Minimal Separating Sequences for All Pairs of States

Rick Smetsers(✉), Joshua Moerman, and David N. Jansen

Institute for Computing and Information Sciences, Radboud University,
Toernooiveld 212, 6525 EC Nijmegen, The Netherlands
{r.smetsers,joshua.moerman,dnjansen}@cs.ru.nl

Abstract. Finding minimal separating sequences for all pairs of inequivalent states in a finite state machine is a classic problem in automata theory. Sets of minimal separating sequences, for instance, play a central role in many conformance testing methods. Moore has already outlined a partition refinement algorithm that constructs such a set of sequences in $\mathcal{O}(mn)$ time, where m is the number of transitions and n is the number of states. In this paper, we present an improved algorithm based on the minimization algorithm of Hopcroft that runs in $\mathcal{O}(m \log n)$ time. The efficiency of our algorithm is empirically verified and compared to the traditional algorithm.

Keywords: Algorithms on automata and words · Partition refinement

1 Introduction

In diverse areas of computer science and engineering, systems can be modelled by *finite state machines* (FSMs). One of the cornerstones of automata theory is minimization of such machines (and many variation thereof). In this process one obtains an equivalent minimal FSM, where states are different if and only if they have different behaviour. The first to develop an algorithm for minimization was Moore [9]. His algorithm has a time complexity of $\mathcal{O}(mn)$, where m is the number of transitions, and n is the number of states of the FSM. Later, Hopcroft improved this bound to $\mathcal{O}(m \log n)$ [6].

Minimization algorithms can be used as a framework for deriving a set of *separating sequences* that show *why* states are inequivalent. The separating sequences in Moore's framework are of minimal length [3]. Obtaining minimal separating sequences in Hopcroft's framework, however, is a non-trivial task. In this paper, we present an algorithm for finding such minimal separating sequences for all pairs of inequivalent states of a FSM in $\mathcal{O}(m \log n)$ time.

Coincidentally, Bonchi and Pous recently introduced a new algorithm for the equally fundamental problem of proving equivalence of states in non-deterministic automata [1]. As both their and our work demonstrate, even classical problems in automata theory can still offer surprising research opportunities.

Supported by NWO project 628.001.009 on Learning Extended State Machines for Malware Analysis (LEMMA).

A.-H. Dediu et al. (Eds.): LATA 2016, LNCS 9618, pp. 181–193, 2016.
DOI: 10.1007/978-3-319-30000-9_14

Moreover, new ideas for well-studied problems may lead to algorithmic improvements that are of practical importance in a variety of applications.

One such application for our work is in *conformance testing*. Here, the goal is to test if a black box implementation of a system is functioning as described by a given FSM. It consists of applying sequences of inputs to the implementation, and comparing the output of the system to the output prescribed by the FSM. Minimal separating sequences are used in many test generation methods [2]. Therefore, our algorithm can be used to improve these methods.

2 Preliminaries

We define a *FSM* as a Mealy machine $M = (I, O, S, \delta, \lambda)$, where I, O and S are finite sets of *inputs*, *outputs* and *states* respectively, $\delta : S \times I \to S$ is a *transition function* and $\lambda : S \times I \to O$ is an *output function*. The functions δ and λ are naturally extended to $\delta : S \times I^* \to S$ and $\lambda : S \times I^* \to O^*$. Moreover, given a set of states $S' \subseteq S$ and a sequence $x \in I^*$, we define $\delta(S', x) = \{\delta(s, x) | s \in S'\}$ and $\lambda(S', x) = \{\lambda(s, x) | s \in S'\}$. The *inverse transition function* $\delta^{-1} : S \times I \to \mathcal{P}(S)$ is defined as $\delta^{-1}(s, a) = \{t \in S | \delta(t, a) = s\}$. Observe that Mealy machines are deterministic and input-enabled (i.e. complete) by definition. The initial state is not specified because it is of no importance in what follows. For the remainder of this paper we fix a machine $M = (I, O, S, \delta, \lambda)$. We use n to denote its number of states, i.e. $n = |S|$, and m to denote its number of transitions, i.e. $m = |S| \times |I|$.

Definition 1. *States s and t are* equivalent *if $\lambda(s, x) = \lambda(t, x)$ for all x in I^*.*

We are interested in the case where s and t are not equivalent, i.e. *inequivalent*. If all pairs of distinct states of a machine M are inequivalent, then M is *minimal*. An example of a minimal FSM is given in Fig. 1.

Definition 2. *a* separating sequence *for states s and t in s is a sequence $x \in i^*$ such that $\lambda(s, x) \neq \lambda(t, x)$. We say x is* minimal *if $|y| \geq |x|$ for all separating sequences y for s and t.*

A separating sequence always exists if two states are inequivalent, and there might be multiple minimal separating sequences. Our goal is to obtain minimal separating sequences for all pairs of inequivalent states of M.

2.1 Partition Refinement

In this section we will discuss the basics of minimization. Both Moore's algorithm and Hopcroft's algorithm work by means of partition refinement. A similar treatment (for DFAs) is given in [4].

A *partition* P of S is a set of pairwise disjoint non-empty subsets of S whose union is exactly S. Elements in P are called *blocks*. If P and P' are partitions of S, then P' is a *refinement* of P if every block of P' is contained in a block of P. A partition refinement algorithm constructs the finest partition under some constraint. In our context the constraint is that equivalent states belong to the same block.

Definition 3. *A partition is* valid *if equivalent states are in the same block.*

Partition refinement algorithms for FSMs start with the trivial partition $P = \{S\}$, and iteratively refine P until it is the finest valid partition (where all states in a block are equivalent). The blocks of such a *complete* partition form the states of the minimized FSM, whose transition and output functions are well-defined because states in the same block are equivalent.

Let B be a block and a be an input. There are two possible reasons to split B (and hence refine the partition). First, we can *split B with respect to output after a* if the set $\lambda(B, a)$ contains more than one output. Second, we can *split B with respect to the state after a* if there is no single block B' containing the set $\delta(B, a)$. In both cases it is obvious what the new blocks are: in the first case each output in $\lambda(B, a)$ defines a new block, in the second case each block containing a state in $\delta(B, a)$ defines a new block. Both types of refinement preserve validity.

Partition refinement algorithms for FSMs first perform splits w.r.t. output, until there are no such splits to be performed. This is precisely the case when the partition is *acceptable*.

Definition 4. *A partition is* acceptable *if for all pairs s, t of states contained in the same block and for all inputs a in I, $\lambda(s, a) = \lambda(t, a)$.*

Any refinement of an acceptable partition is again acceptable. The algorithm continues performing splits w.r.t. state, until no such splits can be performed. This is exactly the case when the partition is stable.

Definition 5. *A partition is* stable *if it is acceptable and for any input a in I and states s and t that are in the same block, states $\delta(s, a)$ and $\delta(t, a)$ are also in the same block.*

Since an FSM has only finitely many states, partition refinement will terminate. The output is the finest valid partition which is acceptable and stable. For a more formal treatment on partition refinement we refer to [4].

2.2 Splitting Trees and Refinable Partitions

Both types of splits described above can be used to construct a separating sequence for the states that are split. In a split w.r.t. the output after a, this sequence is simply a. In a split w.r.t. the state after a, the sequence starts with an a and continues with the separating sequence for states in $\delta(B, a)$. In order to systematically keep track of this information, we maintain a *splitting tree*. The splitting tree was introduced by Lee and Yannakakis [8] as a data structure for maintaining the operational history of a partition refinement algorithm.

Definition 6. *A* splitting tree *for M is a rooted tree T with a finite set of nodes with the following properties:*

- *Each node u in T is labelled by a subset of S, denoted $l(u)$.*
- *The root is labelled by S.*

- *For each inner node u, $l(u)$ is partitioned by the labels of its children.*
- *Each inner node u is associated with a sequence $\sigma(u)$ that separates states contained in different children of u.*

We use $C(u)$ to denote the set of children of a node u. The *lowest common ancestor* (lca) for a set $S' \subseteq S$ is the node u such that $S' \subseteq l(u)$ and $S' \not\subseteq l(v)$ for all $v \in C(u)$ and is denoted by $\text{lca}(S')$. For a pair of states s and t we use the shorthand $\text{lca}(s,t)$ for $\text{lca}(\{s,t\})$.

The labels $l(u)$ can be stored as a *refinable partition* data structure [11]. This is an array containing a permutation of the states, ordered so that states in the same block are adjacent. The label $l(u)$ of a node then can be indicated by a slice of this array. If node u is split, some states in the *slice* $l(u)$ may be moved to create the labels of its children, but this will not change the *set* $l(u)$.

A splitting tree T can be used to record the history of a partition refinement algorithm because at any time the leaves of T define a partition on S, denoted $P(T)$. We say a splitting tree T is valid (resp. acceptable, stable, complete) if $P(T)$ is as such. A leaf can be expanded in one of two ways, corresponding to the two ways a block can be split. Given a leaf u and its block $B = l(u)$ we define the following two splits:

split-output. Suppose there is an input a such that B can be split w.r.t output after a. Then we set $\sigma(u) = a$, and we create a node for each subset of B that produces the same output x on a. These nodes are set to be children of u.

split-state. Suppose there is an input a such that B can be split w.r.t. the state after a. Then instead of splitting B as described before, we proceed as follows. First, we locate the node $v = \text{lca}(\delta(B,a))$. Since v cannot be a leaf, it has at least two children whose labels contain elements of $\delta(B,a)$. We can use this information to expand the tree as follows. For each node w in $C(v)$ we create a child of u labelled $\{s \in B | \delta(s,a) \in l(w)\}$ if the label contains at least one state. Finally, we set $\sigma(u) = a\sigma(v)$.

A straight-forward adaptation of partition refinement for constructing a stable splitting tree for M is shown in Algorithm 1. The termination and the correctness of the algorithm outlined in Sect. 2.1 are preserved. It follows directly that states are equivalent if and only if they are in the same label of a leaf node.

Example 7. Figure 1 shows a FSM and a complete splitting tree for it. This tree is constructed by Algorithm 1 as follows. First, the root node is labelled by $\{s_0, \ldots, s_5\}$. The even and uneven states produce different outputs after a, hence the root node is split. Then we note that s_4 produces a different output after b than s_0 and s_2, so $\{s_0, s_2, s_4\}$ is split as well. At this point T is acceptable: no more leaves can be split w.r.t. output. Now, the states $\delta(\{s_1, s_3, s_5\}, a)$ are contained in different leaves of T. Therefore, $\{s_1, s_3, s_5\}$ is split into $\{s_1, s_5\}$ and $\{s_3\}$ and associated with sequence ab. At this point, $\delta(\{s_0, s_2\}, a)$ contains states that are in both children of $\{s_1, s_3, s_5\}$, so $\{s_0, s_2\}$ is split and the associated sequence is aab. We continue until T is complete.

Input: A FSM M
Result: A valid and stable splitting tree T
initialize T to be a tree with a single node labeled S
repeat
  find $a \in I, B \in P(T)$ such that we can split B w.r.t. output $\lambda(\cdot, a)$
  expand the $u \in T$ with $l(u) = B$ as described in (split-output)
until $P(T)$ *is acceptable*
repeat
  find $a \in I, B \in P(T)$ such that we can split B w.r.t. state $\delta(\cdot, a)$
  expand the $u \in T$ with $l(u) = B$ as described in (split-state)
until $P(T)$ *is stable*

Algorithm 1. Constructing a stable splitting tree

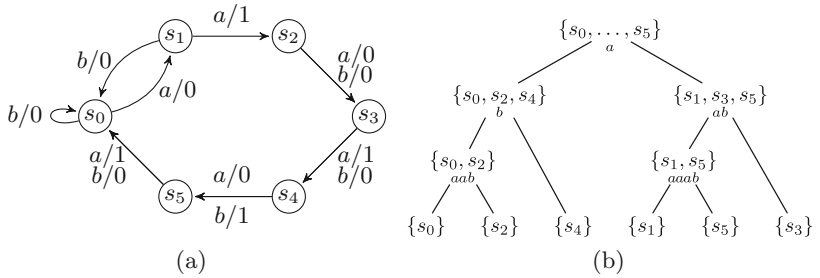


Fig. 1. A FSM (a) and a complete splitting tree for it (b)

3 Minimal Separating Sequences

In Sect. 2.2 we have described an algorithm for constructing a complete splitting tree. This algorithm is non-deterministic, as there is no prescribed order on the splits. In this section we order them to obtain minimal separating sequences.

Let u be a non-root inner node in a splitting tree, then the sequence $\sigma(u)$ can also be used to split the parent of u. This allows us to construct splitting trees where children will never have shorter sequences than their parents, as we can always split with those sequences first. Trees obtained in this way are guaranteed to be *layered*, which means that for all nodes u and all $u' \in C(u)$, $|\sigma(u)| \leq |\sigma(u')|$. Each layer consists of nodes for which the associated separating sequences have the same length.

Our approach for constructing minimal sequences is to ensure that each layer is as large as possible before continuing to the next one. This idea is expressed formally by the following definitions.

Definition 8. *A splitting tree T is* k-stable *if for all states s and t in the same leaf we have $\lambda(s, x) = \lambda(t, x)$ for all $x \in I^{\leq k}$.*

Definition 9. *A splitting tree T is* minimal *if for all states s and t in different leaves $\lambda(s, x) \neq \lambda(t, x)$ implies $|x| \geq |\sigma(\mathrm{lca}(s, t))|$ for all $x \in I^*$.*

Minimality of a splitting tree can be used to obtain minimal separating sequences for pairs of states. If the tree is in addition stable, we obtain minimal separating sequences for all inequivalent pairs of states. Note that if a minimal splitting tree is $(n-1)$-stable (n is the number of states of M), then it is stable (Definition 5). This follows from the well-known fact that $n-1$ is an upper bound for the length of a minimal separating sequence [9].

Algorithm 2 ensures a stable and minimal splitting tree. The first repeat-loop is the same as before (in Algorithm 1). Clearly, we obtain a 1-stable and minimal splitting tree here. It remains to show that we can extend this to a stable and minimal splitting tree. Algorithm 3 will perform precisely one such step towards stability, while maintaining minimality. Termination follows from the same reason as for Algorithm 1. Correctness for this algorithm is shown by the following key lemma. We will denote the input tree by T and the tree after performing Algorithm 3 by T'. Observe that T is an initial segment of T'.

Lemma 10. *Algorithm 3 ensures a $(k+1)$-stable minimal splitting tree.*

Proof. Let us proof stability. Let s and t be in the same leaf of T' and let $x \in I^*$ be such that $\lambda(s,x) \neq \lambda(t,x)$. We show that $|x| > k+1$.

Suppose for the sake of contradiction that $|x| \leq k+1$. Let u be the leaf containing s and t and write $x = ax'$. We see that $\delta(s,a)$ and $\delta(t,a)$ are separated by k-stability of T. So the node $v = \mathrm{lca}(\delta(l(u),a))$ has children and an associated sequence $\sigma(v)$. There are two cases:

- $|\sigma(v)| < k$, then $a\sigma(v)$ separates s and t and is of length $\leq k$. This case contradicts the k-stability of T.
- $|\sigma(v)| = k$, then the loop in Algorithm 3 will consider this case and split. Note that this may not split s and t (it may occur that $a\sigma(v)$ splits different elements in $l(u)$). We can repeat the above argument inductively for the newly created leaf containing s and t. By finiteness of $l(u)$, the induction will stop and, in the end, s and t are split.

Both cases end in contradiction, so we conclude that $|x| > k+1$.

Let us now prove minimality. It suffices to consider only newly split states in T'. Let s and t be two states with $|\sigma(\mathrm{lca}(s,t))| = k+1$. Let $x \in I^*$ be a sequence such that $\lambda(s,x) \neq \lambda(t,x)$. We need to show that $|x| \geq k+1$. Since $x \neq \epsilon$ we can write $x = ax'$ and consider the states $s' = \delta(s,a)$ and $t' = \delta(t,a)$ which are separated by x'. Two things can happen:

- The states s' and t' are in the same leaf in T. Then by k-stability of T we get $\lambda(s',y) = \lambda(t',y)$ for all $y \in I^{\leq k}$. So $|x'| > k$.
- The states s' and t' are in different leaves in T and let $u = \mathrm{lca}(s',t')$. Then $a\sigma(u)$ separates s and t. Since s and t are in the same leaf in T we get $|a\sigma(u)| \geq k+1$ by k-stability. This means that $|\sigma(u)| \geq k$ and by minimality of T we get $|x'| \geq k$.

In both cases we have shown that $|x| \geq k+1$ as required. □

Input: A FSM M with n states
Result: A stable, minimal splitting tree T
initialize T to be a tree with a single node labeled S
repeat
  find $a \in I, B \in P(T)$ such that we can split B w.r.t. output $\lambda(\cdot, a)$
  expand the $u \in T$ with $l(u) = B$ as described in (split-output)
until $P(T)$ *is acceptable*
for $k = 1$ *to* $n - 1$ **do**
  perform Algorithm 3 or Algorithm 4 on T for k

Algorithm 2. Constructing a stable and minimal splitting tree

Input: a k-stable and minimal splitting tree T
Result: T is a $(k + 1)$-stable, minimal splitting tree
forall the *leaves* $u \in T$ *and all inputs* a **do**
  locate $v = \mathrm{lca}(\delta(l(u), a))$
  if v *is an inner node and* $|\sigma(v)| = k$ **then**
    expand u as described in (split-state) (which generates new leaves)

Algorithm 3. A step towards the stability of a splitting tree

Example 11. Figure 2a shows a stable and minimal splitting tree T for the machine in Fig. 1a. This tree is constructed by Algorithm 2 as follows. It executes the same as Algorithm 1 until we consider the node labeled $\{s_0, s_2\}$. At this point $k = 1$. We observe that the sequence of $\mathrm{lca}(\delta(\{s_0, s_2\}, a))$ has length 2, which is too long, so we continue with the next input. We find that we can indeed split w.r.t. the state after b, so the associated sequence is ba. Continuing, we obtain the same partition as before, but with smaller witnesses.

The internal data structure (a refinable partition) is shown in Fig. 2b: the array with the permutation of the states is at the bottom, and every block includes an indication of the slice containing its label and a pointer to its parent (as our final algorithm needs to find the parent block, but never the child blocks).

4 Optimizing the Algorithm

In this section, we present an improvement on Algorithm 3 that uses two ideas described by Hopcroft in his seminal paper on minimizing finite automata [6]: *using the inverse transition set*, and *processing the smaller half*. The algorithm that we present is a drop-in replacement, so that Algorithm 2 stays the same except for some bookkeeping. This way, we can establish correctness of the new algorithms more easily. The variant presented in this section reduces the amount of redundant computations that were made in Algorithm 3.

Using Hopcroft's first idea, we turn our algorithm upside down: instead of searching for the lca for each leaf, we search for the leaves u for which $l(u) \subseteq \delta^{-1}(l(v), a)$, for each potential lca v and input a. To keep the order of splits as before, we define *k-candidates*.

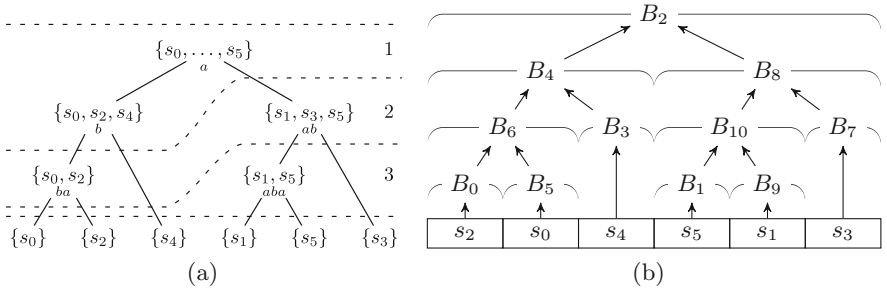


Fig. 2. A complete and minimal splitting tree for the FSM in Fig. 1a (a) and its internal refinable partition data structure (b)

Definition 12. *A* k-candidate *is a node* v *with* $|\sigma(v)| = k$.

A k-candidate v and an input a can be used to split a leaf u if $v = \text{lca}(\delta(l(u), a))$, because in this case there are at least two states s, t in $l(u)$ such that $\delta(s, a)$ and $\delta(t, a)$ are in labels of different nodes in $C(v)$. Refining u this way is called *splitting* u *with respect to* (v, a). The set $C(u)$ is constructed according to (split-state), where each child $w \in C(v)$ defines a child u_w of u with states

$$\begin{aligned} l(u_w) &= \{s \in l(u) \mid \delta(s, a) \in l(w)\} \\ &= l(u) \cap \delta^{-1}(l(w), a) \end{aligned} \tag{1}$$

In order to perform the same splits in each layer as before, we maintain a list L_k of k-candidates. We keep the list in order of the construction of nodes, because when we split w.r.t. a child of a node u before we split w.r.t. u, the result is not well-defined. Indeed, the order on L_k is the same as the order used by Algorithm 2. So far, the improved algorithm still would have time complexity $\mathcal{O}(mn)$.

To reduce the complexity we have to use Hopcroft's second idea of *processing the smaller half*. The key idea is that, when we fix a k-candidate v, all leaves are split with respect to (v, a) simultaneously. Instead of iterating over of all leaves to refine them, we iterate over $s \in \delta^{-1}(l(w), a)$ for all w in $C(v)$ and look up in which leaf it is contained to move s out of it. From Lemma 8 in [7] it follows that we can skip one of the children of v. This lowers the time complexity to $\mathcal{O}(m \log n)$. In order to move s out of its leaf, each leaf u is associated with a set of temporary children $C'(u)$ that is initially empty, and will be finalized after iterating over all s and w.

In Algorithm 4 we use the ideas described above. For each k-candidate v and input a, we consider all children w of v, except for the largest one (in case of multiple largest children, we skip one of these arbitrarily). For each state $s \in \delta^{-1}(l(w), a)$ we consider the leaf u containing it. If this leaf does not have an associated temporary child for w we create such a child (line 9), if this child exists we move s into that child (line 10).

```
   Input: a k-stable and minimal splitting tree T, and a list L_k
   Result: T is a (k+1)-stable and minimal splitting tree, and a list L_{k+1}
 1 L_{k+1} ← ∅
 2 forall the k-candidates v in L_k in order do
 3     let w' be a node in C(v) such that |l(w')| ≥ |l(w)| for all nodes w in C(v)
 4     forall the inputs a in I do
 5         forall the nodes w in C(v) \ w' do
 6             forall the states s in δ^{-1}(l(w), a) do
 7                 locate leaf u such that s ∈ l(u)
 8                 if C'(u) does not contain node u_w then
 9                     add a new node u_w to C'(u)
10                 move s from l(u) to l(u_w)
11         foreach leaf u with C'(u) ≠ ∅ do
12             if |l(u)| = 0 then
13                 if |C'(u)| = 1 then
14                     recover u by moving its elements back and clear C'(u)
15                     continue with the next leaf
16                 set p = u and C(u) = C'(u)
17             else
18                 construct a new node p and set C(p) = C'(u) ∪ {u}
19                 insert p in the tree in the place where u was
20             set σ(p) = aσ(v)
21             append p to L_{k+1} and clear C'(u)
```

Algorithm 4. A better step towards the stability of a splitting tree

Once we have done the simultaneous splitting for the candidate v and input a, we finalize the temporary children. This is done at lines 11–21. If there is only one temporary child with all the states, no split has been made and we recover this node (line 14). In the other case we make the temporary children permanent.

The states remaining in u are those for which $\delta(s, a)$ is in the child of v that we have skipped; therefore we will call it the *implicit child*. We should not touch these states to keep the theoretical time bound. Therefore, we construct a new parent node p that will "adopt" the children in $C'(u)$ together with u (line 16).

We will now explain why considering all but the largest children of a node lowers the algorithm's time complexity. Let T be a splitting tree in which we color all children of each node blue, except for the largest one. Then:

Lemma 13. *A state s is in at most $(\log_2 n) - 1$ labels of blue nodes.*

Proof. Observe that every blue node u has a sibling u' such that $|l(u')| \geq |l(u)|$. So the parent $p(u)$ has at least $2|l(u)|$ states in its label, and the largest blue node has at most $n/2$ states.

Suppose a state s is contained in m blue nodes. When we walk up the tree starting at the leaf containing s, we will visit these m blue nodes. With each

visit we can double the lower bound of the number of states. Hence $n/2 \geq 2^m$ and $m \leq (\log_2 n) - 1$. □

Corollary 14. *A state s is in at most $\log_2 n$ sets $\delta^{-1}(l(u), a)$, where u is a blue node and a is an input in I.*

If we now quantify over all transitions, we immediately get the following result. We note that the number of blue nodes is at most $n - 1$, but since this fact is not used, we leave this to the reader.

Corollary 15. *Let $\mathcal{B}$ denote the set of blue nodes and define*

$$\mathcal{X} = \{(b, a, s) \mid b \in \mathcal{B}, a \in I, s \in \delta^{-1}(l(b), a)\}.$$

Then $\mathcal{X}$ has at most $m \log_2 n$ elements.

The important observation is that when using Algorithm 4 we iterate in total over every element in $\mathcal{X}$ at most once.

Theorem 16. *Algorithm 2 using Algorithm 4 runs in $\mathcal{O}(m \log n)$ time.*

Proof. We prove that bookkeeping does not increase time complexity by discussing the implementation.

Inverse transition. δ^{-1} can be constructed as a preprocessing step in $\mathcal{O}(m)$.

State sorting. As described in Sect. 2.2, we maintain a refinable partition data structure. Each time new pair of a k-candidate v and input a is considered, leaves are split by performing a bucket sort.

First, buckets are created for each node in $w \in C(v) \setminus w'$ and each leaf u that contains one or more elements from $\delta^{-1}(l(w), a)$, where w' is a largest child of v. The buckets are filled by iterating over the states in $\delta^{-1}(l(w), a)$ for all w. Then, a pivot is set for each leaf u such that exactly the states that have been placed in a bucket can be moved right of the pivot (and untouched states in $\delta^{-1}(l(w'), a)$ end up left of the pivot). For each leaf u, we iterate over the states in its buckets and the corresponding indices right of its pivot, and we swap the current state with the one that is at the current index. For each bucket a new leaf node is created. The refinable partition is updated such that the current state points to the most recently created leaf.

This way, we assure constant time lookup of the leaf for a state, and we can update the array in constant time when we move elements out of a leaf.

Largest child. For finding the largest child, we maintain counts for the temporary children and a current biggest one. On finalizing the temporary children we store (a reference to) the biggest child in the node, so that we can skip this node later in the algorithm.

Storing sequences. The operation on line 20 is done in constant time by using a linked list. □

5 Application in Conformance Testing

A splitting tree can be used to extract relevant information for two classical test generation methods: a *characterization set* for the W-method and a *separating family* for the HSI-method. For an introduction and comparison of FSM-based test generation methods we refer to [2].

Definition 17. *A set $W \subset I^*$ is called a* characterization set *if for every pair of inequivalent states s, t there is a sequence $w \in W$ such that $\lambda(s, w) \neq \lambda(t, w)$.*

Lemma 18. *Let T be a complete splitting tree, then $\{\sigma(u) | u \in T\}$ is a characterization set.*

Proof. Let $W = \{\sigma(u) | u \in T\}$. Let $s, t \in S$ be inequivalent states, then by completeness s and t are contained in different leaves of T. Hence $u = lca(s, t)$ exists and $\sigma(u)$ separates s and t. Furthermore $\sigma(u) \in W$. This shows that W is a characterisation set. □

Lemma 19. *A characterization set with minimal length sequences can be constructed in time $\mathcal{O}(m \log n)$.*

Proof. By Lemma 18 the sequences associated with the inner nodes of a splitting tree form a characterization set. By Theorem 16, such a tree can be constructed in time $\mathcal{O}(m \log n)$. Traversing the tree to obtain the characterization set is linear in the number of nodes (and hence linear in the number of states). □

Definition 20. *A collection of sets $\{H_s\}_{s \in S}$ is called a* separating family *if for every pair of inequivalent states s, t there is a sequence h such that $\lambda(s, h) \neq \lambda(t, h)$ and h is a prefix of some $h_s \in H_s$ and some $h_t \in H_t$.*

Lemma 21. *Let T be a complete splitting tree, the sets $\{\sigma(u) | s \in l(u), u \in T\}_{s \in S}$ form a separating family.*

Proof. Let $H_s = \{\sigma(u) | s \in l(u)\}$. Let $s, t \in S$ be inequivalent states, then by completeness s and t are contained in different leaves of T. Hence $u = lca(s, t)$ exists. Since both s and t are contained in $l(u)$, the separating sequence $\sigma(u)$ is contained in both sets H_s and H_t. Therefore, it is a (trivial) prefix of some word $h_s \in H_s$ and some $h_t \in H_t$. Hence $\{H_s\}_{s \in S}$ is a separating family. □

Lemma 22. *A separating family with minimal length sequences can be constructed in time $\mathcal{O}(m \log n + n^2)$.*

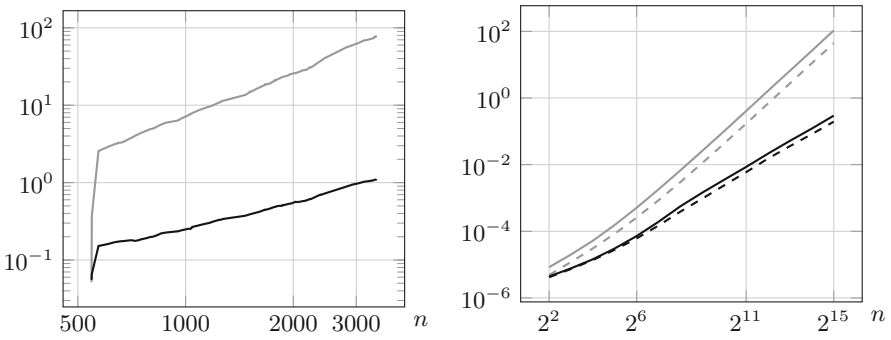
Proof. The separating family can be constructed from the splitting tree by collecting all sequences of all parents of a state (by Lemma 21). Since we have to do this for every state, this takes $\mathcal{O}(n^2)$ time. □

For test generation one moreover needs a transition cover. This can be constructed in linear time with a breadth first search. We conclude that we can construct all necessary information for the W-method in time $\mathcal{O}(m \log n)$ as opposed

the $\mathcal{O}(mn)$ algorithm used in [2]. Furthermore, we conclude that we can construct all the necessary information for the HSI-method in time $\mathcal{O}(m \log n + n^2)$, improving on the reported bound $\mathcal{O}(mn^3)$ in [5]. The original HSI-method was formulated differently and might generate smaller sets. We conjecture that our separating family has the same size if we furthermore remove redundant prefixes. This can be done in $\mathcal{O}(n^2)$ time using a trie data structure.

6 Experimental Results

We have implemented Algorithms 3 and 4 in Go, and we have compared their running time on two sets of FSMs.[1] The first set is from [10], where FSMs for embedded control software were automatically constructed. These FSMs are of increasing size, varying from 546 to 3 410 states, with 78 inputs and up to 151 outputs. The second set is inferred from [6], where two classes of finite automata, A and B, are described that serve as a worst case for Algorithms 3 and 4 respectively. The FSMs that we have constructed for these automata have 1 input, 2 outputs, and $2^2 - 2^{15}$ states. The running times in seconds on an Intel Core i5-2500 are plotted in Fig. 3. We note that different slopes imply different complexity classes, since both axes have a logarithmic scale.



(a) Embedded control software (b) Class A (dashed) and class B (solid)

Fig. 3. Running time in seconds of Algorithms 3 (gray) and 4 (black)

7 Conclusion

In this paper we have described an efficient algorithm for constructing a set of minimal-length sequences that pairwise distinguish all states of a finite state machine. By extending Hopcroft's minimization algorithm, we are able to construct such sequences in $\mathcal{O}(m \log n)$ for a machine with m transitions and n states. This improves on the traditional $\mathcal{O}(mn)$ method that is based on the

[1] Available at https://gitlab.science.ru.nl/rick/partition/.

classic algorithm by Moore. As an upshot, the sequences obtained form a characterization set and a separating family, which play a crucial in conformance testing.

Two key observations were required for a correct adaptation of Hopcroft's algorithm. First, it is required to perform splits in order of the length of their associated sequences. This guarantees minimality of the obtained separating sequences. Second, it is required to consider nodes as a candidate before any one of its children are considered as a candidate. This order follows naturally from the construction of a splitting tree.

Experimental results show that our algorithm outperforms the classic approach for both worst-case finite state machines and models of embedded control software. Applications of minimal separating sequences such as the ones occurring in [2,10] therefore show that our algorithm is useful in practice.

References

1. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: POPL, pp. 457–468 (2013)
2. Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A., Yevtushenko, N.: FSM-based conformance testing methods: a survey annotated with experimental evaluation. Inf. Softw. Technol. **52**(12), 1286–1297 (2010)
3. Gill, A.: Introduction to the Theory of Finite-state Machines. McGraw-Hill, New York (1962)
4. Gries, D.: Describing an algorithm by Hopcroft. Acta Informatica **2**(2), 97–109 (1973)
5. Hierons, R.M., Türker, U.C.: Incomplete distinguishing sequences for finite state machines. Comput. J. **58**, 1–25 (2015)
6. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. In: Theory of Machines and Computations, pp. 189–196 (1971)
7. Knuutila, T.: Re-describing an algorithm by Hopcroft. Theoret. Comput. Sci. **250**(1–2), 333–363 (2001)
8. Lee, D., Yannakakis, M.: Testing finite-state machines: state identification and verification. Computers **43**(3), 306–320 (1994)
9. Moore, E.F.: Gedanken-experiments on sequential machines. Automata Stud. **34**, 129–153 (1956)
10. Smeenk, W., Moerman, J., Vaandrager, F., Jansen, D.N.: Applying automata learning to embedded control software. In: Butler, M., et al. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 67–83. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25423-4_5
11. Valmari, A., Lehtinen, P.: Efficient minimization of DFAs with partial transition functions. In: STACS, pp. 645–656 (2008)