EFFICIENT MINIMIZATION OF DFAS WITH PARTIAL TRANSITION FUNCTIONS

ANTTI VALMARI [1] AND PETRI LEHTINEN [1]

[1] Tampere University of Technology, Institute of Software Systems, PO Box 553, FI-33101 Tampere, Finland
E-mail address: {Antti.Valmari,Petri.Lehtinen}@tut.fi

Abstract. Let *PT-DFA* mean a deterministic finite automaton whose transition relation is a partial function. We present an algorithm for minimizing a PT-DFA in $O(m \lg n)$ time and $O(m + n + \alpha)$ memory, where n is the number of states, m is the number of *defined* transitions, and α is the size of the alphabet. Time consumption does not depend on α, because the α term arises from an array that is accessed at random and never initialized. It is not needed, if transitions are in a suitable order in the input. The algorithm uses two instances of an array-based data structure for maintaining a refinable partition. Its operations are all amortized constant time. One instance represents the classical blocks and the other a partition of transitions. Our measurements demonstrate the speed advantage of our algorithm on PT-DFAs over an $O(\alpha n \lg n)$ time, $O(\alpha n)$ memory algorithm.

1. Introduction

Minimization of a deterministic finite automaton (DFA) is a classic problem in computer science. Let n be the number of states, m the number of transitions and α the size of the alphabet of the DFA. Hopcroft made a breakthrough in 1970 by presenting an algorithm that runs in $O(n \lg n)$ time, treating α as a constant [5]. Gries made the dependence of the running time of the algorithm on α explicit, obtaining $O(\alpha n \lg n)$ [3]. (Complexity is reported using the RAM machine model under the uniform cost criterion [1, p. 12].)

Our starting point was the paper by Knuutila in 2001, where he presented yet another $O(\alpha n \lg n)$ algorithm, and remarked that some versions which have been believed to run within this time bound actually fail to do so [6]. Hopcroft's algorithm is based on using only the "smaller" half of some set (known as *block*) that has been split. Knuutila demonstrated with an example that although the most well-known notion of "smaller" automatically leads to $O(\alpha n \lg n)$, two other notions that have been used may yield $\Omega(n^3)$ when $\alpha = \frac{1}{2}n$. He also showed that this can be avoided by maintaining, for each symbol, the set of those states in the block that have input transitions labelled by that symbol. According to [3], Hopcroft's original algorithm did so. Some later authors have dropped this complication as unnecessary, although it is necessary when the alternative notions of "smaller" are used.

Key words and phrases: deterministic finite automaton, sparse adjacency matrix, partition refinement.
Petri Lehtinen was funded by Academy of Finland, project ALEA (210795).

Knuutila mentioned as future work whether his approach can be used to develop an $O(m \lg n)$ algorithm for DFAs whose transition functions are not necessarily total. For brevity, we call them PT-DFAs. With an ordinary DFA, $O(m \lg n)$ is the same as $O(\alpha n \lg n)$ as $m = \alpha n$, but with a PT-DFA it may be much better. We present such an algorithm in this paper. We refined Knuutila's method of maintaining sets of states with relevant input transitions into a full-fledged data structure for maintaining refinable partitions. Instead of maintaining those sets of states, our algorithm maintains the corresponding sets of transitions. Another instance of the structure maintains the blocks.

Knuutila seems to claim that such a PT-DFA algorithm arises from the results in [7], where an $O(m \lg n)$ algorithm was described for refining a partition against a relation. However, there $\alpha = 1$, so the solved problem is not an immediate generalisation of ours. Extending the algorithm to $\alpha > 1$ is not trivial, as can be appreciated from the extension in [2]. It discusses $O(m \lg n)$ without openly promising it. Indeed, its analysis treats α as a constant. It seems to us that its running time does have an αn term.

In Section 2 we present an abstract minimization algorithm that, unlike [3, 6], has been adapted to PT-DFAs and avoids scanning the blocks and the alphabet in nested loops. The latter is crucial for converting αn into m in the complexity. The question of what blocks are needed in further splitting, has led to lengthy and sometimes unconvincing discussions in earlier literature. Our correctness proof deals with this issue using the "loop invariant" paradigm advocated in [4]. Our loop invariant "knows" what blocks are needed.

Section 3 presents an implementation of the refinable partition data structure. Its performance relies on a carefully chosen combination of simple low-level programming details.

The implementation of the main part of the abstract algorithm is the topic of Section 4. The analysis of its time consumption is based on proving of two lines of the code that, whenever the line is executed again for the same transition, the end state of the transition resides in a block whose size is at most half the size in the previous time. The numbers of times the remaining lines are executed are then related to these lines.

With a time bound as tight as ours, the order in which the transitions are presented in the input becomes significant, since the $\Theta(m \lg m)$ time that typical good sorting algorithms tend to take does not necessarily fit $O(m \lg n)$. We discuss this problem in Section 5, and present a solution that runs in $O(m)$ time but may use more memory, namely $O(m + \alpha)$.

Some measurements made with our implementations of Knuutila's and our algorithm are shown in Section 6.

2. Abstract Algorithm

A *PT-DFA* is a 5-tuple $\mathcal{D} = (Q, \Sigma, \delta, \hat{q}, F)$ such that Q and Σ are finite sets, $\hat{q} \in Q$, $F \subseteq Q$ and δ is explained below. The elements of Q are called *states*, $\hat{q}$ is the *initial state*, and F is the set of *final states*. The set Σ is the *alphabet*. We have $\delta \subseteq Q \times \Sigma \times Q$, and δ satisfies the condition that if $(q, a, q_1) \in \delta$ and $(q, a, q_2) \in \delta$, then $q_1 = q_2$. The elements of δ are *transitions*. In essence, δ is a partial function from $Q \times \Sigma$ to Q. Therefore, if $(q, a, q') \in \delta$, we write $\delta(q, a) = q'$. If $q \in Q$ and $a \in \Sigma$ but there is no q' such that $(q, a, q') \in \delta$, we write $\delta(q, a) = \bot$, where $\bot$ is some symbol satisfying $\bot \notin Q$. We will use $|\delta|$ as the number of transitions, and this number may be much smaller than $|Q||\Sigma|$, which is the number of transitions if δ is a full function.

By $q \xrightarrow{a_1 a_2 \cdots a_n} q'$ we denote that there is a path from state q to state q' such that the labels along the path constitute the word $a_1 a_2 \cdots a_n$. That is, $q \xrightarrow{\varepsilon} q$ holds for

```
1   $(Q, \Sigma, \delta, \hat{q}, F)$ := remove irrelevant states and transitions from $(Q, \Sigma, \delta, \hat{q}, F)$
2   **if** $F = \emptyset$ **then return** empty_DFA($\Sigma$)
3   **else**
4       **if** $Q = F$ **then** $\mathcal{B} := \{F\}$ **else** $\mathcal{B} := \{F,\ Q - F\}$
5       $\mathcal{U} := \{\, (B, a) \mid B \in \mathcal{B} \land a \in \Sigma \land \delta_{B,a} \neq \emptyset \,\}$
6       **while** $\mathcal{U} \neq \emptyset$ **do**
7           $(B, a) :=$ any_element_of($\mathcal{U}$); $\mathcal{U} := \mathcal{U} - \{(B, a)\}$
8           **for** $C \in \mathcal{B}$ such that $\exists q \in C : \delta(q, a) \in B$ **do**
9               $C_1 := \{\, q \in C \mid \delta(q, a) \in B \,\}$; $C_2 := C - C_1$
10              **if** $C_2 \neq \emptyset$ **then**
11                  $\mathcal{B} := \mathcal{B} - \{C\}$; $\mathcal{B} := \mathcal{B} \cup \{C_1, C_2\}$
12                  **if** $|C_1| \leq |C_2|$ **then** $small := 1$; $big := 2$ **else** $small := 2$; $big := 1$
13                  $\mathcal{U} := \mathcal{U} \cup \{\, (C_{small}, b) \mid \delta_{C_{small},b} \neq \emptyset \land b \in \Sigma \,\}$
14                  $\mathcal{U} := \mathcal{U} \cup \{\, (C_{big}, b) \mid \delta_{C_{big},b} \neq \emptyset \land (C, b) \in \mathcal{U} \,\}$
15                  $\mathcal{U} := \mathcal{U} - (\, \{C\} \times \Sigma \,)$
16      $Q' := \mathcal{B}$; $\delta' := \emptyset$; $\hat{q}' := Block(\hat{q})$; $F' := \emptyset$
17      **for** $B \in \mathcal{B}$ **do**
18          $q :=$ any_element_of($B$)
19          **if** $q \in F$ **then** $F' := F' \cup \{B\}$
20          **for** $a \in \Sigma$ such that $\delta(q, a) \neq \bot$ **do** $\delta' := \delta' \cup \{\, (B, a, Block(\delta(q, a))) \,\}$
21      **return** $(Q', \Sigma, \delta', \hat{q}', F')$
```

The set U contains those nonempty splitters that are currently "unprocessed"

Figure 1: Abstract PT-DFA minimization algorithm

every $q \in Q$, and $q \xrightarrow{a_1 a_2 \cdots a_n a_{n+1}} q'$ holds if and only if there is some $q'' \in Q$ such that $q \xrightarrow{a_1 a_2 \cdots a_n} q''$ and $\delta(q'', a_{n+1}) = q' \neq \bot$. The *language* accepted by $\mathcal{D}$ is the set of words labelling the paths from the initial state to final states, that is, $\mathcal{L}(\mathcal{D}) = \{\, \sigma \in \Sigma^* \mid \exists q \in F : \hat{q} \xrightarrow{\sigma} q \,\}$. We will also talk about the languages of individual states, that is, $\mathcal{L}(q) = \{\, \sigma \in \Sigma^* \mid \exists q' \in F : q \xrightarrow{\sigma} q' \,\}$. Obviously $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\hat{q})$.

相关状态
是初始状态
或
能由初始状态可达的状态

We say that a state is *relevant*, if and only if either it is the initial state, or it is reachable from the initial state and some final state is reachable from it. More precisely, $R = \{\hat{q}\} \cup \{\, q \in Q \mid \exists q' \in F : \exists \sigma \in \Sigma^* : \exists \rho \in \Sigma^* : \hat{q} \xrightarrow{\sigma} q \xrightarrow{\rho} q' \,\}$. It is obvious that irrelevant states and their adjacent transitions may be removed from a PT-DFA without affecting its language. The initial state cannot be removed, because otherwise the result would violate the condition $\hat{q} \in Q$ in the definition of a DFA. The removal yields the PT-DFA $(R, \Sigma, \delta', \hat{q}, F')$, where $\delta' = \delta \cap (R \times \Sigma \times R)$ and $F' = F \cap R$.

If no final state is reachable from the initial state, then $\mathcal{L}(\mathcal{D}) = \emptyset$. This is handled as a special case in our algorithm, because otherwise the result might contain unnecessary transitions from the initial state to itself. For this purpose, let empty_DFA(Σ) be $(\{x\}, \Sigma, \emptyset, x, \emptyset)$, where x is just any element. Obviously empty_DFA(Σ) is the smallest PT-DFA with the alphabet Σ that accepts the empty language.

The abstract minimization algorithm is shown in Figure 1. In it, $\mathcal{B}$ denotes a *partition* on Q. That is, $\mathcal{B}$ is a collection $\{B_1, B_2, \ldots, B_n\}$ of nonempty subsets of Q such that $B_1 \cup B_2 \cup \cdots \cup B_n = Q$, and $B_i \cap B_j = \emptyset$ whenever $1 \leq i < j \leq n$. The elements of $\mathcal{B}$ are called *blocks*. By checking all statements that modify the contents of $\mathcal{B}$, it is easy to verify that after its initialization on line 4, $\mathcal{B}$ is a partition on Q throughout the execution of the algorithm, except temporarily in the middle of line 11.

By $\mathit{Block}(q)$ we denote the block to which state q belongs. Therefore, if $q \in Q$, then $q \in \mathit{Block}(q) \in \mathcal{B}$. For convenience, we define $\mathit{Block}(\bot) = \bot \notin \mathcal{B}$. If $\mathit{Block}(q_1) \neq \mathit{Block}(q_2)$ ever starts to hold, then it stays valid up to the end of the execution of the algorithm.

Elements of $\mathcal{B} \times \Sigma$ are called *splitters*. Let $\delta_{B,a} = \{ (q, a, q') \in \delta \mid q' \in B \}$. We say that splitter (B, a) is *nonempty*, if and only if $\delta_{B,a} \neq \emptyset$. The set $\mathcal{U}$ contains those nonempty splitters that are currently "unprocessed". It is obvious from line 8 that empty splitters would have no effect. The main loop of the algorithm (lines 6...15) starts with all nonempty splitters as unprocessed, and ends when no nonempty splitter is unprocessed. The classic algorithm uses either only F or only $Q - F$ for constructing the initial splitters, but this does not work with a partial δ.

The goal of the main loop is to split blocks until they are consistent with δ, without splitting too much. We will now prove in two steps that this is achieved.

Lemma 2.1. *For every $q_1 \in Q$ and $q_2 \in Q$, if $\mathit{Block}(q_1) \neq \mathit{Block}(q_2)$ at any time of the execution of the algorithm in Figure 1, then $\mathcal{L}(q_1) \neq \mathcal{L}(q_2)$.*

Proof. If the algorithm puts states q_1 and q_2 into different blocks on line 4, then either $\varepsilon \in \mathcal{L}(q_1) \wedge \varepsilon \notin \mathcal{L}(q_2)$ or $\varepsilon \notin \mathcal{L}(q_1) \wedge \varepsilon \in \mathcal{L}(q_2)$. Otherwise, it does so on line 11. Then there are i, j, B and a such that $\{i, j\} = \{1, 2\}$, $\delta(q_i, a) \in B$ and $\delta(q_j, a) \notin B$. Let $q'_i = \delta(q_i, a)$.

If $\delta(q_j, a) \neq \bot$, then let $q'_j = \delta(q_j, a)$. We have $q'_j \notin B$. Because the algorithm has already put q'_i and q'_j into different blocks (they were in different blocks on line 9), there is some $\sigma \in \Sigma^*$ such that either $\sigma \in \mathcal{L}(q'_i) \wedge \sigma \notin \mathcal{L}(q'_j)$ or vice versa. As a consequence, $a\sigma$ is in $\mathcal{L}(q_1)$ or in $\mathcal{L}(q_2)$, but not in both.

Assume now that $\delta(q_j, a) = \bot$. Because of lines 1 and 2, $\mathcal{L}(q) \neq \emptyset$ for every $q \in Q$. There is thus some $\sigma \in \Sigma^*$ such that $\sigma \in \mathcal{L}(q'_i)$. We have $a\sigma \in \mathcal{L}(q_i)$. Clearly $a\sigma \notin \mathcal{L}(q_j)$. ∎

At this point it is worth noticing that line 1 is important for the correctness of the algorithm. Without it, there could be two reachable states q_1 and q_2 that accept the same language, and a such that $\delta(q_1, a) = \bot$ while $\delta(q_2, a)$ is a state that accepts the empty language. The algorithm would eventually put q_1 and q_2 into different blocks.

We have shown that the main loop does not split blocks when it should not. We now prove that it splits all the blocks that it should.

Lemma 2.2. *At the end of the algorithm in Figure 1, for every $q_1 \in Q$, $q_2 \in Q$ and $a \in \Sigma$, if $\mathit{Block}(q_1) = \mathit{Block}(q_2)$, then $\mathit{Block}(\delta(q_1, a)) = \mathit{Block}(\delta(q_2, a))$.*

Proof. To improve readability, let $B_1 = \mathit{Block}(\delta(q_1, a))$ and $B_2 = \mathit{Block}(\delta(q_2, a))$. In the proof, $\mathit{Block}()$, B_1 and B_2 are always evaluated with the current $\mathcal{B}$, so their contents change. The proof is based on the following loop invariant:

> On line 6, for every $q_1 \in Q$, $q_2 \in Q$ and $a \in \Sigma$, if $\mathit{Block}(q_1) = \mathit{Block}(q_2)$, then $B_1 = B_2$ or $(B_1, a) \in \mathcal{U}$ or $(B_2, a) \in \mathcal{U}$.

Consider the situation immediately after line 5. If $B_1 \neq \bot$, then $(B_1, a) \in \mathcal{U}$. If $B_2 \neq \bot$, then $(B_2, a) \in \mathcal{U}$. If $B_1 = B_2 = \bot$, then $B_1 = B_2$. Thus the invariant holds initially.

Consider any q_1, q_2, a and instance of executing line 6 such that the invariant holds. Our task is to show that the invariant holds for them also when line 6 is executed for the next time.

The case that the invariant holds because $\mathit{Block}(q_1) \neq \mathit{Block}(q_2)$ is simple. Blocks are never merged, so $\mathit{Block}(q_1) \neq \mathit{Block}(q_2)$ is valid also the next time.

Consider the case $\mathit{Block}(q_1) = \mathit{Block}(q_2)$, $B_1 \neq B_2$ and $(B_i, a) \in \mathcal{U}$, where $i = 1$ or $i = 2$. Let $j = 3 - i$. If (B_i, a) is the (B, a) of line 7, then, when $\mathit{Block}(q_i)$ is the C of the

for-loop, q_i goes to C_1 and q_j goes to C_2. So $\mathit{Block}(q_1) = \mathit{Block}(q_2)$ ceases to hold, rescuing the invariant. If (B_i, a) is not the (B, a) of line 7, then, whenever B_i is split, lines 13 and 14 take care that both halves end up in $\mathcal{U}$. Thus $(B_i, a) \in \mathcal{U}$ stays true keeping the invariant valid, although B_i and $\mathcal{U}$ may change.

Let now $\mathit{Block}(q_1) = \mathit{Block}(q_2)$ and $B_1 = B_2$. To invalidate the invariant, B_1 or B_2 must be changed so that $B_1 = B_2$ ceases to hold. When this happens, line 13 puts (B_i, a) into $\mathcal{U}$, where $i = 1$ or $i = 2$. Like above, lines 13 and 14 keep (B_i, a) in $\mathcal{U}$ although B_i may change until line 6 is entered again.

We have completed the proof that the invariant stays valid.

When line 16 is entered, $\mathcal{U} = \emptyset$. The invariant now yields that if $\mathit{Block}(q_1) = \mathit{Block}(q_2)$, then $\mathit{Block}(\delta(q_1, a)) = \mathit{Block}(\delta(q_2, a))$. ■

It is not difficult to check that lines 16. . . 20 yield a PT-DFA, that is, Q' and Σ are finite sets and so on. In particular, the construction gives $\delta'(B, a)$ a value at most once. We now show that the result is the right PT-DFA.

Lemma 2.3. *Let $\mathcal{D}' = (Q', \Sigma, \delta', \hat{q}', F')$ be the result of the algorithm in Figure 1. We have $\mathcal{L}(\mathcal{D}') = \mathcal{L}(\mathcal{D})$. Furthermore, every PT-DFA that accepts $\mathcal{L}(\mathcal{D})$ has at least as many states and transitions as $\mathcal{D}'$. If it has the same number of states, it is either isomorphic with $\mathcal{D}'$ (ignoring Σ in the comparison), or it is of the form $(\{\hat{q}''\}, \Sigma'', \delta'', \hat{q}'', \emptyset)$ with $\delta'' \neq \emptyset$.*

Proof. The case where the algorithm exits on line 2 is trivial and has been discussed, so from now on we discuss the case where the algorithm goes through the main part.

Let $q \in Q$ and $a \in \Sigma$. Lemma 2.2 implies that $\mathit{Block}(\delta(q, a)) = \mathit{Block}(\delta(q', a))$ for every $q' \in \mathit{Block}(q)$. From this line 20 yields $\delta'(\mathit{Block}(q), a) = \mathit{Block}(\delta(q, a))$. By induction, if $\sigma \in \Sigma^*$, $q' \in Q$ and $q \mathrel{-\sigma\!\!\rightarrow} q'$ in $\mathcal{D}$ then $\mathit{Block}(q) \mathrel{-\sigma\!\!\rightarrow} \mathit{Block}(q')$ in $\mathcal{D}'$, and if $\mathit{Block}(q) \mathrel{-\sigma\!\!\rightarrow} B \neq \perp$ in $\mathcal{D}'$ then there is $q' \in Q$ such that $B = \mathit{Block}(q')$ and $q \mathrel{-\sigma\!\!\rightarrow} q'$ in $\mathcal{D}$. Similarly, lines 4 and 19 guarantee that $q' \in F$ if and only if $\mathit{Block}(q') \in F'$. Together these yield $\mathcal{L}(q) = \mathcal{L}(\mathit{Block}(q))$ and, in particular, $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\hat{q}) = \mathcal{L}(\hat{q}') = \mathcal{L}(\mathcal{D}')$.

Let $(Q'', \Sigma, \delta'', \hat{q}'', F'')$ be any PT-DFA that accepts the same language as $\mathcal{D}'$. Let $q' \in Q'$. Because the algorithm executed the main part, there are some $\sigma \in \Sigma^*$ and $\rho \in \Sigma^*$ such that $\hat{q}' \mathrel{-\sigma\!\!\rightarrow} q'$ and $\rho \in \mathcal{L}(q')$. So $\sigma\rho \in \mathcal{L}(\hat{q}') = \mathcal{L}(\hat{q}'')$, and also Q'' contains a state q'' such that $\hat{q}'' \mathrel{-\sigma\!\!\rightarrow} q''$ and $\mathcal{L}(q'') = \mathcal{L}(q')$. As σ may vary, there may be many q'' with $\mathcal{L}(q'') = \mathcal{L}(q')$. We arbitrarily choose one of them and denote it with $f(q')$. Lemma 2.1 implies that if $q'_1 \neq q'_2$, then $\mathcal{L}(q'_1) \neq \mathcal{L}(q'_2)$, yielding $f(q'_1) \neq f(q'_2)$. So $|Q''| \geq |Q'|$. If $\delta'(q', a) \neq \perp$, then some $a\rho' \in \mathcal{L}(q') = \mathcal{L}(f(q'))$, so $\delta''(f(q'), a) \neq \perp$. As a consequence, $|\delta''| \geq |\delta'|$.

If $|Q''| = |Q'|$, then f is an isomorphism. ■

The proof has the consequence that after the end of the main loop, $\mathit{Block}(q_1) = \mathit{Block}(q_2)$ if and only if $\mathcal{L}(q_1) = \mathcal{L}(q_2)$.

Let us consider the number of times a transition (q, a, q') can be used on line 9. It is used whenever such a (B, a) is taken from $\mathcal{U}$ that $q' \in B$, that is, $\mathit{Block}(q') = B$. So, shortly before using (q, a, q'), $(\mathit{Block}(q'), a) \in \mathcal{U}$ held but ceased to hold (line 7). To use it again, $(\mathit{Block}(q'), a) \in \mathcal{U}$ must be made to hold again. To make $(\mathit{Block}(q'), a) \in \mathcal{U}$ to hold again, line 13 or 14 must be executed such that $\mathit{Block}(q')$ is in the role of C_{small} or C_{big}, and a is in the role of b. But line 14 tests that $(C, b) \in \mathcal{U}$, so it cannot make $(\mathit{Block}(q'), a) \in \mathcal{U}$ to hold if it did not hold already on line 9, although it can keep $(\mathit{Block}(q'), a) \in \mathcal{U}$ valid. So only line 13 can make $(\mathit{Block}(q'), a) \in \mathcal{U}$ to hold again. An important detail of the algorithm is that line 13 puts the *smaller* half of C (paired with a) into $\mathcal{U}$. Therefore, each time $(\mathit{Block}(q'), a) \in \mathcal{U}$ starts to hold again, q' resides in a block whose size is at most half of the size in the previous time. As a consequence, (q, a, q') can be used for splitting at most $\lg |Q| + 1$ times.

3. Refinable Partitions

The refinable partition data structure maintains a partition of the set $\{1, \ldots, \mathit{max}\}$. Our algorithm uses one instance of it with $\mathit{max} = |Q|$ for the blocks and another with $\mathit{max} = |\delta|$ for the splitters. Each set in the partition has an index in the range $1, \ldots, \mathit{sets}$, where *sets* is the current number of sets. The structure supports the following operations.

$\mathit{Size}(s)$**:** Returns the number of elements in the set with index s.

$\mathit{Set}(e)$**:** Returns the index of the set that element e belongs to.

$\mathit{First}(s)$ **and** $\mathit{Next}(e)$**:** The elements of the set s can be scanned by executing first $e := \mathit{First}(s)$ and then **while** $e \neq 0$ **do** $e := \mathit{Next}(e)$. Each element will be returned exactly once, but the ordering in which they are returned is unspecified. While scanning a set, *Mark* and *Split* must not be executed.

$\mathit{Mark}(e)$**:** Marks the element e for splitting of a set.

$\mathit{Split}(s)$**:** If either none or all elements of set s have been marked, returns 0. Otherwise removes the marked elements from the set, makes a new set of the marked elements, and returns its index. In both cases, unmarks all the elements in the set or sets.

$\mathit{No\_marks}(s)$**:** Returns True if and only if none of the elements of s is marked.

The implementation uses the following *max*-element arrays.

*elems***:** Contains $1, \ldots, \mathit{max}$ in such an order that elements that belong to the same set are next to each other.

*loc***:** Tells the location of each element in *elems*, that is, $\mathit{elems}[\mathit{loc}[e]] = e$.

*sidx***:** The index of the set that e belongs to is $\mathit{sidx}[e]$.

first **and** *end***:** The elements of set s are $\mathit{elems}[f]$, $\mathit{elems}[f+1]$, $\ldots$, $\mathit{elems}[\ell]$, where $f = \mathit{first}[s]$ and $\ell = \mathit{end}[s] - 1$.

*mid***:** Let f and ℓ be as above, and let $m = \mathit{mid}[s]$. The marked elements are $\mathit{elems}[f]$, $\ldots$, $\mathit{elems}[m-1]$, and the unmarked are $\mathit{elems}[m]$, $\ldots$, $\mathit{elems}[\ell]$.

Initially $\mathit{sets} = 1$, $\mathit{first}[1] = \mathit{mid}[1] = 1$, $\mathit{end}[1] = \mathit{max} + 1$, and $\mathit{elems}[e] = \mathit{loc}[e] = e$ and $\mathit{sidx}[e] = 1$ for $e \in \{1, \ldots, \mathit{max}\}$. Initialization takes $O(\mathit{max})$ time and $O(1)$ additional memory.

The implementation of the operations is shown in Figure 2. Each operation runs in constant time, except *Split*, whose worst-case time consumption is linear in the number M of marked elements. However, also *Split* can be treated as constant-time in the analysis of our algorithm, because it is amortized constant time. When calling *Split*, there had been M calls of *Mark*. They are unique to this call of *Split*, because *Split* unmarks the elements in question. The total time consumption of these calls of *Mark* and *Split* is $\Theta(M)$, but the same result is obtained even if *Split* is treated as constant-time.

4. Block-splitting Stage

In this section we show how lines 4…15 of the abstract algorithm can be implemented in $O(|\delta| \lg |Q|)$ time and $O(|\delta|)$ memory assuming that δ is available in a suitable ordering. The implementation of abstract lines 1…3 and 16…21 in $O(|Q| + |\delta|)$ time and memory

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

Set(e)
return $sidx[e]$

First(s)
return $elems[first[s]]$

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

Mark(e)
$s := sidx[e]$; $\ell := loc[e]$; $m := mid[s]$
if $\ell \geq m$ then
    $elems[\ell] := elems[m]$; $loc[elems[\ell]] := \ell$
    $elems[m] := e$; $loc[e] := m$; $mid[s] := m + 1$

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

No_marks(s)
if $mid[s] = first[s]$ then return True
else return False
```

Figure 2: Implementation of the refinable partition data structure

is easy and not discussed further in this paper. (By "abstract lines" we refer to lines in Figure 1).

The implementation relies on the following data structures. The "simple sets" among them are all initially empty. They have only three operations, all $O(1)$ time: the set is empty if and only if *Empty* returns True, *Add*(i) adds number i to the set without checking if it already is there, and *Remove* removes any number from the set and returns the removed number. The implementation may choose freely the element that *Remove* removes and returns. One possible efficient implementation of a simple set consists of an array that is used as a stack.

tail, *label* **and** *head*: The transitions have the indices 1, ..., $|\delta|$. If t is the index of the transition (q, a, q'), then $tail[t] = q$, $label[t] = a$, and $head[t] = q'$.

In_trs: This stores the indices of the input transitions of state q. The ordering of the transitions does not matter. This is easy to implement efficiently. For instance, one may use an array *elems* of size $|\delta|$, together with arrays *first* and *end* of size $|Q|$, so that the indices of the input transitions of q are $elems[first[q]]$, $elems[first[q] + 1]$,

$\ldots$, $elems[end[q]-1]$. The array can be initialized in $O(|Q|+|\delta|)$ time with counting sort, using $head[t]$ as the key.

BRP: This is a refinable partition data structure on $\{1,\ldots,|Q|\}$. It represents $\mathcal{B}$, that is, the blocks. The index of the set in *BRP* is used as the index of the block also elsewhere in the algorithm. Initially *BRP* consists of one set that contains the indices of the states.

TRP: This is a refinable partition data structure on $\{1,\ldots,|\delta|\}$. Each of the sets in it consists of the indices of the input transitions of some nonempty splitter (B,a). That is, *TRP* stores $\{\,\delta_{B,a} \mid B \in \mathcal{B} \wedge a \in \Sigma \wedge \delta_{B,a} \neq \emptyset\,\}$. The index of $\delta_{B,a}$ in *TRP* is used as the index of (B,a) also elsewhere in the algorithm. For this reason, we will occasionally use the word "splitter" also of the sets in *TRP*. Initially *TRP* consists of $\{\,\delta_{Q,a} \mid a \in \Sigma \wedge \delta_{Q,a} \neq \emptyset\,\}$, that is, two transitions are in the same set if and only if they have the same label. This can be established as follows:

```
for $a \in \Sigma$ such that $\delta_{Q,a} \neq \emptyset$ do
    for $t \in \delta_{Q,a}$ do $TRP.Mark(t)$
    $TRP.Split(1)$
```

If transitions are pre-sorted such that transitions with the same label are next to each other, then this runs in $O(|\delta|)$ time and $O(1)$ additional memory.

Unready_Spls: This is a simple set of numbers in the range 1, ..., $|\delta|$. It stores the indices of the unprocessed nonempty splitters. That is, it implements the $\mathcal{U}$ of the abstract algorithm. Because each nonempty splitter has at least one incoming transition and splitters do not share transitions, $|\delta|$ suffices for the range.

Touched_Blocks: This is a simple set of numbers in the range 1, ..., $|Q|$. It contains the indices of the blocks C that were met when backwards-traversing the incoming transitions of the current splitter on abstract line 8. It is always empty on line 19.

Touched_Spls: This is a simple set of numbers in the range 1, ..., $|\delta|$. It contains the indices of the splitters that were affected when scanning the incoming transitions of the smaller of the new blocks that resulted from a split. It is empty on line 4.

The block-splitting stage is shown in Figure 3. We explain its operation in the proof of the following theorem.

Theorem 4.1. *Given a PT-DFA all whose states are relevant and that has at least one final state, the algorithm in Figure 3 computes the same $\mathcal{B}$ (represented by BRP) as lines 4...15 of Figure 1.*

Proof. Let us first investigate the operation of *Split_block*. As was told earlier, *BRP* models $\mathcal{B}$, *TRP* models the set of all nonempty splitters (or the sets of their input transitions), and *Unready_Spls* models $\mathcal{U}$. The task of *Split_block* is to update these three variables according to the splitting of a block C. Before calling *Split_block*, the states q that should go to one of the halves have been marked by calling $BRP.Mark(q)$ for each of them.

Line 1 unmarks all states of C and either splits C in *BRP* updating $\mathcal{B}$, or detects that one of the halves would be empty, so C should not be split. In the latter case, line 2 exits the procedure. The total effect of the call and its preceding calls of *BRP.Mark* is zero (except that the ordering of the states in *BRP* may have changed).

From now on assume that both halves of C are nonempty. Line 3 makes b the index of the bigger half B and b' the index of the smaller half B'. Because C is no more a block, for each $a \in \Sigma$, the pairs (C,a) are no more splitters, and must be replaced by (B,a) and (B',a), to the extent that they are nonempty. For this purpose, lines 4, 5 and 10 scan

```
    Split_block(b)
1   b' := BRP.Split(b)
2   if b' ≠ 0 then
3       if BRP.Size(b) < BRP.Size(b') then b' := b
4       q := BRP.First(b')
5       while q ≠ 0 do
6           for t ∈ In_trs[q] do
7               p := TRP.Set(t)
8               if TRP.No_marks(p) then Touched_Spls.Add(p)
9               TRP.Mark(t)
10          q := BRP.Next(q)
11      while ¬Touched_Spls.Empty do
12          p := Touched_Spls.Remove
13          p' := TRP.Split(p)
14          if p' ≠ 0 then Unready_Spls.Add(p')

    Main_part
15  Initialize TRP to { $\delta_{Q,a}$ | $a \in \Sigma \wedge \delta_{Q,a} \neq \emptyset$ }
16  for p := 1 to TRP.sets do Unready_Spls.Add(p)
17  for q ∈ F do BRP.Mark(q)
18  Split_block(1)
19  while ¬Unready_Spls.Empty do
20      p := Unready_Spls.Remove
21      t := TRP.First(p)
22      while t ≠ 0 do
23          q := tail[t]; b' := BRP.Set(q)
24          if BRP.No_marks(b') then Touched_Blocks.Add(b')
25          BRP.Mark(q)
26          t := TRP.Next(t)
27      while ¬Touched_Blocks.Empty do
28          b := Touched_Blocks.Remove
29          Split_block(b)
```

Figure 3: Implementation of lines 4...15 of the abstract algorithm

B' and line 6 scans the incoming transitions of the currently scanned state of B'. Line 9 marks, for each $a \in \Sigma$, the transitions that correspond to (B', a). Line 7 finds the index of (C, a) in *TRP*, and line 8 adds it to *Touched_Spls*, unless it is there already. After all input transitions of B' have been scanned, lines 11 and 12 discharge the set of affected splitters (C, a). Line 13 updates (C, a) to those of (B, a) and (B', a) that are nonempty.

Line 14 corresponds to the updating of $\mathcal{U}$. If both (B, a) and (B', a) are nonempty splitters, then the index of (B', a) is added to *Unready_Spls*, that is, (B', a) is added to $\mathcal{U}$. In this case, (B, a) inherits the index of (C, a) and thus also the presence or absence in $\mathcal{U}$. If (B, a) is empty, then (B', a) inherits the index and $\mathcal{U}$-status of (C, a). If (B', a) is empty, then (C, a) does not enter *Touched_Spls* in the first place. To summarize, if $(C, a) \in \mathcal{U}$, then all of its nonempty heirs enter $\mathcal{U}$; otherwise only the smaller heir enters $\mathcal{U}$, and only if it is nonempty. This is equivalent to abstract lines 13...14. Regarding abstract line 15, (C, a) disappears automatically from $\mathcal{U}$ because its index is re-used.

Lines 15…18 implement the total effect of abstract lines 4…5. The initial value of *BRP* corresponds to $\mathcal{B} = \{Q\}$. Line 15 makes *TRP* contain the sets of input transitions of all nonempty splitters (Q, a) (where $a \in \Sigma$), and line 16 puts them all to $\mathcal{U}$. If $Q = F$, then lines 17 and 18 have no effect. Otherwise, they update $\mathcal{B}$ to $\{F, Q - F\}$, update *TRP* accordingly, and update *Unready_Spls* to contain all current nonempty splitters.

Lines 19 and 20 match trivially abstract lines 6 and 7. They choose some nonempty splitter (B, a) for processing. Lines 21…26 can be thought of as being executed between abstract lines 7 and 8. They mark the states in C_1 for every C that is scanned by abstract line 8, and collect the indices of those C into *Touched_Blocks*. Lines 27 and 28 correspond to abstract line 8, and abstract lines 9…15 are implemented by the call *Split_block*(b). Lines 1 and 2 have the same effect as abstract lines 9…11. Line 3 implements abstract line 12. The description of line 14 presented above matches abstract lines 13…15. ■

Theorem 4.2. *Given a PT-DFA all whose states are relevant and that has at least one final state, and assuming that the transitions that have the same label are given successively in the input, the algorithm in Figure 3 runs in $O(|\delta| \lg |Q|)$ time and $O(|\delta|)$ memory.*

Proof. The data structures have been listed in this section and they all consume $O(|Q|)$ or $O(|\delta|)$ memory. Their initialization takes $O(|Q| + |\delta|)$ time. Because all states are relevant, we have $|Q| \leq |\delta| + 1$, so $O(|Q|)$ terms are also $O(|\delta|)$.

We have already seen that each individual operation in the algorithm runs in amortized constant time, except for line 15, which takes $O(|\delta|)$ time. We also saw towards the end of Section 2 that each transition is used at most $\lg |Q| + 1$ times on line 9 of the abstract algorithm. This implies that line 25, and thus lines 23…26, are executed at most $|\delta|(\lg |Q| + 1)$ times. The same holds for lines 28 and 29, because the number of *Add*-operations on *Touched_Blocks* is obviously the same as *Remove*-operations. Because *TRP*-sets are never empty, lines 20 and 21 are not executed more often than line 25, and lines 22 and 27 are executed at most twice as many times as line 25. Line 19 is executed once more than line 20, and lines 15…18 are executed once. Line 16 runs in $O(|\delta|)$ and line 17 in $O(|Q|)$ time.

Lines 1…4 are executed at most once more than line 29. If $BRP.Size(b) \geq BRP.Size(b')$ on line 3, then each of the states scanned by lines 5 and 10 was marked on line 17 or 25. Otherwise the number of scanned states is smaller than the number of marked states. Therefore, line 10 is executed at most as many times as lines 17 and 25, and line 5 at most twice as many times. Whenever lines 7…9 are executed anew (or for the first time) for some transition, the end state of the transition belongs to a block whose size is at most half of the size in the previous time (or originally), because the block was split on line 1 and the smaller half was chosen on line 3. Therefore, lines 7…9 are executed at most $|\delta| \lg |Q|$ times. Line 6 is executed as many times as lines 7 and 10 together. The executions of lines 12…14 are determined by line 8, and of line 11 by lines 4 and 8. ■

5. Sorting Transitions

In *TRP*, transitions are sorted such that those with the same label are next to each other. Transitions are not necessarily in such an order in the input. Therefore, we must take the resources needed for sorting into account in our analysis.

Transitions can of course be sorted according to their labels with heapsort in $O(|\delta| \lg |\delta|)$ time and $O(|\delta|)$ memory. This is inferior to the time consumption of the rest of the algorithm. Because the labels need not be in alphabetical order, a suitable ordering can also

```
$TRP.sets := 0$
for $t \in \delta$ do
    $a := label[t]$; $i := idx[a]$
    if $i < 1 \vee i > TRP.sets \vee TRP.mid[i] \neq a$ then
        $i := TRP.sets + 1$; $TRP.sets := i$
        $idx[a] := i$; $TRP.mid[i] := a$; $TRP.end[i] := 1$
    else $TRP.end[i] := TRP.end[i] + 1$
$TRP.first[1] := 1$; $TRP.end[1] := TRP.end[1] + 1$; $TRP.mid[1] := TRP.end[1]$
for $i := 2$ to $TRP.sets$ do
    $TRP.first[i] := TRP.end[i-1]$
    $TRP.end[i] := TRP.first[i] + TRP.end[i]$; $TRP.mid[i] := TRP.end[i]$
for $t \in \delta$ do
    $i := idx[label[t]]$; $\ell := TRP.mid[i] - 1$; $TRP.mid[i] := \ell$
    $TRP.elems[\ell] := t$; $TRP.loc[t] := \ell$; $TRP.sidx[t] := i$
```

Figure 4: Initialization of *TRP* in $O(|\delta|)$ time and $O(|\Sigma|)$ additional memory

be found by putting the transitions into a hash table using their labels as the keys. Then nonempty hash lists are sorted and concatenated. This takes $O(|\delta|)$ time on the average, and $O(|\delta|)$ memory. However, the worst-case time consumption is still $O(|\delta| \lg |\delta|)$.

A third possibility runs in $O(|\delta|)$ time even in the worst case, but it uses $O(|\Sigma|)$ additional memory. That its time consumption may be smaller than memory consumption arises from the fact that it uses an array *idx* of size $|\Sigma|$ that need not be initialized at all, not even to all zeros. It is based on counting the occurrences of each label as in exercise 2.12 of [1], and then continuing like counting sort. The pseudocode is in Figure 4.

6. Measurements and Conclusions

Table 1 shows some measurements made with our test implementations of Knuutila's and our algorithm. They were written in C++ and executed on a PC with Linux and 1 gigabyte of memory. No attempt was made to optimise either implementation to the extreme. The implementation of Knuutila's algorithm completes the transition function to a full function with a well-known construction. Namely, it adds a "sink" state to which all originally absent transitions and all transitions starting from itself are directed.

The input DFAs were generated at random. Because of the difficulty of generating a precise number of transitions according to the uniform distribution, sometimes the generated number of transitions was slightly smaller than the desired number. Furthermore, the DFAs may have unreachable states and/or reachable irrelevant states that are processed separately by one or both of the algorithms. Running time depends also on the size of the minimized DFA: the smaller the result, the less splitting of blocks. We know that the joint effects of these phenomena were small, because, in all cases, the numbers of states and transitions of the minimized DFAs were $> 99.4\,\%$ of $|Q|$ and $|\delta|$ in the table. Therefore, instead of trying to avoid the imperfections by fine-tuning the input (which would be difficult), we always used the first input DFA that our generator gave for the given parameters.

The times given are the fastest and slowest of three measurements, made with $|F| = \frac{|Q|}{2} + d$, where $d \in \{-1, 0, 1\}$. They are given in seconds. The number of transitions $|\delta|$ varies between $10\,\%$ and $100\,\%$ of $|Q||\Sigma|$. The times contain the special processing of unreachable and irrelevant states, but they do not contain the reading of the input DFA from and writing

Table 1: Running time measurements. $|\delta| = p|Q||\Sigma|$, where p is given as %.
A: $|Q| =$ 1 000 and $|\Sigma| = 100$. B: $|Q| =$ 1 000 and $|\Sigma| = 1\,000$.
C: $|Q| = 10\,000$ and $|\Sigma| = 100$. D: $|Q| = 10\,000$ and $|\Sigma| = 1\,000$.

	alg.	10 %		30 %		50 %		70 %		90 %		100 %	
A	our	0.004	0.005	0.013	0.014	0.024	0.025	0.036	0.037	0.052	0.060	0.061	0.062
	Knu	0.026	0.026	0.034	0.035	0.040	0.041	0.045	0.046	0.048	0.049	0.053	0.054
B	our	0.059	0.061	0.277	0.279	0.549	0.551	0.855	0.865	1.181	1.211	1.330	1.416
	Knu	0.467	0.486	0.645	0.651	0.785	0.795	0.893	0.907	0.971	0.979	1.033	1.040
C	our	0.070	0.071	0.296	0.301	0.574	0.581	0.887	0.893	1.210	1.229	1.424	1.434
	Knu	0.526	0.529	0.730	0.734	0.901	0.904	1.027	1.035	1.128	1.130	1.200	1.202
D	our	1.224	1.238	4.038	4.087	7.132	7.164	10.50	10.57	14.18	14.34	16.41	16.48
	Knu	6.324	6.356	8.606	8.705	10.46	10.64	11.91	11.95	13.00	13.04	13.83	13.89

the result to a file. With $|Q| = |\Sigma| = 10\,000$, Knuutila's algorithm ran out of memory, while our algorithm spent about 15 s when $p = \frac{|\delta|}{|Q||\Sigma|} = 10\,\%$ and 32 s when $p = 20\,\%$.

The superiority of our algorithm when p is small is clear. That our algorithm loses when p is big may be because it uses both F and $Q - F$ in the initial splitters, whereas Knuutila's algorithm uses only one of them. Also Knuutila's algorithm speeds up as p becomes smaller. Perhaps the reason is that when p is, say, 10 %, the block that contains the sink state has an unproportioned number of input transitions, causing blocks to split to a small and big half roughly in the ratio of 10 % to 90 %. Thus small blocks are introduced quickly. As a consequence, the average size of the splitters that the algorithm uses during the execution is smaller than when $p = 100\,\%$. The same phenomenon also affects indirectly our algorithm, probably explaining why its running time is not linear in p.

Of the three notions of "smaller" mentioned in the introduction, our analysis does not apply to the other two. It seems that they would require making *Split_block* somewhat more complicated. This is a possible but probably unimportant topic for further work.

A near-future goal of us is to publish a much more complicated, true $O(m \lg n)$ algorithm for the problem in [2], that is, the multi-relational coarsest partition problem.

References

[1] Aho, A. V., Hopcroft, J. E., Ullman, J. D.: *The Design and Analysis of Computer Algorithms.* Addison-Wesley 1974.
[2] Fernandez, J.-C.: An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming* 13 (1989/90) 219–236.
[3] Gries, D.: Describing an Algorithm by Hopcroft. *Acta Informatica* 2 (1973) 97–109.
[4] Gries, D.: *The Science of Programming.* Springer 1981.
[5] Hopcroft, J.: An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. *Technical Report* CS-190, Stanford University, 1970.
[6] Knuutila, T.: Re-describing an Algorithm by Hopcroft. *Theoret. Computer Science* 250 (2001) 333–363.
[7] Paige, R., Tarjan, R.: Three Partition Refinement Algorithms. *SIAM J. Computing*, 16 (1987) 973–989.