-

```
    Group_transitions
1   reserve α words of memory for index
2   Clusters.sets := 0
3   for t ∈ {1, 2, ..., m} do
4       a := label[t] ; i := index[a]
5       if i < 1 or i > Clusters.sets or found_labels[i] ≠ a then
6           Clusters.sets := Clusters.sets + 1 ; i := Clusters.sets
7           found_labels[i] := a ; index[a] := i ; count[i] := 1
8       else count[i] := count[i] + 1
9   Clusters.end[1] := 1
10  for c := 1 to Clusters.sets − 1 do
11      Clusters.end[c + 1] := Clusters.end[c] + count[c]
12  for t ∈ {1, 2, ..., m} do
13      i := index[ label[t] ] ; ℓ := Clusters.end[i]
14      Clusters.elems[ℓ] := t ; Clusters.end[i] := ℓ + 1
15  release the memory of index
```

Figure 6. Initialization of $Clusters.sets$, $Clusters.elems$ and $Clusters.end$.

fore the transitions in the input. In our prototype implementation, 1 array of size n and 5 arrays of size m are reserved before using $index$, and 7 arrays of size n, 7 of size m, 1 of size $\max\{n, m\}$, and 1 of size $|\mathcal{I}|$ are reserved after using $index$. Therefore, if memory is circulated well, the use of $index$ does not increase memory consumption if $\alpha \leq 7m + 7n$. If $\alpha > 7m$, more than 85 % of the labels are unused. So the memory used by $index$ is usually not significant in practice.

The method is shown in Figure 6. It resembles counting sort, but the counting stage is done differently on lines 2 to 8. These lines are based on [1, Exercise 2.12]. The problem of grouping the transitions and the same solution were briefly discussed in [14].

Lemma 4.2. $Group\_transitions$ makes $Clusters = \{\, \Delta_{a,S} \mid a \in L \wedge \Delta_{a,S} \neq \emptyset \,\}$. It runs in $\Theta(m)$ time and $\Theta(m + \alpha)$ memory.

Proof:
On lines 2 to 8, the number of different labels of the transitions is counted with $Clusters.sets$, the different labels are collected to the array $found\_labels$, and for each $1 \leq i \leq Clusters.sets$, the number of times the label $found\_labels[i]$ occurs is counted with $count[i]$. For each found label a, $index[a]$ tells its location in $found\_labels$. If the label a investigated on line 4 has been found, line 4 assigns its position in $found\_labels$ to i, all tests on line 5 fail, and line 8 increments the counter for that label. If a has not yet been found, then i gets an arbitrary value from an uninitialized slot of $index$. If that value points outside the used part of $found\_labels$, then the first or second test on line 5 matches, and the excution continues on line 6 without doing the third test. Otherwise the third test matches. In both cases, the label is added to $found\_labels$, $index[a]$ is set to point to it, and its counter is set to 1.

Lines 9 to 11 compute the intended starting point of each nonempty $\Delta_{a,S}$ in $Clusters.elems$. Lines 12 to 14 put the transitions into their places in $Clusters.elems$, and increment the $Clusters.end[i]$ so that they become the end points and not the start points of each nonempty $\Delta_{a,S}$. □

```
     Main_loop
1    current_cluster := 1 ;  current_block := 2
2    while current_cluster ≤ Clusters.sets do
3       Split_blocks( current_cluster )
4       current_cluster := current_cluster + 1
5       while current_block ≤ Blocks.sets do
6          Split_clusters( current_block )
7          current_block := current_block + 1
```

Figure 7. Main loop of the bisimilarity minimization algorithm.

To save memory, $found\_labels$ is the same array as $Clusters.end$, and $count$ is the same array as $Clusters.elems$. The names $found\_labels$ and $count$ were only used to improve the readability of the code.

Memory for all the remaining arrays can be reserved now as needed.

The next problem is to initialize the arrays $link$ and l_count as expected by the main loop in Subsection 4.4. A counter holding $|\Delta_{s,a,S}|$ must be made for each nonempty $\Delta_{s,a,S}$, and the link of each $t \in \Delta_{s,a,S}$ must be made to point to the counter. To prepare for that, the prototype implementation sorts each nonempty $\Delta_{a,S}$ within $Clusters.elems$ according to the start states of the transitions, using heapsort. Let $m_a = |\Delta_{a,S}|$. Each sorting operation takes $O(m_a \log m_a)$ time. Because $m_a \leq n^2$ we have $\log m_a \leq 2 \log n$ and $\sum_{a \in L} m_a \log m_a \leq \sum_{a \in L} m_a (2 \log n) = 2m \log n$, so the sorting operations do not violate the $O(m \log n)$ time bound. Alternatively, we could have sorted the transitions according to their start states with counting sort before the grouping in $\Theta(m + n) = \Theta(m)$ time.

Because of the sorting made above, the transitions of each $\Delta_{s,a,S}$ are adjacent. To initialize $link$ and l_count, it is now easy to scan $Clusters.elems$ in $\Theta(m)$ time, introducing a new l_count-counter each time when $label[t]$ or $tail[t]$ changes, making $link[t]$ point to the current counter, and incrementing the current counter for each transition t. The counters s_count are initialized to zeros in $\Theta(n)$ time.

The rest of $Initialize\_blocks$ can be executed now. Also all the remaining data structures can be initialized. Their initialization does not require special techniques.

4.4. The Main Loop

We now concentrate on the main loop. It is shown in Figure 7.

The main loop consists of two splitting operations that are executed repeatedly until nothing splits any more. Line 3 splits blocks according to one cluster, and line 6 splits clusters according to one block. These operations will be discussed in detail below. The purpose of lines 1, 2, 4, 5, and 7 is to ensure that, with one exception, each cluster and each but one block is used for splitting precisely once. That this suffices will be proven rigorously later in this subsection. Intuitively, the reason is that when a block has been used for splitting and then is split to subblocks, it suffices to use all but one of the subblocks for further splitting, and similarly with clusters. This observation was already used by Hopcroft [7]. The exception mentioned above is that if there are no clusters at all, then the algorithm fails to use blocks for splitting. However, that is not a problem, because then the blocks have nothing to split.

```
    Split_clusters( current_block )
1   for s ∈ Blocks( current_block ) do
2       for t ∈ In_transitions[s] do
3           c := Clusters.Mark(t) ; if c ≠ 0 then Touched_items.Add(c)
4   while ¬Touched_items.Empty do Clusters.Split( Touched_items.Remove )
```

Figure 8. Using a block to split clusters.

One initial block but no initial clusters may be skipped in the splitting. The reason for this difference is that every transition has an end state in some block, but not necessarily every state has an output transition with a given label in some cluster. Experiments with the prototype implementation have demonstrated that no cluster and no more than one block may be skipped.

The body of the main loop splits first with one unused cluster and then with all unused blocks that exist at the time. There is nothing deep in this precise structure of the main loop. The only important thing is that when the main loop stops, there are no unused blocks or clusters except what was mentioned above. The chosen structure ensures this without unnecessary tests. In [12, 14], after splitting each block, the appropriate subblocks were immediately used for splitting clusters, without waiting until the $Touched\_items$ in Figure 9 is empty. As a consequence, blocks and clusters had to have an own $Touched\_items$ each. The present algorithm survives with one $Touched\_items$, saving memory.

The proof of the promised performance trusts that each subblock or subcluster that will be used in the future is at most half as big as the block or cluster from which it was split. If any subset is larger than that, then the splitting operation in Section 3 makes it inherit the number of the original set. Therefore, the right subblocks and subclusters may be chosen simply by using the block and cluster numbers in sequence. The main loop does that. This is a simplification compared to earlier work, where there was an explicit data structure for keeping track of (compound) blocks, clusters, or splitters that have to be used in the future, and tests to choose the right item to be put into the set [1, 3, 5–7, 9, 11–14].

$Split\_clusters$ is shown in Figure 8. In the figure, "**for** $s \in Blocks(\ current\_block\ )$ **do**" denotes the scanning of the states of the current block as was shown in Figure 4. The effect and operation of $Split\_clusters$ are described in the following lemma and its proof.

Lemma 4.3. If B is the current block, then $Split\_clusters$ replaces every cluster $\Delta_{a,U}$ by those of $\{\,(s,a,s') \in \Delta_{a,U} \mid s' \in B\,\}$ and $\{\,(s,a,s') \in \Delta_{a,U} \mid s' \notin B\,\}$ that are nonempty.

Proof:
Line 1 scans the states in the block and line 2 scans their input transitions. Each input transition is marked in its cluster. As was discussed in Section 3, the marking operation returns the number of the cluster if and only if it did not already contain marked transitions, otherwise it returns 0. Therefore, the **if**-statement on line 3 adds the number of each cluster that contains marked transitions precisely once to $Touched\_items$. Line 4 discharges $Touched\_items$ and splits the clusters whose numbers were collected to it. □

(Preventing cluster numbers from being added twice is actually unnecessary. It only saves a little of time. $Touched\_items$ is big enough to store also the duplicates and $Split$ does not change anything, if its parameter does not have marked elements.)

```
    Split_blocks( current_cluster )
1   for t ∈ Clusters( current_cluster ) do s_count[ tail[t] ] := s_count[ tail[t] ] + 1

    /* Extract left blocks */
2   for t ∈ Clusters( current_cluster ) do
3       s := tail[t] ; i := link[t]
4       if s_count[s] = l_count[i] then
5           b := Blocks.Mark(s) ; if b ≠ 0 then Touched_items.Add(b)
6           s_count[s] := 0
7   while ¬Touched_items.Empty do Blocks.Split( Touched_items.Remove )

    /* Extract middle blocks */
8   for t ∈ Clusters( current_cluster ) do
9       s := tail[t] ; i := link[t]
10      if s_count[s] < 0 then
11          link[t] := −s_count[s]
12      else if s_count[s] > 0 then
13          b := Blocks.Mark(s) ; if b ≠ 0 then Touched_items.Add(b)
14          lcounters := lcounters + 1
15          l_count[ lcounters ] := s_count[s] ; l_count[i] := l_count[i] − s_count[s]
16          s_count[s] := −lcounters ; link[t] := lcounters
17  for t ∈ Clusters( current_cluster ) do s_count[ tail[t] ] := 0
18  while ¬Touched_items.Empty do Blocks.Split( Touched_items.Remove )
```

Figure 9. Using a cluster to split blocks.

Split_blocks, shown in Figure 9, implements a much more complicated operation. To discuss it, we must introduce the notions of *compound clusters* and *outsets*.

In this article, and unlike in [11, 12], compound clusters are only a conceptual tool. Nothing in the algorithm corresponds directly to them. A compound cluster is always a union of ordinary clusters whose label components are the same. To simplify the discussion, we do not require that a compound cluster is nonempty. Initially the compound clusters are the sets $\Delta_{a,S}$ for each a. When a cluster that is a subset of a compound cluster is split, both parts remain subsets of the compound cluster. When a cluster is used for splitting blocks in *Split_blocks*, it is subtracted from its compound cluster and becomes a compound cluster on its own right. This is the only way in which compound clusters change. It follows from this that each cluster is a subset of precisely one compound cluster and each transition belongs to precisely one compound cluster.

An outset is any nonempty $\Delta_{s,a,U}$ where s is a state and $\Delta_{a,U}$ is a compound cluster. Each transition belongs to precisely one outset.

The operation of *Split_blocks* is captured by the following lemma.

Lemma 4.4. Let C be the current cluster, and let $\widehat{C}$ be the compound cluster such that $C \subseteq \widehat{C}$. *Split_blocks* replaces $\widehat{C}$ by C and $\widehat{C} \setminus C$, and maintains the following "counter invariant" property:

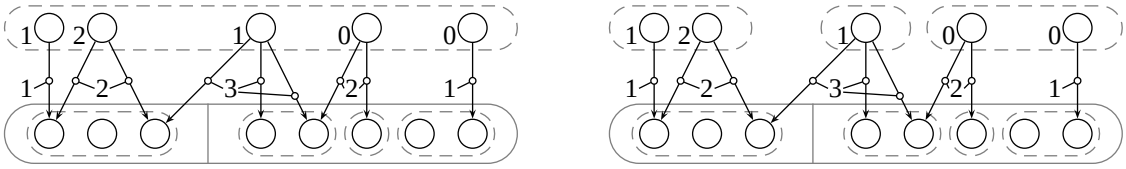

Figure 10. Illustrating three-way splitting with counters.

For each $t \in \Delta$, $l\_count[\, link[t]\,]$ holds the value $|\Delta_{s,a,U}|$, where $\Delta_{s,a,U}$ is the outset that contains t. Furthermore, if also $t' \in \Delta_{s,a,U}$, then $link[t] = link[t']$.

It also splits each block B to those of the following three parts that are nonempty:

Left block:	$\{\, s \in B \mid \exists t \in C : tail[t] = s \wedge \nexists t' \in \widehat{C} \setminus C : tail[t'] = s \,\}$
Middle block:	$\{\, s \in B \mid \exists t \in C : tail[t] = s \wedge \exists t' \in \widehat{C} \setminus C : tail[t'] = s \,\}$
Right block:	$\{\, s \in B \mid \nexists t \in C : tail[t] = s \,\}$

Proof:
That $\widehat{C}$ is replaced by C and $\widehat{C} \setminus C$ is immediate from the definition of compound clusters.

Initially the compound clusters are the $\Delta_{a,S}$, so the outsets are the nonempty $\Delta_{s,a,S}$. In Section 4.3 the counters and links were initialized so that there is a counter for each nonempty $\Delta_{s,a,S}$, it contains the value $|\Delta_{s,a,S}|$, and, if $t \in \Delta_{s,a,S}$, then $link[t]$ points to the counter of $\Delta_{s,a,S}$. Thus the counter invariant holds initially.

Let $C = \Delta_{a,V}$ and $\widehat{C} = \Delta_{a,U}$. All counters s_count hold initially 0 and line 17 resets them after use, so they hold 0 when $Split\_blocks$ is started. Line 1 makes $s\_count[s] = |\Delta_{s,a,V}|$ for each $s \in S$. Because the counter invariant held beforehand, $l\_count[\, link[t]\,] = |\Delta_{s,a,U}|$ for each $t \in C$, where $\Delta_{s,a,U}$ is the outset that contains t. Figure 10 illustrates the situation. U consists of all and V consists of the three leftmost states in the bottom row. Dashed lines show the blocks, and a is not shown. The numbers are the values of the counters.

The s on line 3 or 9 clearly belongs to the left or middle block of its block.

If s belongs to the left block, then $|\Delta_{s,a,U\setminus V}| = 0$, so $|\Delta_{s,a,V}| = |\Delta_{s,a,U}| = l\_count[\, link[t]\,]$. If s has not yet been processed, then lines 4 to 6 detect this situation, mark s, add the number of the block of s to $Touched\_items$ if necessary, and reset $s\_count[s]$ so that s will not be processed a second time. Line 7 splits each nonempty left block apart from the rest, provided that also the rest is not empty.

If s belongs to the middle block, then $|\Delta_{s,a,U\setminus V}| > 0$. If s has not yet been processed, then lines 13 to 16 are executed. First they mark s and keep track of the block number. Then they create a new counter, make t point to it, and assign $|\Delta_{s,a,V}|$ to it. It becomes the counter of $\Delta_{s,a,V}$. The old counter of t continues as the counter of $\Delta_{s,a,U\setminus V}$. Its value is changed to $|\Delta_{s,a,U}| - |\Delta_{s,a,V}| = |\Delta_{s,a,U\setminus V}|$. Due to the assignment $s\_count[s] := -lcounters$, when other transitions that belong to $\Delta_{s,a,V}$ are detected later, lines 10 and 11 re-direct their links to point to the new counter of $\Delta_{s,a,V}$. In this way the links and counters are updated to match the splitting of $\widehat{C}$ to C and $\widehat{C} \setminus C$. States that belong to the right block remain unmarked. Line 18 separates the nonempty middle blocks from the nonempty right blocks.

As a consequence, the counter invariant is re-established for outsets that correspond to states in middle blocks. Outsets that correspond to states in left and right blocks are not split, so no manipulation of their links and counters is needed. Furthermore, blocks are split as promised. □

Let a block be called *ready* if and only if its number is less than $current\_block$, and similarly with clusters and $current\_cluster$. To discuss the overall operation of the main loop, six invariants are used.

Lemma 4.5. The following hold each time when entering line 2 of the main loop.

1. If two transitions are in different clusters, then they have different labels or end in different blocks.
2. Transitions that have different labels are in different clusters.
3. If two transitions end in different blocks, then they are in different clusters or at least one of the blocks is not ready.
4. If two states are in different blocks, then they were in different blocks initially or one of them has an output transition that belongs to a cluster that does not contain any output transition of the other.
5. States which were in different blocks initially are always in different blocks.
6. For each compound cluster $\widehat{C}$, at most one cluster in it is ready. If $\widehat{C}$ has a ready cluster, then, for each block, either every or no state in the block has an output transition in $\widehat{C}$.

Proof:
The "then"-parts of Invariants 1 and 4 list all situations where the algorithm, initialization included, may put states to different blocks and transitions to different clusters. When a block is split, the states in the left and middle block have an output transition in C, but the states in the right block do not have. The states in the middle block also have an output transition in some cluster of $\widehat{C} \setminus C$, but the states in the left block do not have. The "then"-parts cannot become invalid after they have become valid, because blocks are not merged, the same holds for clusters, and the labels, start states and end states of transitions are not changed. Therefore, Invariants 1 and 4 hold.

Invariant 2 is made to hold in the initialization and cannot be invalidated afterwards. Invariant 5 is obvious.

Invariant 3 holds initially, because then only one block is ready, namely block 1. Later on, the "if"-part may only become valid when a block is split. Then at least the subblock that gets a new block number will not be ready, making the "then"-part valid. The only way in which an unready block can become ready is that it is used for splitting. Splitting separates the transitions that end in the block in question from those that lead elsewhere, so the invariant remains valid.

Invariant 6 holds initially, because then no cluster is ready. When an unready cluster is split, both subclusters will be unready, and when a ready cluster is split, one subcluster will be ready and the other unready. So the number of ready clusters in $\widehat{C}$ does not change, and the invariant remains valid. The only way in which $\widehat{C}$ changes is that it is divided to C and $\widehat{C} \setminus C$, where C is a cluster. C becomes a compound cluster that consists of only one cluster. C becomes ready but has just been used for splitting, so the invariant holds for it. $\widehat{C} \setminus C$ inherits the number of ready clusters from $\widehat{C}$. If $\widehat{C} \setminus C$ has a ready cluster, then also $\widehat{C}$ had. If no state of block B had an output transition in $\widehat{C}$, then no state of B has an output transition in $\widehat{C} \setminus C$. If every state of B had an output transition in $\widehat{C}$, then no state of the left subblock of B and every state of the middle and right subblocks of B has an output transition in $\widehat{C} \setminus C$. So $\widehat{C} \setminus C$ satisfies the invariant. A cluster becomes ready only in the case that was just discussed. □

Using the previous invariants, we show next that the algorithm does not split too much but does split as much as needed.

Lemma 4.6. If the algorithm puts two states into different blocks, then they are in different blocks in every partition that is a refinement of $\mathcal{I}$ and compatible with Δ.

Proof:
To derive a contradiction, consider the first moment in time when the algorithm puts two states into different blocks although there is a partition $\mathcal{B}$ that is a refinement of $\mathcal{I}$ and compatible with Δ where they are in the same block. Because they are in the same block of a refinement of $\mathcal{I}$, they were in the same block initially. By Invariant 4, one of them, say s_1, has an output transition (s_1, a, s'_1) that belongs to a cluster C that does not contain transitions that start at the other state, say s_2. By compatibility, there is a transition (s_2, a, s'_2) such that s'_1 and s'_2 are in the same block of $\mathcal{B}$. By Invariant 1, s'_1 and s'_2 are not in the same block of the algorithm. This is in contradiction with the assumption with which we started. So the lemma holds. □

Lemma 4.7. When the algorithm terminates, the partition is a refinement of $\mathcal{I}$ and compatible with Δ.

Proof:
The resulting partition is a refinement of $\mathcal{I}$ by Invariant 5. To prove that it is compatible with Δ, assume that s_1 and s_2 are in the same block, and $s_1 -a\rightarrow s'_1$. Consider the cluster C that contains this transition and the compound cluster $\widehat{C}$ that has C as a subset. When the algorithm terminates, all clusters are ready. By Invariant 6, C is the only cluster in $\widehat{C}$ and also s_2 has an output transition $s_2 -b\rightarrow s'_2$ in C. By Invariant 2, $b = a$. By Invariant 3 and the fact that all blocks are ready, s'_1 and s'_2 are in the same block. So the partition is compatible with Δ. □

Also the performance of the main loop must be addressed.

Lemma 4.8. The main loop runs in $O(m \log n)$ time.

Proof:
It is obvious from the main loop that each time when a transition is used anew for splitting a block, it belongs to a cluster whose number is greater than in the previous time. Because *Split* always makes a largest subset inherit the original number, the size of the new cluster is at most half of the size of the cluster in the previous time. By Invariant 2, the size of each cluster is at most n^2. Therefore, when $n > 0$, each transition is used at most $\log_2 n^2 + 1 = 2 \log_2 n + 1$ times in *Split_blocks*. Similar reasoning applies to states and blocks used for splitting clusters in *Split_clusters* yielding an upper bound $\log_2 n + 1$, which also applies to the number of times a transition is used on lines 2 and 3 of *Split_clusters*. The time consumptions of the calls of *Split* are proportional to the times consumed in the preceding marking operations. All other operations are constant time. Therefore, the main loop runs in $O(m \log n)$ time.
□

The memory consumption of the algorithm as a whole is clearly $\Theta(m + \alpha)$. The results in this and the previous subsection imply the following.

Theorem 4.1. The algorithm described in this section finds the coarsest partition of S that is a refinement of $\mathcal{I}$ and compatible with Δ. If $n \leq 2m$, then it runs in $O(m \log n)$ time and $\Theta(m + \alpha)$ memory. The variant that uses hash tables for grouping transitions runs in $O(m \log n)$ time on the average and $\Theta(m)$ memory, assuming that $n \leq 2m$.

4.5. Writing the Output

The writing of the output deserves a brief discussion. Let B and B' be blocks in the final partition, and $a \in L$. The transition $B -a\rightarrow B'$ has to be introduced if and only if there are $s \in B$ and $s' \in B'$ such that $s -a\rightarrow s'$. Because the final partition is compatible with Δ, any state in B can be used for producing the output transitions of B. One state from each block b can be easily found with $Blocks.first[b]$. Because the prototype implementation does not contain a data structure for output transitions of states, it produces all transitions of the result by scanning the original transitions once and testing, whether $tail[t]$ is the first state in its block.

There may be many a-transitions from s to B'. To prevent $B -a\rightarrow B'$ from being introduced more than once, the prototype implementation keeps track with $l\_count[\, link[t]\,]$ whether an a-transition from s to B' has been introduced. This is easy, because all a-transitions from s to B' and only they have the same value in $link[t]$.

In applications, it is often necessary to know from which initial block each final block came. For this purpose, the prototype implementation contains an extra array of size $|\mathcal{I}|$. The end locations of blocks are copied to it immediately after executing $Initialize\_blocks$. The states of each initial block occupy the same segment in $Blocks.elems$ as originally, although they may have been reordered within the segment and divided to smaller blocks. Final blocks can be listed one initial block at a time by scanning $Blocks.elems$, recognizing the change of the initial block with the extra array, and printing the number of the final block each time when encountering a state that is first in its block. Each time when the initial block changes, 0 is printed to separate the lists. Final blocks that correspond to the first initial block need not be listed, because they can be distinguished as the final blocks whose numbers are not in any list.

5. Experiments and Conclusions

The algorithm in this article may look complicated. To a large 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. Algorithm descriptions in research articles (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 article.

A prototype implementation of the algorithm was written in C++ simultaneously with the writing of this article, using the earlier implementation of the conference version [12] as a starting point. The implementation is faithful to the description of the algorithm in this article, except that it has so-called Markov chain state lumping [3, 13] integrated to it. It was tested with more than 200 randomly generated inputs of various sizes and densities. Each input graph and initial partition were 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, with a corresponding change in the order in which the transitions were given in the input. The first output was used as the fourth input.

Table 1. Sample running times.

input				output		time	time	time
n	α	m	$\lvert\mathcal{I}\rvert$	n'	m'	sec	sec	sec
1 000	2	1 000 000	10	10	200	1.3	1.3	1.3
1 000	5	999 890	10	10	500	1.2	1.2	1.2
1 000	10	999 999	10	1 000	999 999	2.0	2.0	2.0
1 000	20	999 800	10	1 000	999 800	1.8	1.8	1.8
1 000	1 000	1 000 000	10	1 000	1 000 000	2.1	2.1	2.1
1 000	10	10 000 000	10	10	1 000	15.2	15.2	15.3
1 000	50	10 000 000	10	10	5 000	13.7	13.7	13.6
1 000	77	9 999 885	10	10	7 700	13.4	13.4	13.4
1 000	78	10 000 000	10	1 000	10 000 000	24.5	24.7	24.7
1 000	80	10 000 000	10	1 000	10 000 000	24.9	24.6	24.8
1 000	100	9 999 986	10	1 000	9 999 986	23.6	23.8	23.7
1 000	1 000	10 000 000	10	1 000	10 000 000	26.0	25.9	26.0
10 000	1	9 999 982	1	1	1	10.9	10.9	10.8
10 000	10	10 000 000	1	1	10	10.0	10.0	10.1
10 000	10	10 000 000	8	8	640	16.6	16.6	16.7
10 000	10	10 000 000	9	9 979	9 978 707	31.1	30.9	30.8
10 000	10	10 000 000	10	9 974	9 973 688	30.4	30.4	30.5

Our prototype implementation has twelve arrays of size m ($tail$, $name$, $head$, one in $In\_transitions$, six in $Clusters$, $link$, and l_count), eight of size n (one in $In\_transitions$, six in $Blocks$, and s_count), one of size $\max\{m, n\}$ ($Touched\_items$), and one of size $|\mathcal{I}|$ (see Subsection 4.5), making altogether $12m + 8n + \max\{m, n\} + |\mathcal{I}| + O(1)$ words of memory. The algorithm in [12] uses 21 arrays of size m, one of size $\lfloor m/2 \rfloor$, nine of size n, and one of size $|\mathcal{I}|$.

The middle states approach of [4] needs $tail$, $name$, $head$, and the array of size $|\mathcal{I}|$ to store the input, together with the arrays needed to solve the problem in the absence of labels. The description of the Paige–Tarjan algorithm in [11] is neither detailed nor optimized enough for a fair comparison. However, an adaptation of our algorithm to the absence of labels yields essentially a memory-optimized version of the Paige–Tarjan algorithm, so let us use it in the comparison. Instead of $Clusters$, it needs a temporary array of size n for taking a copy of the block being used for splitting, to prevent confusion if it splits itself. Furthermore, n slots suffice for $Touched\_items$. The adding of states in the middle of transitions makes the sizes of most arrays grow to $2m$ or $m + n$. The total memory consumption is thus $19m + 10n + |\mathcal{I}| + O(1)$ words.

To give some idea of the practical performance of the algorithm, Table 1 shows some running times on a laptop with 2 GiB of RAM and 1.6 GHz clock rate. Each experiment was done three times and all times are reported. The times do not include the reading of the input and the writing of the output. With big inputs, these were usually clearly longer than the processing time. Because generating high-quality random graphs is not trivial, the testing environment suffers from some restrictions. Sometimes it produces sligthly less transitions than the desired number. This is why m is not always a power of ten in the table. The inequality $n^2\alpha < 2^{31}$ must hold, significantly restricting the maximal n. The sizes of the files containing the input graphs restricted m in the tests.

When $m \gg n\alpha|\mathcal{I}|$, every state has a transition with each label to each initial block, and no block splitting occurs. Otherwise, splitting tends to continue until almost every block contains only one state. Indeed, almost every output in the table is either of the same size as the input or has $|\mathcal{I}|$ states and $\alpha|\mathcal{I}|^2$ transitions. We tried to also report experiments that are in the middle ground, but such parameters were difficult to find, as the cases with $\alpha = 77$ and $\alpha = 78$ or $|\mathcal{I}| = 8$ and $|\mathcal{I}| = 9$ show. Not surprisingly, the running time depends strongly on the size of the output. However, the dependency is not fully straightforward, as the lines with $n = 1\,000$ and $m \approx 10\,000\,000$ show. Altogether, the dependency of the running time on the input parameters is so complicated that it would be a research topic of its own.

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.

Acknowledgements

I thank the reviewers for detailed comments.

References

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

[2] Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science* 59, 115–131 (1988)

[3] Derisavi, S., Hermanns, H., Sanders, W.H.: Optimal State-space Lumping in Markov Chains. *Information Processing Letters* 87(6), 309–315 (2003)

[4] Dovier, A., Piazza, C., Policriti, A.: An Efficient Algorithm for Computing Bisimulation Equivalence. *Theoretical Computer Science* 311, 221–256 (2004)

[5] Fernandez, J.-C.: An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming* 13, 219–236 (1989/90)

[6] Gries, D.: Describing an Algorithm by Hopcroft. *Acta Informatica* 2, 97–109 (1973)

[7] Hopcroft, J.: *An $n \log n$ Algorithm for Minimizing States in a Finite Automaton*. Technical Report STAN-CS-71-190, Stanford University (1971)

[8] Kanellakis, P., Smolka, S.: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In: *2nd ACM Symposium on Principles of Distributed Computing*, 228–240 (1983)

[9] Knuutila, T.: Re-describing an Algorithm by Hopcroft. *Theoretical Computer Science* 250, 333–363 (2001)

[10] Milner, R.: *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, NJ (1989)

[11] Paige, R., Tarjan, R.: Three Partition Refinement Algorithms. *SIAM Journal of Computing* 16(6), 973–989 (1987)

[12] Valmari, A.: Bisimilarity Minimization in $O(m \log n)$ time. In: Franceschinis, G., Wolf, K. (eds.): *Petri Nets 2009, Lecture Notes in Computer Science* 5606, 123–142, Springer, Heidelberg (2009)

[13] Valmari, A., Franceschinis, G.: Simple $O(m \log n)$ Time Markov Chain Lumping. In: Esparza, J., Majumdar, R. (eds.) *TACAS 2010, Lecture Notes in Computer Science* 6015, 38–52, Springer, Heidelberg (2010)

[14] 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, 645–656. http://drops.dagstuhl.de/volltexte/2008/1328/ (2008)