ELSEVIER

Theoretical Computer Science 250 (2001) 333–363

Theoretical Computer Science

www.elsevier.com/locate/tcs

Re-describing an algorithm by Hopcroft

Timo Knuutila

Department of Computer Science, University of Turku, Lemminkäisenkatu 14 A, SF-20520 Turku, Finland

Received September 1997; revised March 1998
Communicated by M. Crochemore

Abstract

J. Hopcroft introduced already in 1970 an $O(n \log n)$-time algorithm for minimizing a finite deterministic automaton of n states. Although the existence of the algorithm is widely known, its theoretical justification, correctness and running time analysis are not. We give here a tutorial reconstruction of Hopcroft's algorithm focusing on a firm theoretical basis, clear correctness proofs and a well-founded computational analysis. Our analysis reveals that if the size of the input alphabet m is not fixed, then Hopcroft's original algorithm does not run in time $O(mn \log n)$ as is commonly believed in the literature. The $O(mn \log n)$ holds, however, for the variation presented later by D. Gries and for a new variant given in this article. We also propose a new efficient routine for refining the equivalence classes constructed in the algorithm and suggest a computationally sound heuristics as an enhancement. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Finite automata; Algorithms; Minimization

1. Introduction

Minimization of a *deterministic finite automaton* (DFA) is a well-studied problem of formal languages. A survey due to Watson [24] gives a broad introduction to most known minimization algorithms. The asymptotically most efficient algorithm for this problem was introduced in 1970 by Hopcroft [11, 12]. Hopcroft's algorithm is able to perform its task in time $O(|A| \log |A|)$, where A is the state set of the DFA. The algorithm was presented under the implicit assumption that the size of the input alphabet X of the DFA is a constant. Actually, the Algol 60 code found in [11] is specialized to the case $|X| = 2$. Blum [2] gives a new and detailed implementation of the algorithm.

E-mail address: knuutila@cs.utu.fi (T. Knuutila).

0304-3975/01/$ - see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0304-3975(99)00150-4

It is to be noted that there are some special cases of DFA for which more efficient algorithms are known: *acyclic* DFA can be minimized in time $O(|X||A|)$ [19] and *single-letter alphabet* DFA in time $O(|A|)$ [17]. There exist also more general partition refinement algorithms (of similar time complexity) from which the DFA minimization can be drawn as a special case [5, 16].

Although Hopcroft's algorithm is a remarkable improvement over the classical Moore algorithm and its later refinements, only the classical ones are usually given in the literature. The more advanced algorithm is typically either totally omitted or just mentioned briefly with the standard references. For example, Watson's survey [24] skips the correctness proof and running time analysis because they are considered too complicated. The only textbooks to the author's knowledge attempting to clarify the ideas of the Hopcroft algorithm are the ones by Brauer [3] and Mikolajczak [15]. Both of these are more or less just repeating Hopcroft's original article or its re-description due to Gries [9].

Gries criticized the correctness proof and run-time analysis of Hopcroft [12], and aimed to present Hopcroft's algorithm in an understandable way with the help of the then (and even now) popular structured top-down approach. Though Gries unarguably managed to describe the algorithm and the different analyses more formal and clear, there is still room for some development – at least these clarifications have not lead to a general adoption of the algorithm in textbooks. Even in the ones mentioned above the exposition has remained unreadable. The best text in the author's opinion on Hopcroft's minimization algorithm is in [1], but there it is given under the title of partitioning a set with respect to a single function on this set (a task for which even a linear time algorithm is nowadays known), and the more general case needed to minimize a DFA is left as an exercise. It will turn out that this exercise is not simple.

We give in this paper yet another presentation that actually leads us to an algorithm very similar to Hopcroft's. Our goal is to perform a 'tutorial reconstruction' with clear and exact correctness proofs and computational analyses. Unlike earlier texts on the subject using 'guess and verify'-techniques in the time analysis, we are able to present the complexity analysis in a constructive way. Our analysis reveals that the later texts [9, 3, 15] concerning Hopcroft's original algorithm have made the quick and unjustified conclusion that this algorithm has time complexity $O(|X||A|\log|A|)$ for an arbitrary alphabet X. This bound can, however, be achieved by applying a simple enhancement.

The rest of the article is constructed as follows. Section 2 contains the basic definitions and notation needed in the sequel. We also review some of the most fundamental properties of DFA with the emphasis on the ones concerning state equivalence and minimization. After these preliminaries we develop, implement and analyze an efficient minimization algorithm in Sections 3, 4 and 5, respectively. All this work leads to a common algorithmic framework over existing variants of the Hopcroft algorithm, and a common analysis tool, *colored derivation tree*, that will be of central importance both in the correctness proofs and time analyses. Section 6 concludes the article with discussion on practical aspects and possible extensions.

2. Preliminaries

We define formally the concepts of *strings* and *DFA*. The results presented here are just stated without proofs, which can be found in most textbooks on the subject (e.g. [20, 13]).

2.1. Sets, relations and mappings

We present here some elementary concepts that can be just scanned for terminology and notation.

The *cardinality* of a set A is denoted by $|A|$. Let A and B be sets and $\rho \subseteq A \times B$ a (binary) *relation* from A to B. The fact that $(a,b) \in \rho$ $(a \in A, b \in B)$ is also expressed by writing $a\rho b$. For any $a \in A$, we denote by $a\rho$ the set of elements of B that are in relation ρ with a, i.e. $a\rho = \{b \in B \mid a\,\rho\, b\}$. The *converse* of ρ is the relation $\rho^{-1} = \{(b,a) \mid a\,\rho\, b\}$. Obviously $b\rho^{-1} = \{a \mid a\,\rho\, b\}$.

Next we consider *relations on a set* A, i.e. subsets of $A \times A$. These include the *diagonal relation* $\omega_A = \{(a,a) \mid a \in A\}$, and the *universal relation* $\iota_A = A \times A$. The *powers* ρ^n $(n \geqslant 0)$ of a relation ρ are defined as follows: $\rho^0 = \omega_A$, and $\rho^{n+1} = \rho^n \circ \rho$ for $n \geqslant 0$. The relation ρ is called *reflexive* if $\omega_A \subseteq \rho$; *symmetric* if $\rho^{-1} \subseteq \rho$; and *transitive* if $\rho^2 \subseteq \rho$.

A relation on A is called an *equivalence relation* on A, if it is reflexive, symmetric and transitive. The set of all equivalence relations on a set A is denoted by $\mathrm{Eq}(A)$. It is obvious that both $\omega_A \in \mathrm{Eq}(A)$ and $\iota_A \in \mathrm{Eq}(A)$. Let $\rho \in \mathrm{Eq}(A)$. The *ρ-class* $a\rho$ of an element a $(\in A)$ is also denoted by a/ρ. The *quotient set* or the *partition* of A with respect to ρ, is $A/\rho = \{a\rho \mid a \in A\}$. If $\pi \in \mathrm{Eq}(A)$ and $\pi \subseteq \rho$, then the partition A/π is a *refinement* of A/ρ (each ρ-class is a union of some π-classes); this is also expressed by saying that π is *finer* than ρ or that ρ is *coarser* than π. We often define an equivalence relation ρ on A via the set A/ρ, i.e. in the form $A/\rho = \{C_1, \ldots, C_m\}$, where the sets C_i are the classes of ρ.

The cardinality of A/ρ is called the *index* of ρ; especially, if $|A/\rho|$ is finite, then ρ is said to have a *finite index*. For any subset H of A, H/ρ denotes $\{a\rho \mid a \in H\}$. The equivalence ρ *saturates* the subset H, if H is the union of some ρ-classes. Hence, ρ saturates H iff $a\rho \in H/\rho$ implies $a\rho \subseteq H$.

A *mapping* or a *function* $\phi : A \to B$ is a relation $\phi \subseteq A \times B$ such that $|a\phi| = 1$ for all $a \in A$. If $a\phi b$, then b is the *image* of a and a is a *preimage* of b. That b is an image of a is expressed by writing $b = \phi(a)$ and that a is a preimage of b is written as $a \in \phi^{-1}(b)$. The notation $a = \phi^{-1}(b)$ is justified, when ϕ is bijective (see below). The *restriction* of a mapping $\phi : A \to B$ to a set $C \subseteq A$ is the mapping $\phi|C : C \to B$ where $\phi|C = \phi \cap (C \times B)$.

The *composition* of two mappings $\phi : A \to B$ and $\psi : B \to C$ is the mapping $\phi\psi : A \to C$, where $\phi\psi$ is the product of ϕ and ψ as relations. The *kernel* $\phi\phi^{-1}$ of a mapping $\phi : A \to B$, also denoted by $\ker \phi$, is an equivalence relation on A and $a\,\phi\phi^{-1}\,b$ iff $a\phi = b\phi$ $(a, b \in A)$. A mapping $\phi : A \to B$ is called *injective* if

$\ker \phi = \omega_A$; *surjective* (or *onto*) if $A\phi = B$; and *bijective* if it is both injective and surjective.

2.2. Recognizable string languages and DFA

An *alphabet* is a finite nonempty set of *letters*. In what follows X always denotes an alphabet. A finite sequence of letters from an alphabet X is called a *string* over X. Consider a string w of the form $w = x_1 x_2 \ldots x_n$, where each $x_i \in X$. If $n = 0$, then w is the *empty string*, denoted by ε. The *length* of w, written as $|w|$, is n. Especially, $|\varepsilon| = 0$. The set of all strings over X is denoted by X^*. A *language* over X, or an X-*language* is any subset of X^*. We will later on assume that X is clear from the context and talk only about *languages*.

Definition 1. A *deterministic finite automaton* (*DFA*), $\mathfrak{A} = (A, X, \delta, a_0, A')$ consists of
(1) a finite, nonempty set A of *states*,
(2) the *input alphabet* X,
(3) a *transition function* $\delta : A \times X \to A$,
(4) an *initial state* $a_0 \in A$, and
(5) a set $A' \subseteq A$ of *final states*.

The function δ is extended to a function $\hat{\delta} : A \times X^* \to A$ as usual (we will omit the cap from $\hat{\delta}$ in the sequel): $\hat{\delta}(a, \varepsilon) = a$, and $\hat{\delta}(a, xw) = \hat{\delta}(\delta(a, x), w)$ ($a \in A$, $w \in X^*$, $x \in X$). The *language recognized* by a DFA $\mathfrak{A}$ is now defined as

$$L(\mathfrak{A}) = \{w \in X^* \mid \hat{\delta}(a_0, w) \in A'\}.$$

A language L is *recognizable*, if there exists a DFA $\mathfrak{A}$ such that $L = L(\mathfrak{A})$. Two DFA $\mathfrak{A}$ and $\mathfrak{B}$ are said to be *equivalent* iff $L(\mathfrak{A}) = L(\mathfrak{B})$.

2.3. Minimal DFA

DFA $\mathfrak{A}$ is said to be *minimal*, if no DFA with fewer states recognizes $L(\mathfrak{A})$. The minimal DFA for any recognizable language is unique up to isomorphism. It is also well-known that given an arbitrary DFA $\mathfrak{A}$, one can effectively construct the minimal DFA equivalent to $\mathfrak{A}$. We review here briefly the classical construction.

DFA $\mathfrak{A} = (A, X, \delta, a_0, A')$ is said to be *connected*, if $A = \{\delta(a_0, w) \mid w \in X^*\}$. We assume hereafter that our DFA are connected (the construction of a connected equivalent DFA from a given one is trivial).

Two states a and b of $\mathfrak{A}$ are *equivalent*, which we express by writing $a \rho_{\mathfrak{A}} b$, if

$$(\forall w \in X^*)\ (\delta(a, w) \in A' \Leftrightarrow \delta(b, w) \in A').$$

DFA $\mathfrak{A}$ is *reduced*, if $a \rho_{\mathfrak{A}} b$ implies $a = b$, i.e. $\rho_{\mathfrak{A}} = \omega_A$.

A relation $\rho \in \mathrm{Eq}(A)$ is a *congruence of* $\mathfrak{A}$, if
(1) $a \rho b$ implies $\delta(a,x) \rho \delta(b,x)$ for all $a,b \in A$ and all $x \in X$, and
(2) ρ saturates A'.

We denote by $\mathrm{Con}(\mathfrak{A})$ the set of all congruences of $\mathfrak{A}$. It is well-known that the relation $\rho_{\mathfrak{A}}$ is the greatest (coarsest) congruence of $\mathfrak{A}$. For any $\rho \in \mathrm{Con}(\mathfrak{A})$, the *quotient DFA* $\mathfrak{A}/\rho$ is defined as $\mathfrak{A}/\rho = (A/\rho, X, \delta/\rho, a_0/\rho, A'/\rho)$, where $\delta/\rho(a/\rho, x) = \delta(a,x)/\rho$.

We state next a few facts showing how the minimal DFA can be constructed from a given one.

Proposition 2. *Let* $\mathfrak{A} = (A, X, \delta, a_0, A')$ *be a DFA and* $\rho \in \mathrm{Con}(\mathfrak{A})$. *Then* $L(\mathfrak{A}) = L(\mathfrak{A}/\rho)$, *and in particular*, $L(\mathfrak{A}) = L(\mathfrak{A}/\rho_{\mathfrak{A}})$.

Proposition 3. *Let* $\mathfrak{A} = (A, X, \delta, a_0, A')$ *be a DFA. Then the quotient DFA* $\mathfrak{A}/\rho_{\mathfrak{A}}$ *is minimal.*

2.4. DFA as unary algebras

The exposition of the following material is simplified somewhat by regarding DFA as unary algebras as proposed by Büchi and Wright (cf. [4], for example). This means that X is viewed as a set of unary operation symbols and the transition function δ of $\mathfrak{A} = (A, X, \delta, a_0, A')$ is replaced by the X-indexed family $(x^{\mathfrak{A}} : x \in X)$ of unary operations which are defined so that for any $x \in X$ and $a \in A$, $x^{\mathfrak{A}}(a) = \delta(a,x)$. We shall omit $\mathfrak{A}$ from the superscript and write simply $x(a)$ for $\delta(a,x)$. Clearly, a string $w = x_1 x_2 \ldots x_n$ is accepted by $\mathfrak{A}$ if and only if $x_n(\ldots(x_2(x_1(a_0)))\ldots) \in A'$.

3. Developing an efficient minimization algorithm

The process of minimizing a DFA $\mathfrak{A}$ is essentially the same as the computation of the relation $\rho_{\mathfrak{A}}$.

3.1. The classical algorithm

The classical minimization algorithm is based on the following 'layer-wise' definition for the equivalence relation $\rho_{\mathfrak{A}}$.

Proposition 4. *Let* $\mathfrak{A} = (A, X, \delta, a_0, A')$ *and a series* ρ_i $(i \geqslant 0)$ *of equivalence relations on A be defined as follows*:

$$\rho_0 = \{(a,b) \mid a,b \in A'\} \cup \{(a,b) \mid a,b \in A - A'\},$$

$$\rho_{i+1} = \{(a,b) \in \rho_i \mid (\forall x \in X)\ (x(a), x(b)) \in \rho_i\}.$$

Then the following hold.

(1) $\rho_0 \supseteq \rho_1 \supseteq \cdots$.

(2) *If* $\rho_i = \rho_{i+1}$ *then* $\rho_i = \rho_{i+j}$ *for all* $j > 0$ *and furthermore,* $\rho_i = \rho_{\mathfrak{A}}$.

(3) *There exists* $0 \leqslant k \leqslant |A|$ *such that* $\rho_k = \rho_{k+1}$.

When the construction of Proposition 4 is implemented, each refinement step leading from ρ_i to ρ_{i+1} consists of a series of refinements on individual classes of ρ_i. The refinement of a single class B can be implemented with suitably chosen data structures for equivalence relations to run in time $O(|X||B|)$, and each refinement from ρ_i to ρ_{i+1} takes thus time $O(|X||A|)$.

The classical minimization algorithm is ineffective, since we can construct (the worst kind of) a DFA $\mathfrak{A} = (\{a_1, \ldots, a_m\}, X, \delta, a_1, \{a_m\})$, where $x(a_i) = a_{i+1}$ for $1 \leqslant i \leqslant m-1$ and $x(a_m) = a_m$ $(x \in X)$, and $y(a_i) = a_i$ for all $a_i \in A$ and $y \in X$, $y \neq x$. Now $A/\rho_0 = \{\{a_m\}, A - \{a_m\}\}$, $A/\rho_{\mathfrak{A}} = \omega_A$, and each refinement step from ρ_i to ρ_{i+1} removes always one state from the only nonsingleton class. The work performed by the algorithm (even if we optimize it to avoid considering singleton classes) will then be proportional to $|X| \sum_{i=1}^{|A|-2} (|A| - i)$, which leads to $O(|X||A|^2)$ execution time.

3.2. *Atomic refinements* 原子细化，实际上是划分的每个block的refine

The classical approach just described represents the end of one line of evolution starting from Proposition 4. Due to the facts noted, it cannot be made more efficient by just using smarter and more efficient data structures. In order to develop an asymptotically faster minimization algorithm, we must try to find other ways to perform the construction.

Note that the construction of Proposition 4 reaches the situation $\rho_i = \rho_{i+1}$ when ρ_i becomes a congruence, i.e. $(\forall a, b \in A,\ x \in X)\ a\,\rho_i\,b \Rightarrow x(a)\,\rho_i\,x(b)$. Now consider the situation where $\rho_i \neq \rho_{i+1}$. Obviously,

$$\begin{aligned}
\rho_i \neq \rho_{i+1} \quad &\Leftrightarrow \quad (\exists a, b \in A,\ x \in X) && (a,b) \in \rho_i \text{ and } (x(a), x(b)) \notin \rho_i \\
&\Leftrightarrow \quad (\exists B \in A/\rho_i,\ x \in X) && a, b \in B \text{ and } (x(a), x(b)) \notin \rho_i \\
&\Leftrightarrow \quad (\exists B, C \in A/\rho_i,\ x \in X) && a, b \in B \text{ and } x(a) \in C \text{ and } x(b) \notin C \\
&\Leftrightarrow \quad (\exists B, C \in A/\rho_i,\ x \in X) && x(B) \cap C \neq \emptyset \text{ and } x(B) \not\subseteq C.
\end{aligned}$$

The step from ρ_i to ρ_{i+1} can thus be understood as a series of 'atomic' refinements performed for such $x \in X$, $B, C \in A/\rho_i$ that

$$x(B) \cap C \neq \emptyset \quad \text{and} \quad x(B) \not\subseteq C \tag{1}$$

holds. Each such atomic refinement partitions then B into $B \cap x^{-1}(C)$ and $B - (B \cap x^{-1}(C))$. We denote these refinements of B with $B_{C,x}$ and $B^{C,x}$, respectively. We use the notation B' and B'' for these refinements of B when their relation to the pair (C,x) has no importance.

Let us next rewrite Proposition 4 in the terms of these atomic refinements.

Proposition 5. *Let* $\mathfrak{A}=(A,X,\delta,a_0,A')$ *be a DFA and a series* θ_i $(i\geqslant 0)$ *of equivalence relations on* A *be defined as follows*:

$$A/\theta_0=\{A',A-A'\},$$
$$A/\theta_{i+1}=\begin{cases}(A/\theta_i-\{B\})\cup\{B_{C,x},B^{C,x}\} & \text{if (1) holds for some } B,C\in A/\theta_i,\\ & x\in X,\\ A/\theta_i & \text{otherwise.}\end{cases}$$

Then there exists a $k\leqslant|A|$ *such that* $\theta_{k+l}=\theta_k$ *for all* $l\geqslant 0$ *and* $\theta_k=\rho_{\mathfrak{A}}$.

Proof. The upper bound comes from the observation, that each atomic refinement increases the index of θ_i by one. Naturally this index cannot be increased more than $|A|-1$ times.

It is clear that θ_k is both an equivalence relation saturating A' (it is a refinement of θ_0) and a congruence of $\mathfrak{A}$ (since Eq. (1) does not hold for θ_k). What remains, is to show that θ_k is also the *greatest* congruence $\rho_{\mathfrak{A}}$ of $\mathfrak{A}$.

We show first that $\theta_i\supseteq\rho_{\mathfrak{A}}$ for all $i\geqslant 0$. When contraposed, the claim is that if $(a,b)\notin\theta_i$ then $(a,b)\notin\rho_{\mathfrak{A}}$ (for all $i\geqslant 0$). This is clearly true for θ_0, since final and non-final states are not in $\rho_{\mathfrak{A}}$. Suppose then that the claim holds for all $0\leqslant l\leqslant i$ and let $(a,b)\in\theta_i$. If $(a,b)\notin\theta_{i+1}$, it must be the case that for some $x\in X$, $(x(a),x(b))\notin\theta_i$. But this implies (by IA) that $x(a)$ and $x(b)$ are not in $\rho_{\mathfrak{A}}$. Thus, a and b become inequivalent in θ_{i+1} only when they are shown to be inequivalent in $\rho_{\mathfrak{A}}$, formally $(a,b)\notin\theta_{i+1}\Rightarrow(a,b)\notin\rho_{\mathfrak{A}}$.

It now holds that $\theta_0\supseteq\theta_1\supseteq\cdots\supseteq\theta_k$ (by definition), and consequently $\theta_0\supseteq\theta_1\supseteq\cdots\supseteq\theta_k\supseteq\rho_{\mathfrak{A}}$. Since $\rho_{\mathfrak{A}}$ is the greatest congruence of $\mathfrak{A}$, and θ_k is a congruence, it must be the case that $\theta_k=\rho_{\mathfrak{A}}$.

Note that the construction given in Proposition 5 does not fix the order in which the triples B, C, x are exploited to refine θ_i. Thus, all the different orderings yield the same result at the end: the unique greatest congruence.

3.3. A change of view

Algorithm 1 (*Computing* $\rho_{\mathfrak{A}}$ *using atomic refinements*)

EQUIVALENCE($\mathfrak{A}$)
1 $A/\theta\leftarrow\{A',A-A'\}$
2 **while** $(\exists B,C\in A/\theta,\ x\in X)$ *s.t. Eq.* (1) *holds* **do**
3 $\quad A/\theta\leftarrow(A/\theta-\{B\})\cup\{B_{C,x},B^{C,x}\}$
4 **return** θ

Proposition 5 leads directly to Algorithm 1. As such, it is yet rather abstract, and in particular we have to decide how to efficiently find some triple B,C,x for which Eq. (1) holds. Given some class $B\in A/\theta$ it seems natural to try to consider, instead

of $(A/\theta)\times X$, only those class-letter pairs (C,x) for which some B exists such that Eq. (1) holds. Let us call these pairs as *refiners of B in θ*, shortly $ref(B,\theta)$:

$$ref(B,\theta)=\{(C,x)\in(A/\theta)\times X \mid x(B)\cap C\neq\emptyset \text{ and } x(B)\not\subseteq C\}.$$

The crucial change of view made in [11] was that, instead of iterating over the classes B that *get* refined, we iterate over the refiners (C,x) *causing* the classes B to get refined.

As each B is related to $ref(B,\theta)$, so is each pair (C,x) related to its *objects of refinement* in θ, $obj(C,x,\theta)=\{B\in A/\theta \mid (C,x)\in ref(B,\theta)\}$. The definition of $obj(C,x,\theta)$ leads us directly to Algorithm 2, which is easily shown to implement the construction of Proposition 5. It just avoids considering all the irrelevant classes B which would not get refined with respect to pair (C,x).

Algorithm 2 (*Refiner-driven implementation*)

EQUIVALENCE($\mathfrak{A}$)
1 $A/\theta \leftarrow \{A', A-A'\}$
2 **while** *some* $(C,x)\in(A/\theta)\times X$ *with* $obj(C,x,\theta)\neq\emptyset$ *exists*
3 **for** $B\in obj(C,x,\theta)$ **do**
4 *replace* B *with* $B_{C,x}$ *and* $B^{C,x}$ *in* A/θ
5 **return** θ

We next consider the selection of the pairs (C,x) for which $obj(C,x,\theta)\neq\emptyset$, or equivalently, the rejection of the pairs for which the opposite holds. Suppose we start with all the pairs created from $A/\theta=\{A', A-A'\}$ and X, pick some (C,x) and refine the classes in $obj(C,x,\theta)$. The following lemma (from [9]) tells us that (in the case C remains unmodified) at least (C,x) itself can be counted out in the next iterations.

Lemma 6. *Let $\mathfrak{A}=(A,X,\delta,a_0,A')$, $\theta\in Eq(A)$, $B,C\in A/\theta$ and $x\in X$. Suppose we refine $B\in A/\theta$ into $B_{C,x}$ and $B^{C,x}$ with respect to (C,x). Let D be a subset of $B_{C,x}$ or $B^{C,x}$. Then $D\notin obj(C,x,\theta)$.*

Proof. Let us first consider the case $D\in\{B_{C,x},B^{C,x}\}$. Then either $x(a)\in C$ for all $a\in D$ or $x(a)\notin C$ for all $a\in D$. The same property holds naturally for all subsets of D.

The bookkeeping needed for telling whether a particular pair (C,x) has been used can be implemented as follows:

- We maintain a set L of *candidate refiners*, shortly *candidates*, in $(A/\theta)\times X$. Initially $L=\{A', A-A'\}\times X$.
- Pairs (C,x) are now selected from L, not (blindly) from all of $(A/\theta)\times X$.
- Every time we select a pair (C,x) from L, we *remove* it from L. This is justified by Lemma 6.

- After refining θ to θ', we must also update L to contain only classes of θ'. We do the following for each $x \in X$ and each class B refined to B' and B'':
  - If $(B,x) \in L$, we remove (B,x) and add both (B',x) and (B'',x) to L. This is (indirectly) justified by Proposition 5, since only the current θ-classes are used in the construction.
  - If $(B,x) \notin L$, we simply add (B',x) and (B'',x) to L.

Note that new items are inserted into L *only* when some class gets refined. Thus, L will eventually become empty, because each iteration removes one element from L, and the total number of elements added into L is bounded by $2|X||A|$ (the maximum number of different equivalence classes created in any refinement sequence is $2|A|-1$). The ideas above are collected into Algorithm 3.

Algorithm 3 (*Set-driven implementation*)

```
EQUIVALENCE(𝔄)
1  A/θ ← {A', A − A'}
2  L ← (A/θ) × X
3  while L ≠ ∅ do
4      remove a pair (C,x) from L
5      for each B ∈ obj(C,x,θ) do
6          replace B with B_{C,x} and B^{C,x} in A/θ
7          for each y ∈ X do
8              if (B, y) ∈ L then
9                  replace (B, y) with (B', y) and (B'', y) in L
10             else
11                 insert (B', y) and (B'', y) to L
12 return θ
```

3.4. Derivation trees

Consider some $\mathfrak{A} = (A, X, \delta, a_0, A')$ and the execution of Algorithm 3. At each iteration of the main loop, the refinement process leading from the initial θ_0 to the current θ can be described by a binary tree DT_θ, the *derivation tree of* θ.

- Each node of DT_θ is labeled with a subset of A. We denote by $DT_\theta(B)$ the subtree rooted at a node labeled with B.
- The label of the root is A, and its (immediate) descendants are labeled with A' and $A - A'$.
- The leaves of the tree are labeled with the (current) classes of θ.
- Each intermediate node labeled B has two descendants labeled with B' and B''.

Note that the exact shape of DT_θ depends on the order in which the candidates are selected from L. However, all orderings lead eventually to the same result, namely $\rho_{\mathfrak{A}}$.

3.5. Colored derivation trees

We introduce here a marking technique that helps us to establish properties of derivation trees in a compact manner. It is to be noted that these marks, *colors*, are not used directly in our algorithms but induced by its execution.

The set L is manipulated via the following operations during the execution of Algorithm 3: deletion (line 4), insertion (line 11), and replacement (line 9). These operations can be indicated in DT_θ by allowing each $x \in X$ to *color* the nodes of DT_θ. The color assigned by x to some node B tells us what is the operational history behind the pair (B,x).

For each pair (B,x), where B is in DT_θ and $x \in X$, we define the *x-color of B*, $Col(x)(B)$ (the curried form is used here on purpose), to be one of the following:

- *black* $(B,x) \in L$, because it was *inserted* (lines 2, 9, and 11).
- *green* $(B,x) \notin L$, because it was *removed* (line 4).
- *amber* $(B,x) \notin L$, because it was *replaced* (line 9).

Note that if $Col(x)(B) = amber$ for some $x \in X$, then B is necessarily an inner node of DT_θ, and if $Col(x)(B) = black$, then B must be a leaf node. Green nodes may appear both as leaves and as inner nodes. The root of the tree may be colored green for all $x \in X$, although (A,x) has not to be used in the refinement process (it will not refine any class).

3.6. Reducing the number of candidates

Consider once again the execution of Algorithm 3, where some class B (leaf of DT_θ) is refined with respect to some (C,x). Then DT_θ will change in such a way that B is given two descendants, which are colored black for each letter $y \in X$. $Col(x)(B)$ either stays green or changes from black to amber. Note that the latter case implies that of the triple $(B,y),(B',y),(B'',y)$ only the last two are (possibly) used as future refiners. The fact that this is exactly the way in which Proposition 5 works implies immediately the following corollary.

Corollary 7. *Let* $\mathfrak{A} = (A,X,\delta,a_0,A')$, $\theta \in Eq(A)$ *and* $B \in A/\theta$. *Suppose we refine B into* B' *and* B''. *Then, for any* $x \in X$, *refining all the classes of* θ *with respect to* (B',x) *and* (B'',x) *yields the same result as refining* θ *with respect to* (B,x), (B',x) *and* (B'',x).

Now consider DT_θ, $y \in X$ and the nodes B for which $Col(y)(B) = green$. If, after refining θ with respect to (B,y), B itself gets refined, we add both (B',y) and (B'',y) into L. However, as seen in Fig. 1, from the viewpoint of any $D \subseteq A$, (B,y) refines D to parts $\{D_1 \cup D_2, D_3\}$. Now clearly refining $\{D_1 \cup D_2, D_3\}$ with either (B',y) or (B'',y) gives the same result, the set $\{D_1,D_2,D_3\}$. Combining Corollary 7 and this remark we get the following Lemma (a modification of Lemma 6 in [9]).

Lemma 8. *Let* $\mathfrak{A} = (A,X,\delta,a_0,A')$, $\theta \in Eq(A)$ *and* $B \in A/\theta$. *Suppose we refine B into* B' *and* B''. *Then, for any* $y \in X$, *refining all the classes of* θ *with respect to any two*

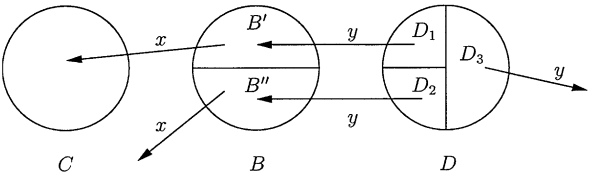

Fig. 1. Illustration of Lemma 8.

of the pairs (B, y), (B', y) *and* (B'', y) *gives the same result as refining them with respect to all three of them.*

Proof. Consider an arbitrary class $D \in A/\theta$, $a \in D$ and $y \in X$. As seen in Fig. 1, the transition $y(a)$ satisfies exactly one of the following: $y(a) \in B'$ (1), $y(a) \in B''$ (2), or $y(a) \notin B$ (3). Now each possible refinement sequence with respect to any two of the sets B, B' and B'' and letter y partitions D into D_1, D_2, D_3, which is the same as the result yielded by performing all the three refinements.

Corollary 9. *Let* $\mathfrak{A} = (A, X, \delta, a_0, A')$, $\theta \in Eq(A)$, $x \in X$, $B \in A/\theta$, *and* $\{B', B''\}$ *a refinement of* B. *If* $obj(B, x, \theta) = \emptyset$ *and* $obj(B', x, \theta) = \emptyset$ *then* $obj(B'', x, \theta) = \emptyset$.

Corollary 9 can be exploited to enhance Algorithm 3 as follows: when we add new refiners (B', y) and (B'', y) into L, we first check, whether $Col(y)(B)$ is green. If this is the case, then only either one of the pairs has to be added. This modification leads to a new kind of nodes B in DT_θ: those for which a pair (B, x) has *never* been added to L. We augment our coloring to include also this case by using a new color *red* and defining $Col(x)(B) = red$ for these nodes.

Finally, the following corollary will be of use when initializing our list L.

Corollary 10. *Let* $\mathfrak{A} = (A, X, \delta, a_0, A')$, $A/\theta = \{A', A - A'\}$, *and* $x \in X$. *Then refining* θ *with respect to either* (A', x) *or* $(A - A', x)$ *yields the same result as refining* θ *with respect to both of them.*

When we have the freedom of making a choice between two candidates, which one of them should be included in L? The natural basis for this selection is the amount of computation caused by the alternatives. However, we postpone the efficiency issues for the moment, and simply assume we have an auxiliary function AddBetter(B', B'', x, L) performing the informed update of L.

3.7. Reducing further the number of candidates

In order to apply Corollary 9 to enhance our minimization algorithm we have to check somehow whether $Col(y)(B) = green$ when considering pairs (B', y) and (B'', y). One possible way to perform this checking would be to just implement the correspond-

ing colored trees and inspect them. Hopcroft, however, simply tests in [12] whether $(B, y) \in L$ or not.

At the first glimpse this simple test seems not to be enough, since it may be the case that $Col(y)(B) = red$, and the 'better sibling' of B was added to L with y. However, if we just knew that the refinement process performed by the parent class of B and y had been done, then we could again use Lemma 8 to justify the addition of only either of B' and B''. In general, if we had the right to – in addition to taking only either one of the refinements of the green nodes – take only either one of the refinements of the red ones, then the simple check $(B, y) \in L$ would suffice.

The required justification will be given in the following section (Theorem 14). Before that, let us rewrite Algorithm 3 using the optimizations discussed. Algorithm 4 uses the test $(B, y) \in L$? to control the addition process, and it uses subprogram AddBetter to make the choice when we have one. We have split the loop over the set $obj(C, x, \theta)$ into two parts (lines 6 and 8) in order to make it resemble more the final implementation (see next section) where this separation yields some computational benefits.

Algorithm 4 (*Optimized set-driven algorithm*)

```
Equivalence(𝔄)
1  A/θ ← {A', A − A'}
2  L ← ∅
3  for x ∈ X do AddBetter(A', A − A', x, L)
4  while L ≠ ∅ do
5      remove a pair (C, x) from L
6      for each B ∈ obj(C, x, θ) do
7          replace B with B_{C,x} and B^{C,x} in A/θ
8      for each B just refined do
9          for each y ∈ X do
10             if (B, y) ∈ L then
11                 replace (B, y) with (B', y) and (B'', y) in L
12             else
13                 AddBetter(B', B'', y, L)
14 return θ
```

Example 11. Let us trace the execution of Algorithm 4 with the DFA

$$\mathfrak{A} = (\{a_1, \ldots, a_8\}, \{x, y\}, \delta, a_1, \{a_8\}),$$

where $x(a_i) = a_{i+1}$ for $1 \leqslant i \leqslant 7$ and $x(a_8) = a_8$, and $y(a_i) = a_i$ for all $a_i \in A$. In the beginning we have $A/\theta = \{B_1, B_2\}$ where $B_1 = \{a_7\}$ and $B_2 = \{a_1, \ldots, a_6\}$.

Let us use the sizes of classes as our evaluation measure in routine AddBetter. Since the root node A is treated as green for all letters, we get $Col(x)(B_1) = Col(y)(B_1) = \text{black}$ and $Col(x)(B_1) = Col(y)(B_2) = red$. Suppose the algorithm chooses the pair (B_1, y) as a candidate. Then $obj(B_1, y, \theta) = \{B_1\}$ and no refinement is done. However,

$Col(y)(B_1) = green$ after using the candidate. The only pair remaining in L is (B_1, x) and $obj(B_1, x, \theta) = \{B_1, B_2\}$. Now B_2 is refined to $B_3 = \{a_6\}$ and $B_4 = \{a_1, \ldots, a_5\}$. Updates of L add pairs (B_3, x) and (B_3, y) into L leaving $Col(x)(B_4) = Col(y)(B_4) = red$. The process continues similarly until only singleton classes are left.

3.8. Properties of derivation trees

We are next to show that Algorithm 4 still computes $\rho_{\mathfrak{A}}$. Before that we make some remarks on the colors of the nodes our latest algorithm assigns to the nodes of DT_θ. It is straightforward to show that the following properties hold.

(1) When a green or red node is refined ($(B, y) \notin L$), its descendants are colored black and red (line 13).
(2) Black nodes may change to green (line 5) or amber (line 11). In the latter case both descendants of the node are colored black.
(3) Green, red, and amber nodes do not change their color.
(4) Immediate ancestors of red nodes are either red or green.
(5) Each red node has a green ancestor (recall that the root of DT_θ is colored green for all $x \in X$).

Definition 12. Let $x \in X$, $B \in DT_\theta$, and $Col(x)(B) \in \{amber, green\}$. We define the *green fringe* of B in DT_θ, $gf(B, x)$ as follows.

$$gf(B,x) = \begin{cases} \{B\} & \text{if } Col(x)(B) = green; \\ gf(B', x) \cup gf(B'', x) & \text{if } Col(x)(B) = amber. \end{cases}$$

Lemma 13. *Let $x \in X$, $B \in DT_\theta$, and $Col(x)(B) = amber$. Then*
(1) *$gf(B, x)$ is well-defined, and*
(2) *$gf(B, x)$ is a partition of B.*

Proof. The first claim is a direct consequence of the fact that the initially black (direct) descendants of B can later become only green or amber. Thus, each path emanating from B must contain a green node, and all the intermediate nodes in the path to first such node are amber. The second claim follows directly from the definition of $gf(B, x)$.

We are now ready to establish the correctness of Algorithm 4.

Theorem 14. *Let $\mathfrak{A} = (A, X, \delta, a_0, A')$ and θ be the relation returned by Algorithm 4. Then $obj(B, x, \theta) = \emptyset$ for all $B \in DT_\theta$ and $x \in X$.*

Proof. The proof branches on $Col(x)(B)$. Note that $Col(x)(B) \neq black$, since L is empty when the algorithm terminates.

- *green* The claim clearly holds, since (B, x) has been used to refine θ.
- *amber* Let $gf(B, x) = \{B_1, B_2, \ldots, B_m\}$ $(m \geqslant 2)$. We know that θ has been refined with all the pairs (B_i, x) $(1 \leqslant i \leqslant m)$ and thus $obj(B_i, x, \theta) = \emptyset$. Consequently (by a

simple extension of Corollary 9) $obj(B_1 \cup \cdots \cup B_m, x, \theta) = \emptyset$, and, since $gf(B,x)$ is a partition of B, we have that $obj(B,x,\theta) = \emptyset$.

- *red* Recalling the fact that B must have a green ancestor, let G be the nearest of them, and let $B = R_1, \ldots, R_m, G$ $(m \geqslant 1)$ be the path from B to G. Since $Col(x)(G) = green$ and $Col(x)(R_m) = red$, it must be the case that the sibling of R_m, say R', has $Col(x)(R') \neq red$. The cases above tell us that $obj(R', x, \theta) = \emptyset$. Since also $obj(G,x,\theta) = \emptyset$, it follows directly from Corollary 9 that $obj(R_m, x, \theta) = \emptyset$. The same reasoning can now be continued downwards the path by using R_m in the place of G to show that $obj(R_{m-1}, x, \theta) = \emptyset$ and, similarly, all the way down to $R_1 = B$.

Corollary 15. *The relation θ returned by Algorithm 4 is the greatest congruence of the given DFA $\mathfrak{A}$.*

3.9. Comparison to previous work

Hopcroft gives in [11, 12] proof on the property that the relation θ computed by the minimization algorithm is the greatest congruence of the input DFA. The proof is correct under the assumption that all relevant candidates are considered (directly or indirectly) in the algorithm. This assumption is neither stated explicitly nor established as a property of the algorithm.

Gries tackles in [9] the task of showing that the effect of refining the relation with respect to all candidates is actually achieved. This is done by establishing the following loop invariant (quotation rewritten to the formalism used here) for the main loop of the algorithm.

> If $(C,x) \notin L$ then, for all classes B, either (1) does not hold for B, C and x or we are assured by other means that it does not hold when the algorithm terminates.

The 'other means' are related to the execution of line 13 of Algorithm 4. If (B,x) is not in L but $Col(x)(B) = green$, it is clearly justified to add either of the refinements of B by Lemma 8. If, however, $Col(x)(B) = red$, we have to show, where our 'other means' come from: in [9] they are not given a formal description.

Aho and Ullman give a different proof in their textbook on algorithms [1]. When rewritten for the formalism used in this article and nonunary alphabets, their loop invariant can be stated as follows.

> Pair (B,x) where $B \subseteq A$ is said to be *safe* for θ if $obj(B,x,\theta) = \emptyset$. If $(B,x) \notin L$ then there is a set $\{B_1, \ldots, B_k\}$ such that $(B_1,x), \ldots, (B_k,x) \in L$ and $(B \cup B_1 \cup \cdots \cup B_k, x)$ is safe for θ.

The proof (in the restricted case of unary alphabets) found in [1] of the invariant above is lengthy and technical. Our proof of the same property for the general case (Theorem 14) is a dozen lines long yet intuitive (of course, the prerequisites take care of most of the content).

4. Implementation

Before we proceed to time analysis we have to address the implementation of the various data structures and operations needed by Algorithm 4. Here we also introduce different variants of the algorithm caused by different implementations of routine AddBetter.

4.1. Basic data structures

In what follows, we assume that each letter $x \in X$, state $a \in A$ and class $B \in A/\theta$ can be interpreted as a natural number. Then, for example, the transitions $x : A \to A$ are trivially implemented as a $|X| \times |A|$ matrix `trans`, where `trans[x][a]` contains the value of $x(a)$.

Most of the set-like structures we manipulate (e.g. $x^{-1}(a)$) are partitions of A. Furthermore, the only nontrivial operation on these sets is the refining of them. These remarks enable us to use a simple and efficient implementation where these sets are represented as *segments* of a fixed-size array. Segments are accessed via header elements containing the index of the first element and the size of the segment.

We use the following names for our partition structures:

- `inv_head` is an $|X| \times |A|$ matrix of headers of the segments $x^{-1}(a)$.
- `inv_elts` is an $|X| \times |A|$ matrix of segment elements.
- `cls_head` is an array of size $|A|$ of headers of equivalence classes.
- `cls_elts` is an array of size $|A|$ of segment elements.

For example, `inv_elts[x][inv_head[x][a].first]` contains the first element of $x^{-1}(a)$.

Each state $a \in A$ is linked to its class information via entries in array `states`, where each element `states[a]` contains the following fields:

- `cls_of`: the number corresponding to class a/θ; and
- `idx_of`: the index of a in its class segment (in array `cls_elts`).

The data structure implementing the set L has to support additions, deletions and membership tests. Since we are allowed (when choosing the current candidate) to remove an arbitrary element of L, we can use a very simple implementation where L is presented just as a list of integer pairs, and the membership tests are done using an $|A| \times |X|$ bit matrix `L_member`.

4.2. Refinement step

Consider the implementation of the refinement loop (lines 6 and 7) of Algorithm 4. We will save some space and time by re-using the names of the refined classes for either of their refinements. We must, however, be careful not to refine the class C of the current refiner (C,x) before all the classes intersecting $x^{-1}(C)$ have been considered. Thus, the refinement is implemented in two steps:

(1) Collect all the classes B that *might* be refined (those for which $x(B) \cap C \neq \emptyset$) into list `suspects`.
(2) Refine the classes in `suspects` (omit B for which $x(B) \cap C \subseteq C$).

In the implementation, we augment the class header records to contain (for each $B \in A/\theta$) fields `counter` for the number $|x^{-1}(C) \cap B|$, and `move` for the set (list) $x^{-1}(C) \cap B$. The algorithm will be composed in such a way that the counters are always 0 before the first (collection) step is performed.

The list of suspect classes intersecting $x^{-1}(C)$ and states that would be moved in the execution of the actual refinement are constructed by iterating over all the states a in $x^{-1}(C)$, and adding a into the list of states to be moved in future from a/θ. At the same time, we can add class a/θ in the suspect list (if not already there) and increase its `counter` field. Note that the maximum number of elements in all `move`-lists constructed above is at most $|A|$. Moreover, we can re-use the space needed for these lists at each iteration. After the whole set $x^{-1}(C)$ has been considered, a simple check (`counter = size`) tells us whether $x^{-1}(C) \cap B = B$ or not.

Algorithm 5 contains the implementation of procedure COLLECT performing this collection step. Routine INSERT used by COLLECT just places its first argument to the beginning of the second argument (list).

Algorithm 5 (*Collecting classes and states*)

```
COLLECT(x, C, suspects)
1   suspects ← nil; CH ← cls_head[C]
2   for i ← CH.first to  CH.first + CH.size do
3       inv_x_a ← inv_head[x][cls_elts[i]]
4       for j ← inv_x_a.first to  inv_x_a.first + inv_x_a.size do
5          b ← inv_elts[j]
6          B ← states[b].cls_of
7          if (cls_head[B].counter = 0) then
8              INSERT(B, suspects)
9              cls_head[B].move ← nil
10         cls_head[B].counter ← cls_head[B].counter + 1
11         INSERT(b, cls_head[B].move)
```

The implementation of the second step, procedure REFINE, where the actual refinement takes place is given in Algorithm 6. We have amalgamated the updates of L into this step, since here we actually know what classes are really refined, and, what are the (names of the) corresponding refinements. Updating L immediately after the creation of a new class releases us from maintaining yet another data structure relating sibling classes with each other. Note also that the re-using of old class names means that when both (B', x) and (B'', x) are to be inserted into L (descendants of amber nodes), we know that (B'', x) is already in L (since B'' uses the same class name as B). The implementation of routine SPLIT refining an individual class is discussed in the next section.

Algorithm 6 (*Performing the refinements*)

```
REFINE(suspects)
1   for each B in suspects do
2       if cls_head[B].counter < cls_head[B].size then
3         B' ← SPLIT(B, cls_head[B].move)
4         cls_head[B].counter ← 0
5         cls_head[B'].counter ← 0
6         for each x in X do
7            if L_member[B][x] then ADD(B', x, L)
8            else ADDBETTER(B, B', x, L)
```

4.3. Splitting an individual class

Routine REFINE calls SPLIT with a class B and a proper subset (list of states) B' of B. SPLIT is then supposed to update the relevant data structures in such a way that
(1) B is split into B' and $B - B'$,
(2) $B - B'$ is given the same name that was assigned to B, and
(3) B' is assigned a new class name.
Algorithm 7 shows the implementation of SPLIT. It exploits the widely used technique of a 'moving wheel', where one location of an array is used as a 'hole' for moving data in the array. In addition to the fact that the partitioning clearly takes time linear to the number of elements in the input list, the moving wheel approach is known to be very efficient in practice. New class names are drawn from a global variable `MaxClass`.

Algorithm 7 (*Splitting a class*)

```
SPLIT(B, move): Integer
1   Bnew ← MaxClass
2   MaxClass ← MaxClass + 1
3   hole ← cls_head[B].first
4   for each a in move do
5       apos ← states[a].idx_of
6       b ← cls_elts[hole]
7       states[a].idx_of ← hole
8       states[b].idx_of ← apos
9       cls_elts[hole] ← a
10      cls_elts[apos] ← b
11      states[a].cls_of ← Bnew
12      hole ← hole + 1
13  cls_head[B].first ← hole
14  cls_head[Bnew].first ← cls_head[B].first
15  cls_head[B].size ← cls_head[B].size − move.size
16  cls_head[Bnew].size ← move.size
17  return Bnew
```

4.4. Minimization algorithm

We have now implemented the actions needed in the main loop of the minimization algorithm. What remains, are the initialization actions required to construct the inverse mappings x^{-1}, the initial equivalence relation and list L, and a simple loop calling COLLECT and REFINE until L becomes empty. Algorithm 8 does exactly this.

It is assumed in the implementation that the input DFA `A` contains its transitions in `A.trans` and the set of final states in list `A.final` (suitable as an input for SPLIT) and that the first (universal) equivalence class is given name 1. After the algorithm terminates, the equivalence of states can be obtained either via `cls_head` or `states`, whichever is appropriate.

Algorithm 8 (*Final minimization algorithm*)

```
EQUIVALENCE(A)
1   -- construct the inverse of A.trans
2   θ ← A × A; MaxClass = 1
3   SPLIT(1, A.final)
5   for each x in X do ADDBETTER(1, 2, x, L)
6   while not EMPTY(L) do
7     C, x ← REMOVE(L)
8     COLLECT(x, C, suspects)
9     REFINE(suspects)
```

4.5. Variations

Here we finally address the selection criterion used in the updates of L, that is, the implementation of routine ADDBETTER, which is supposed to add the computationally more feasible of the pairs (B', y) and (B'', y) into L. Excluding the time required to execute the updates of L themselves, the work done in routines COLLECT and REFINE depends (mainly) on $|x^{-1}(C)|$. This number should thus be the most natural measure to use in the selection. The measures used in the literature are, however

- $|B'| \leqslant |B''|$ ([9, 1]), and
- $|x^{-1}(A) \cap B'| \leqslant |x^{-1}(A) \cap B''|$ ([12], also as an enhancement in [9]).

Let us write shortly $x^{-1} \cap B$ for $x^{-1}(A) \cap B$. Note that, for arbitrary B and x, both $|x^{-1}(B)| \geqslant |x^{-1} \cap B|$ and $|B| \geqslant |x^{-1} \cap B|$, whereas the relation of $|x^{-1}(B)|$ and $|B|$ is not fixed. In the updates of L, it may well be the case that $|x^{-1}(B')| < |x^{-1}(B'')|$ although $|B'| > |B''|$ or even when $|x^{-1} \cap B'| > |x^{-1} \cap B''|$ (only a few transitions enter B'), but also vice versa (many transitions enter a small subset of B''). Thus, these three criteria are not generally comparable.

4.6. Measure $|B|$

The implementation of the required test is simple (no other modifications are required in Algorithm 8):

```
AddBetter(B1, B2, x, L)
1   if cls_head[B1].size ⩽ cls_head[B2].size then Add(B1, x, L)
2   else Add(B2, x, L)
```

Note that with this measure a call Collect(x, C, suspects) does not necessarily take time proportional to $|x^{-1}(C)|$, because the first for-loop iterating over the elements of class C (line 3) may sometimes take more time than this. The reason for the extra work is, that it might happen that most $a \in C$ have $x^{-1} \cap \{a\} = \emptyset$, and, henceforth, $|C|$ supersedes $|x^{-1}(C)|$. A single execution of Collect takes thus time $O(\max(|C|, |x^{-1}(C)|))$. To our knowledge, this point was covered in somewhat disguised way in the analysis of Gries, but it seems to be forgotten in other texts.

4.6.1. Measures $|x^{-1} \cap B|$ and $|x^{-1}(B)|$

Measures $|x^{-1} \cap B|$ and $|x^{-1}(B)|$ have many common features and are thus covered in the same section. In particular, their efficient uses require the same additional data structures, namely the sets $x^{-1} \cap B$. This is because $x^{-1}(B)$ can be easily computed from $x^{-1} \cap B$ without notable computational overhead. A direct implementation would construct sets $x^{-1}(B)$ as segments of segment headers of $x^{-1}(a)$ for $a \in B$. Sets $x^{-1} \cap B$ are implemented just as our equivalence classes.

We store the segment headers of $x^{-1} \cap B$ and numbers $|x^{-1}(B)|$ in an $|X| \times |A|$ matrix named `in_class` containing the usual segment information (`first`, `size`) and $|x^{-1}(B)|$ in field `inv_size`. The segment elements are stored (for each letter) in a $|X| \times |A|$ matrix `in_c_elts`. Furthermore, we need an $|X| \times |A|$ matrix `in_c_idx_of` for the indices of states a within their segments. These entries contain some special number (say, -1) for $a \in A$ with $x^{-1}(a) = \emptyset$.

The routine AddBetter can now be written as below for measure $|x^{-1} \cap B|$. Note how our more informed measures guide the minimization algorithm better than the sizes $|B|$. In particular, if $|x^{-1} \cap B| = 0$ (and consequently $|x^{-1}(B)| = 0$), we do not have to insert pair (B,x) into L. These pairs can be treated as green nodes in our correctness analysis.

```
AddBetter(B1, B2, x, L)
1   if (in_class[x][B1].size = 0) or
2     (in_class[x][B2].size = 0) then
3     return
4   if (in_class[x][B1].size ⩽ in_class[x][B2].size) then
5     Add(B1, x, L)
6   else Add(B2, x, L)
```

For measure $|x^{-1}(B)|$ we just change line 4 to

```
4   if (in_class[x][B1].inv_size ⩽ in_class[x][B2].inv_size) then
```

With a set $x^{-1} \cap B$ available, the elements of the set $x^{-1}(B)$ can be effectively found by traversing the lists $x^{-1}(a)$ for each $a \in x^{-1} \cap B$. Note that $x^{-1}(a) \cap x^{-1}(b) = \emptyset$ for

all $a \neq b$ $(a, b \in A)$, since x is a mapping. Thus, we do not have to be afraid of visiting any state of $x^{-1}(B)$ more than once.

When some class B is refined to B' and B'', we must also refine $x^{-1} \cap B$ into $x^{-1} \cap B'$ and $x^{-1} \cap B''$ for each $x \in X$. At the same time, we can compute the numbers $|x^{-1}(B')|$ and $|x^{-1}(B'')|$. Supposing $B' = B_{C,x}$ we have that

$$|x^{-1}(B')| = \sum_{a \in B'} |x^{-1}(a)|,$$

$$|x^{-1}(B'')| = |x^{-1}(B)| - |x^{-1}(B')|.$$

The partitioning itself can be done as in Algorithm 7, the only change being that we skip those states of the `move` -list which are not in $x^{-1} \cap B$. Each update takes then $|X|$ iterations over each element of $x^{-1} \cap B'$.

Below is a list of other changes needed in the algorithm. The latter change gives us the extra enhancement that COLLECT now makes always $|x^{-1}(C)|$ iterations, since the useless elements of C are automatically skipped over.

- The sets $x^{-1} \cap B$ and numbers $|x^{-1}(B)|$ for $B \in \{A', A - A'\}$ must be initialized before the main loop of the algorithm begins. This initialization can be implemented to run in time linear in $|X||A|$.
- The first for-loop of COLLECT is rewritten as

```
InxC ← in_class[x][C]
for i ← InxC.first to InxC.first + InxC.size do
  inv_x_a ← inv_head[x][in_c_elts[x][i]]
```

5. Time analysis

It is easy to show that the initialization steps done before the main loop are bounded by $O(|A||X|)$ (in all variants). We also know from the previous discussion that the time spent in the updates of L is bounded by $O(|X||A|)$. The complexity of the remaining steps of the main loop is analyzed in the following parts:

- In routine COLLECT we have to consider separately its outer and inner loop when $|B|$ is used as the selection measure. For other measures, the inner loop is dominating.
- In routine REFINE we separate the time spent in constructing the refinements (equal to the inner loop of COLLECT) and the time needed to update the sets $x^{-1} \cap B$. Of course, the latter work is not needed when $|B|$ is used as the measure.

In what follows, we first point out some general properties of cost functions defined on derivation trees. These properties enable us to establish (for each $x \in X$) an $O(|A| \log |A|)$ bound for

- the work done in COLLECT and (consequently) in the refinement part of REFINE for measure $|x^{-1}(B)|$, and
- the outer for-loop of COLLECT for measure $|B|$.

After that we give an $O(|X||A|\log|A|)$ bound for the number of steps taken in the inner for-loop of COLLECT (and consequently for REFINE) for measure $|B|$. This analysis suits also for the measure $|x^{-1}\cap B|$.

Next we tackle the problem of updating the sets $x^{-1}\cap B$. It turns out that the traditional implementation may take in some cases $O(|X|^2|A|)$ steps. If $|X|$ is of the same magnitude as $|A|$ then methods exploiting these sets run in time $O(|A|^3)$, whereas the method based on $|B|$ has complexity $O(|A|^2\log|A|)$. A simple enhancement suffices, however, to take us back to the $O(|X||A|\log|A|)$ bound.

Summing up all of the above, we have that all the different variants (provided we apply our enhancement) run in time $O(|X||A|\log|A|)$, whereas the original implementation of the Hopcroft algorithm (using either $|x^{-1}\cap B|$ or $|x^{-1}(B)|$) may take $O(|A|^3)$ when X is of the same magnitude as A.

Finally, we study some properties of the algorithm based on the measure $|x^{-1}(B)|$ that suggest an ordering on the members of L.

5.1. Cost functions on derivation trees

Given a derivation tree DT_θ and a letter x, we define the *cost* of each subtree $DT_\theta(B)$, $cost(B,x)$ in the natural way as $cost(B,x)=work(B,x)+cost(B',x)+cost(B'',x)$, where $work(B,x)\geqslant 0$ is the work assigned to the pair (B,x). The cost of the whole DT_θ is then $\sum_{x\in X} cost(A,x)$. The *work*-functions we use have some useful properties described in the following definition.

Definition 16. Function $work(B,x)$ is *well-behaving* if

- $work(B,x)>0$ only when $Col(x)(B)=green$, and
- $work(B,x)\geqslant work(B_1,x)+\cdots+work(B_n,x)$ for any partition $B_1,\ldots,B_n$ of B with $Col(x)(B)=green$.

We denote the work done with the green nodes as $wg(B,x)$ in the sequel. Note that we are allowed to write $wg(B,x)$ even if $Col(x)(B)\neq green$: in that case it implies the work that would be done *if* the pair (B,x) were selected from L.

Consider now $work(B,x)$ as the number of steps needed to collect and refine classes with measure $|x^{-1}(B)|$. We clearly execute some $c|x^{-1}(B)|$ $(c>0)$ instructions for each selected candidate pair (B,x), and pairs with other colors do not participate in this work. Furthermore, for any partition $B_1,\ldots,B_n$ of B we have that $\sum_{i=1}^n c|x^{-1}(B_i)|=c|x^{-1}(B)|$. Similarly, the outer loop of COLLECT is executed $|B|$ times when $|B|$ is used as the selection measure, and $\sum_{i=1}^n c|B_i|=c|B|$ $(c>0)$.

5.1.1. Proper colorings

Since the colors assigned to derivation trees depend on the measure used in the selection, not all possible colorings are legal.

Definition 17. Let $m(B,x)$ be the selection measure used in the updates of L. Mapping $Col(x)$ is a *proper coloring* of DT_θ, if the following conditions hold.

(1) $Col(x)(A) = green$ (root).
(2) Leaf nodes are not colored amber.
(3) If $Col(x)(B) \in \{red, green\}$ and $m(B',x) < m(B'',x)$ then $Col(x)(B') \in \{green, amber\}$ and $Col(x)(B'') = red$. If $m(B',x) = m(B'',x)$ an arbitrary choice can be made.
(4) If $Col(x)(B) = amber$ then both $Col(x)(B') \neq red$ and $Col(x)(B'') \neq red$.

It should be clear that all colorings induced by Algorithm 8 are proper.

5.1.2. Colorings with maximal cost

The definition of a well-behaving *work*-function leads directly to the intuition that $cost(A,x)$ has its maximum when x colors as many of the nodes green as possible, i.e. none of the nodes are amber. We show next that this is indeed the case for proper colorings.

Definition 18. The *distance* of colorings $Col(x)$ and $Col(y)$ is the number of nodes in which they disagree, that is $|\{B \in DT_\theta \mid Col(x)(B) \neq Col(y)(B)\}|$.

Suppose we have a *red–green coloring* $Col(x)$ on DT_θ such that $Col(x)(B) \in \{green, red\}$ for all nodes B of DT_θ. If we define a new coloring $Col(x')$ such that the color of one inner node B in DT_θ is changed from green to amber, then B' and B'' must be colored green (or amber with subsequent changes to descendant nodes) in order to make also $Col(x')$ proper. The minimal changes from $Col(x)$ to $Col(x')$ are (supposing B'' is red): $Col(x')(B) = amber$ and $Col(x')(B'') = green$. The resulting $Col(x')$ is proper since the numbers $m(B,x)$ do not change. We denote this transformation with $Col(x') = g2a(B, Col(x))$ in the sequel.

Note that the set $\{g2a(B, Col(x)) \mid B \in DT_\theta, B \notin A/\theta\}$ contains all proper colorings within minimal distance (2) from $Col(x)$. As a consequence, for any coloring $Col(x')$ on DT_θ with k amber nodes, we have a red-green coloring $Col(x)$ and nodes B_i $(1 \leqslant i \leqslant k)$ of DT_θ such that

$$Col(x') = g2a(B_1, g2a(B_2, \ldots, g2a(B_k, Col(x)) \cdots)),$$

i.e. $Col(x')$ can be obtained from $Col(x)$ with k iterations of $g2a$ on appropriate nodes of DT_θ. The next lemma tells us that the red-green colorings maximize $cost(A,x)$.

Lemma 19. *Let DT_θ be a partition tree, $Col(x)$ a proper red–green coloring, g2a the transformation described above, and $Col(x')$ a new coloring given by $k \geqslant 0$ applications of g2a on $Col(x)$. Then $cost(A,x) \geqslant cost(A,x')$.*

Proof. The claim clearly holds for $k = 0$ ($Col(x') = Col(x)$). Let the assumption hold for $k = l$ ($l \geqslant 0$) and consider the case $k = l + 1$. Here we have two subcases:
(1) The region of the amber nodes is contiguous in the sense that there is only one amber node with a nonamber parent.

(2) The coloring consists of a multitude of amber regions.

Consider case (1) and let B be the topmost node of the amber region. Then the colorings $Col(x)$ and $Col(x')$ must agree for all nodes below $gf(B,x')$ by the definition of $g2a$.

In the following derivation we use a shorthand $cb(B,x)$ (cost below) for the number $cost(B,x) - work(B,x)$. Since $work(D,x') = 0$ for all nodes D in the amber region (*work* is well-behaving), we have that

$$
\begin{aligned}
cost(B,x') &= \sum_{D \in gf(B,x')} cost(D,x') && 1\\
&= \sum_{D \in gf(B,x')} [work(D,x') + cb(D,x')] && 2\\
&= \sum_{D \in gf(B,x')} wg(D,x') + \sum_{D \in gf(B,x')} cb(D,x') && 3\\
&= \sum_{D \in gf(B,x')} wg(D,x) + \sum_{D \in gf(B,x')} cb(D,x) && 4\\
&\leqslant wg(B,x) + \sum_{D \in gf(B,x')} cb(D,x) && 5\\
&\leqslant work(B,x) + \sum_{D \in gf(B,x')} cost(D,x) && 6\\
&\leqslant cost(B,x) && 7
\end{aligned}
$$

The reasoning steps above are justified by the following facts:

1–2 Definition of *cb*.

2–3 $Col(x')(D) = green$.

3–4 $Col(x)$ and $Col(x')$ agree under D, and $wg(D,x) = wg(D,x')$ does not depend on coloring.

4–5 *work* (and consequently *wg*) is well-behaving.

5–6 $Col(x)(B) = green$ and $work(D,x) \geqslant 0$.

6–7 $work(E,x) \geqslant 0$ in the nodes E between B and $gf(B,x')$.

Consider then case (2). Each of the separate regions contains at most l amber nodes. Let the topmost nodes of these regions be $B_1, \ldots, B_l$ $(1 \leqslant l \leqslant |B|)$. Then it must hold by the construction of $Col(x')$ that $Col(x)(B_i) = green$ for all $1 \leqslant i \leqslant l$, and subsequently (by IA) that $cost(B_i,x) \geqslant cost(B_i,x')$.

Corollary 20. *Let DT_θ be a partition tree, $x \in X$, $Col(x)$ a proper coloring of DT_θ, and $work(B,x)$ well-behaving. Then $cost(A,x)$ has its maximum when $Col(x)(B) \in \{red, green\}$ for all nodes B in DT_θ.*

5.1.3. Recurrence relations for execution times

We next rewrite our formula for $cost(B,x)$ in a form specialized to red-green colorings and to the case where $m = \mathrm{wg}$, that is, the selection measure relates itself directly to the actual work. This is the case for measure $|x^{-1}(B)|$ and the whole main loop; measures $|x^{-1} \cap B|$ and $|B|$ are related only to the outer for-loop of the COLLECT routine.

We denote by $\mathrm{cg}(n)$ the cost of a subtree rooted at a green node B with $\mathrm{wg}(B,x)=n$, and similarly with $\mathrm{cr}(n)$ the cost of a subtree rooted at a red node. If B is refined to B' and B'' where B' has the smaller measure $(\mathrm{wg}(B',x)=q\leqslant \mathrm{wg}(B,x)/2)$, then the following hold $(\mathrm{cg}(0)=\mathrm{cr}(0)=0)$:

$$\mathrm{cg}(n)=n+\mathrm{cg}(q)+\mathrm{cr}(n-q),$$
$$\mathrm{cr}(n)=\mathrm{cg}(q)+\mathrm{cr}(n-q)=\mathrm{cg}(n)-n.$$

Combining the above we have that

$$\mathrm{cg}(n)=n+\mathrm{cg}(q)+\mathrm{cg}(n-q)-(n-q)=q+\mathrm{cg}(q)+\mathrm{cg}(n-q).$$

Note that the equation for $\mathrm{cg}(n)$ above is very similar to $f(n)=n+f(q)+f(n-q)$ describing the complexity of *quicksort* [10, 21]. However, placing $q=1$ at each step we get $\mathrm{cg}(n)=n$, whereas in the quicksort case we have $f(n)=n^2$. If we place $q=n/2$, we get in both cases $n\log n$.

We next show that $\mathrm{O}(n\log n)$ is also the upper bound for $\mathrm{cg}(n)$ for any $1\leqslant q\leqslant n/2$. If $\mathrm{wg}(A,x)$ is initially bounded by $k|A|$ $(k>0)$, as it is with all of our measures, then consequently $cost(A,x)$ is bounded by $\mathrm{O}(|A|\log|A|)$.

Lemma 21. *Let* $\mathrm{cg}(n)=q+\mathrm{cg}(q)+\mathrm{cg}(n-q)$, *where* $1\leqslant q\leqslant\lfloor n/2\rfloor$. *Then* cg *is bounded by* $\mathrm{O}(n\log n)$.

Proof. The proof follows immediately from the worst-case execution time analysis of quicksort (see [7], for example). Guessing $\mathrm{cg}(n)\leqslant kn\log n$ $(k>0)$ and substituting the recursive right-hand side of the equation with this guess, we get

$$\begin{aligned}\mathrm{cg}(n) &\leqslant \max_{1\leqslant q\leqslant n/2}[q+\mathrm{cg}(q)+\mathrm{cg}(n-q)]\\ &= \max_{1\leqslant q\leqslant n/2}[q+kq\log q+k(n-q)\log(n-q)].\end{aligned}$$

To find the maximum of the function $f(q)=q+k(q\log q+(n-q)\log(n-q))$, we differentiate it with respect to q and get

$$f'(q)=k'+k(\log q-\log(n-q))\ (k'>0),$$
$$f''(q)=(k/\ln 2)(1/q+1/(n-q)).$$

Since $f''(q)>0$ for all $1\leqslant q\leqslant n/2$, the maximum of f is either $f(1)$ or $f(n/2)$. Our previous remarks tell us that $f(1)=n$ and $f(n/2)=n\log n$.

Corollary 22. *Algorithm* 8 *executes* $\mathrm{O}(|X||A|\log|A|)$ *steps in* Collect *and* Refine (*excluding the updates of sets* $x^{-1}\cap B$) *when* $m(B,x)=|x^{-1}(B)|$.

Corollary 23. *Algorithm* 8 *executes* $\mathrm{O}(|X||A|\log|A|)$ *steps in the outer loop of* Collect *when* $m(B,x)=|B|$ *or* $m(B,x)=|x^{-1}\cap B|$ $(<|B|)$.

Note that the average case cannot be handled by simply assuming that all q in the range $1..\lfloor n/2 \rfloor$ are equally likely, and taking the average of the execution times (as is done in the analysis of quicksort), since we should also consider all different proper colorings and their distribution. This distribution is not necessarily uniform, since $Col(x)$ may depend on some $Col(y)$. It is also unclear, whether all possible proper colorings (for a fixed X and A) are induced by some DFA.

5.2. Measures $|B|$ and $|x^{-1} \cap B|$ and the remaining parts of the algorithm

For measures $|B|$ and $|x^{-1} \cap B|$, the $O(|x^{-1}(B)|)$ work consumed in REFINE and in the innermost loop of COLLECT is not directly proportional to $m(B,x)$. Hence, the analysis of the previous section does not apply to them. We next show that the total number of these executions is also of order $O(|X||A| \log |A|)$. This is done by setting an upper bound for the *accesses* of one particular transition $x(a) = b$ when doing the refinements. Lemma 24 below considers only the variant $m(B,x) = |B|$, since the case $m(B,x) = |x^{-1} \cap B|$ can be drawn from it by simply replacing $|B|$ with $|x^{-1} \cap B|$ in appropriate places.

Lemma 24. *Let $a \in A$ and $x \in X$. Then the maximum number of times a pair (B,x) with $a \in B$ is used as a parameter to* COLLECT *is bounded by* $\log |A|$.

Proof. Let us reinterpret the claim in the terms of derivation trees and their colorings. Let B in DT_θ, $a \in B$, and $Col(x)(B) \in \{green, red\}$. We claim that if state a occurs in some node D of a proper subtree of $DT_\theta(B)$, and $Col(x)(D) = green$, then $|D| \leqslant |B|/2$. In particular, the claim holds for the *nearest* such D. The original claim follows from the above, since A can be halved at most $\log |A|$ times.

We establish the claim by induction on the height h of $DT_\theta(B)$. The case $h = 1$ (B is a leaf) is clear, since $0 \leqslant \log |B|$. Assume that the claim holds for all trees of height k or less, $h = k + 1$, and (without loss of generality) that $|B'| \leqslant |B''|$. Then $Col(x)(B') \in \{green, amber\}$ and $Col(x)(B'') = red$. We have the following cases:

$a \in B'$: Here we know that $|B'| \leqslant |B|/2$. If $Col(x)(B') = green$, we are done. In the amber case, if a ever occurs in a green node D under B', it must also hold that $|D| \leqslant |B'|$.

$a \in B''$: We know that $|B''| < |B|$, and (by IA) that all the (possible) green nodes D under B'' that contain a have the property $|D| \leqslant |B''|/2$.

Corollary 25. *The total number of iterations done at the innermost for-loop of* COLLECT *is bounded by* $O(|X||A| \log |A|)$ *for both* $m(B,x) = |B|$ *and* $m(B,x) = |x^{-1} \cap B|$.

Proof. We make $|x^{-1}(a)|$ iterations whenever transition $x(a)$ is considered. Summing all these up we have that

$$\sum_{a \in A, x \in X} |x^{-1}(a)| \log |A| = \sum_{x \in X} |A| \log |A| = |A||X| \log |A|.$$

5.3. Manipulation of sets $x^{-1} \cap B$

Here we tackle the problem of updating the numbers $|x^{-1}(B)|$ and sets $x^{-1} \cap B$.

5.3.1. Built and remaining classes

Consider the work performed when some class B is actually refined in routine REFINE with respect to some pair (C,x): $B_{C,x}$ is built out of the states that are moved from B whereas $B^{C,x}$ will simply consist of those states that remain in B. Similarly, in the updates of $y^{-1} \cap B$, states of $B_{C,x}$ are moved to build up $y^{-1} \cap B_{C,x}$ for each $y \in X$, but no work is done for the states in the remaining part $B^{C,x}$. Thus, for each refinement of a class, we create two kinds of new classes: *built* and *remaining* classes, of which only the first ones consume execution time in the updates of the sets $x^{-1} \cap B$. It follows from the definition that each node of the partition tree has a built and a remaining descendant.

We next consider how much the building work takes in total. If we want to preserve the time complexity within the $O(|X||A| \log |A|)$ bound, we should establish the following claim.

Claim 26. *Let* $Built = \{B_1, \ldots, B_m\}$ $(0 \leqslant m \leqslant |A|)$ *be the set of all built classes created in the minimization algorithm. Then*

$$|Built| = \sum_{i=1}^{m} |B_i| \leqslant k|A| \log |A| \quad (k > 0).$$

Our claim actually holds for the case causing maximal work for the other parts of the algorithm, namely balanced derivation trees. This is because exactly half of the classes at each level of the tree are built, and the sum of the sizes of these classes is $|A|/2$ at each level. Since a balanced tree has $\log |A|$ levels, $|Built| = (|A|/2) \log |A|$.

5.3.2. A counter-example to the $O(|X||A| \log |A|)$ bound

We here construct a DFA such that $|Built|$ is quadratic in $|A|$. The trick is to fool the selection routine to select for $1 \leqslant i \leqslant |X|$ a pair that causes a class of size $|A|/2 - i$ to be built. With X large enough (e.g. $|X| \approx |A|/2$), the total update time will be quadratic in $|A|$ thus falsifying Claim 26.

Let $\mathfrak{A} = (A, X, \delta, a_0, A')$, where $A = \{a_1, \ldots, a_{2n}\}$, $X = \{x_1, \ldots, x_n\}$, $A' = \{a_1, \ldots, a_n\}$, and for all $x_i \in X$

$$x_i(a_j) = a_{n+j} \quad \text{for } 1 \leqslant j \leqslant n,$$
$$x_i(a_{n+i}) = a_{n+i},$$
$$x_i(a_j) = a_i \quad \text{for } n < j \neq n+i \leqslant 2n.$$

Let us trace the execution of Algorithm 8 with $m(B,x) = |x^{-1} \cap B|$. Initially $A/\theta = \{B_1, B_2\} = \{\{a_1, \ldots, a_n\}, \{a_{n+1}, \ldots, a_{2n}\}\}$. It follows from the construction that $|x^{-1} \cap B_1| = 1$ and $|x^{-1} \cap B_2| = |B_2| = n$ for all $x \in X$. Also, for each x_i, all but one state from B_2 map to a single state a_i in B_1.

At the initialization step, only pairs (B_1,x) are inserted into L. After refining θ with some (B_1,x_i), $n-1$ states are used to build class B_3 from B_2 leaving only state a_{n+i} in the original class. Assuming (without loss of generality) $i=1$ we have $A/\theta=\{B_1,B_2,B_3\}$ with $B_2=\{a_{n+1}\}$ and $B_3=\{a_{n+2}\ldots,a_{2n}\}$. When L is updated, only pairs (B_2,x) are added since $|x^{-1}\cap B_2|=1$ and $|x^{-1}\cap B_3|=n-1$ for all $x\in X$.

List L contains now two kinds of candidates: (B_1,x_i) $(i\neq 1)$ and (B_2,x_i). If we pick a pair (B_1,x_i) at the next iteration, we must again build a class of size $n-2$ from B_3. Since we do not want to exploit any particular order of choosing pairs from L, let us see what happens if we pick a pair (B_2,x_i) instead.

Refining θ with (B_2,x_i) will move state a_1 from B_1 because $x(a_1)=a_{n+1}$ for all $x\in X$, and the result is $A/\theta=\{B_1,B_2,B_3,B_4\}$ with $B_1=\{a_2,\ldots,a_n\}$ and $B_4=\{a_1\}$. When L is updated, we note that (B_1,x_i) are already in L (except for x_1) and they are thus replaced with both of their descendants. For x_1, we do not add anything, since $x_1^{-1}(B_1)=\emptyset$.

Consider now the contents of L. Candidates (B_2,x) do not refine θ, since both $x(B_1)\cap B_2=\emptyset$ and $x(B_3)\cap B_2=\emptyset$. Similarly, both $x(B_1)\cap B_4=\emptyset$ and $x(B_3)\cap B_4=\emptyset$ for all x and candidates (B_4,x) do not cause any refinement, either. Thus, we are bound to eventually use a member (B_1,x) $(x\neq x_1)$ of L, which forces us to build a class of size $n-2$.

From the above, we have that the each use of candidate (B_1,x_i) (which we eventually will always end up to) builds a class of size $(n-i)$ (supposing we try the letters in the order of their indices). This arithmetic series sums up to $n(n-1)/2=|A|^2/8-|A|/4$.

The DFA used above is enough to fool also the metric $|x^{-1}(B)|$. Here we have at the beginning $|x^{-1}(B_1)|=n-1$ and $|x^{-1}(B_2)|=n+1$ for all $x\in X$. Picking (B_1,x_1) causes us again to build a class of size $n-1$, and the resulting partition is $\{B_1,B_2,B_3\}=\{\{a_1,\ldots,a_n\},\{a_{n+1}\},\{a_{n+2}\ldots,a_{2n}\}\}$. In the update of L we have always that $|x^{-1}(B_2)|=2$ and $|x^{-1}(B_3)|=n-1$. Now pairs (B_2,x) will only move a_1 out of B_1, and the resulting class (B_4) will not refine θ any further. Thus, sooner or later, we must hit a candidate (B_1,x) now building a class of size $n-2$.

5.3.3. A simple yet powerful enhancement

The counterexample given in the previous section exploited the fact that our selection measures could be lurked to build sets $x^{-1}\cap B$ of unacceptable sizes. When some class B is first refined, we have no choice in selecting whether we build $B_{C,x}$ or $B^{C,x}$: only members of the former are collected into the `move`-list from the elements of $x^{-1}(C)$. However, after $B_{C,x}$ and $B^{C,x}$ are once created, we are no longer forced to build sets $y^{-1}\cap B_{C,x}$, too. In fact, we have a total freedom of choice, and naturally, we choose to build the smaller one.

Algorithm 9 (*Splitting the sets* $x^{-1}\cap B$)

```
SPLITINCOME(Old, Large, Small)
1 for each x∈X do
2     hole ← in_class[x][Old].first
```

```
3     in_sum ← 0; s ← 0
4     for each a in Small do
5        apos ← in_c_idx_of[x][a]
6        if apos ⩾ 0 then
7           in_sum ← in_sum + inv_head[x][a].size
8           b ← in_c_elts[x][hole]
9           c_idx_of[x][a] ← hole
10          c_idx_of[x][b] ← apos
11          in_c_elts[x][hole] ← a
12          in_c_elts[x][apos] ← b
13          hole ← hole + 1; s ← s + 1
14    in_class[x][Large].size ← in_class[x][Old].size − s
15    in_class[x][Small].size ← s
16    in_class[x][Small].first ← in_class[x][Old].first
17    in_class[x][Large].first ← hole
18    in_class[x][Large].in_size ←
        in_class[x][Old].in_size − in_sum
19    in_class[x][Small].in_size ← in_sum
```

Algorithm 9 shows the implementation of this enhanced update method. It is mostly the same as Algorithm 7, but the list `move` is replaced with the segment header of the smaller class. The routine is invoked by augmenting REFINE to contain lines

```
if cls_head[Bnew].size <= cls_head[B].size then
   SPLITINCOME(B, B, Bnew)
else SPLITINCOME(B, Bnew, B)
```

after line 3 (building `Bnew`). Notice that the order in which the information is updated in the end of REFINEINCOME is important, since either `Small` or `Large` is the same as `Old`.

Consider then $|Built|$ from the viewpoint of sets $x^{-1} \cap B$. At each inner node $DT_\theta(B)$ with $|B| = n$, we have built a set of size $q \leqslant n/2$, and the rest of the building work is performed in the child nodes. Thus, we have again our familiar recurrence relation $f(n) = kq + f(q) + f(n-q)$ with the known upper bound $kn \log n$.

Corollary 27. *Let* $Built = \{B_1, \ldots, B_m\}$ *be the set of classes iterated over when building sets* $x^{-1} \cap B$ *with Algorithm* 8 *and* 9. *Then* $|Built| = \sum_{i=1}^{m} |B_i| \leqslant |A| \log |A|$.

5.4. A note on uniform partitioning

Number $|x^{-1}(C)|$ is the most accurate selection measure, since it is directly related to the number of steps taken in the main loop of the minimization algorithm. This accuracy enables us to study some computational properties in more detail.

Recall that in the solving of the upper bound for the recurrence relation for $\mathrm{cg}(n)$ we had complexity $\mathrm{O}(n)$ for the case $q=1$ and $\mathrm{O}(n\log n)$ for $q=n/2$. This implies that $cost(B,x)$ (where $Col(x)(B)=green$) has the smaller value the less work is assigned to the green descendant of B. We next show that this is indeed the case, i.e. $cost(B,x)$ grows monotonically with the fraction assigned to the green descendant.

Let us assume that each node B with $|x^{-1}(B)|=n$ of DT_θ is divided uniformly to partitions B' and B'' such that $|x^{-1}(B')|=\alpha n$ and $|x^{-1}(B'')|=(1-\alpha)n$, where $0<\alpha\leqslant 1/2$ is a constant. The function cg can now be rewritten as $\mathrm{cg}(n)=k\alpha n+\mathrm{cg}(\alpha n)+\mathrm{cg}((1-\alpha)n)$. We next show that the closed form for cg is

$$\mathrm{cg}(n)=k'n-\frac{k\alpha n\log n}{\alpha\log\alpha+(1-\alpha)\log(1-\alpha)},$$

where $k'>0$ is some constant (note that the latter term is actually the more expensive one):

$$\begin{aligned}
\mathrm{cg}(\alpha n)+\mathrm{cg}((1-\alpha)n) &= k'\alpha n-\frac{k\alpha(\alpha n)\log(\alpha n)}{\alpha\log\alpha+(1-\alpha)\log(1-\alpha)}\\
&= k'(1-\alpha)n-\frac{k\alpha(1-\alpha)n\log(1-\alpha)n}{\alpha\log\alpha+(1-\alpha)\log(1-\alpha)}\\
&= k'n-\left(\frac{c\alpha(\alpha n)(\log\alpha+\log n)}{\alpha\log\alpha+(1-\alpha)\log(1-\alpha)}\right.\\
&\quad\left.+\frac{k\alpha(1-\alpha)n(\log(1-\alpha)+\log n)}{\alpha\log\alpha+(1-\alpha)\log(1-\alpha)}\right)\\
&= k'n-k\alpha n-\frac{c\alpha n\log n)}{\alpha\log\alpha+(1-\alpha)\log(1-\alpha)}\\
&= \mathrm{cg}(n)-k\alpha n.
\end{aligned}$$

If we define cg as a function of α (and keep n constant), we have that

$$\begin{aligned}
\mathrm{cg}'(\alpha) &= -\frac{k\,n\log n}{\alpha\log\alpha+(1-\alpha)\log(1-\alpha)}+\frac{k\alpha n\log n(\log\alpha-\log(1-\alpha))}{(\alpha\log\alpha+(1-\alpha)\log(1-\alpha))^2}\\
&= \cdots\\
&= -\frac{kn\log n}{(\alpha\log\alpha+(1-\alpha)\log(1-\alpha))^2}\log(1-\alpha).
\end{aligned}$$

Thus, $\mathrm{cg}'(\alpha)>0$ for all $0<\alpha\leqslant 1/2$.

The monotonicity of $\mathrm{cg}(\alpha)$ can be exploited as a heuristics to select items from L: we try to minimize α (and thus $\mathrm{cg}(\alpha)$) by always taking the candidate (B,x) with minimal $|x^{-1}(B)|$. This can be achieved by implementing L as a heap. Assuming that X is not of greater magnitude than A the management of the heap does not increase the asymptotic time complexity of the algorithm. This is because L may contain at most $|X||A|$ items, and insertions to the heap are bounded by $\mathrm{O}(\log|X|+\log|A|)$.

6. Conclusion and future work

We have presented a tutorial reconstruction of Hopcroft's minimization algorithm. As a result of this, the different variations of the algorithm could be placed in a common framework. With the invention of the concept 'derivation tree', we were able to give a firm and understandable correctness proof for the algorithm. Derivation trees enabled us also to make the computational analysis of the algorithm in a constructive way. In the implementation of the algorithm, we presented a new approach to refine the partition structures, and proposed a simple enhancement that is required to keep the more informed variants within the $O(|X||A| \log |A|)$ bound.

All variants are of the same time complexity, but it is important to know, if some of them is generally more efficient in practice than others. Although the method based on measure $|B|$ makes less informed choices in the updates of L, it does not need to update sets $x^{-1} \cap B$. Initial benchmarks on these variants suggest that the savings gained from the wise updates of L are not enough to cover the penalty caused by the set updates even with binary alphabets. We are going to test, what is the effect of implementing L as a heap and to run our algorithms against other known implementations [6, 18, 22, 23, 14, 2].

In many real-life applications, the transition table of the DFA need not be defined totally, since most of the transitions end up to a single 'garbage state'. One point of further work is thus to study, whether the minimization algorithm can be modified to gracefully adapt to partially defined transition functions. Here the goal is to find an algorithm with running time $O(|\delta| \log |A|)$, where $|\delta| \leqslant |X||A|$ is the number of *defined* transitions. Although the more general approach presented in [16] is already capable of achieving this bound, we expect our approach to be more efficient in practice.

Some of our techniques (in particular colored derivation trees and Algorithm 7) could be usable in the analysis and implementations of other kinds of partition refinement problems as graph refinement [5] and relational coarsest partition problem [16]. Finally, we are working to extend the minimization algorithm to handle tree automata [8].

Acknowledgements

The author would like to thank Olli Nevalainen, Timo Raita and Magnus Steinby for their support during the preparation of the article. The detailed comments and suggestions of the referee greatly helped to improve the paper.

References

[1] A.V. Aho, J. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1974.
[2] N. Blum, An $O(n \log n)$ implementation of the standard method for minimizing n-state finite automata, Inform. Process. Lett. 57(2) (1996) 65–69.
[3] W. Brauer, Automatentheorie, B. G. Teubner, Stuttgart, 1984.

[4] J.R. Büchi, in: D. Siefkes (Ed.), Finite Automata, Their Algebras and Grammars, Springer, New York, 1989.
[5] A. Cardon, M. Crochemore, Partitioning a graph in $O(|A| \log |V|)$, Theoret. Comput. Sci. 19 (1982) 85–98.
[6] J.-M. Champarnaud, G. Hansel, Automate, a computing package for automata and finite semigroups, J. Symbolic Comput. 12 (1991) 197–220.
[7] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[8] F. Gécseg, M. Steinby, Tree Automata, Akadémiai Kiadó, Budapest, 1984.
[9] D. Gries, Describing an algorithm by Hopcroft, Acta Inform. 2 (1973) 97–109.
[10] C.A.R. Hoare, Quicksort, Comput. J. 5(1) (1962) 10–15.
[11] J. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, Technical Report CS-190, Stanford University, 1970.
[12] J. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Z. Kohavi, A. Paz (Eds.), Proc. Internat. Symp. on the Theory of Machines and Computations, Haifa, Israel, Academic Press, New York, 1971, pp. 189–196.
[13] J. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, Reading, MA, 1979.
[14] O. Matz, A. Miller, A. Potthoff, W. Thomas, E. Valkema, Report on the program AMoRE, Technical Report 9507, Inst. f. Informatik u. Prakt. Math., CAU Kiel, 1995.
[15] B. Mikolajczak (Ed.), Algebraic and Structural Automata Theory, Annals of Discrete Mathematics, vol. 44, North-Holland, Amsterdam, 1991.
[16] R. Paige, R.E. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16(6) (1987) 973–989.
[17] R. Paige, R.E. Tarjan, R. Bonic, A linear time solution for the single function coarsest partition problem, Theoret. Comput. Sci. 40(1) (1984) 67–84.
[18] D. Raymond, D. Wood, Grail: A C++ library for automata and expressions, J. Symbolic Comput. 17(4) (1994) 341–350.
[19] D. Revuz, Minimization of acylic deterministic automata in linear time, Theoret. Comput. Sci. 92 (1990) 181–189.
[20] A. Salomaa, Theory of Automata, Pergamon Press, Oxford, 1969.
[21] R. Sedgewick, Implementing quicksort programs, Commun. ACM 21(10) (1978) 847–857.
[22] K. Sutner, Implementing finite state machines, in: N. Dean, G. Shannon (Eds.), Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 15, American Mathematical Society, Providence, RI, 1994, pp. 347–364.
[23] B. Watson, The design and implementation of the FIRE engine, Technical Report Computer Science Note 94/22, Eindhoven University of Technology, Faculty of Mathematics and Computing Science, 1994.
[24] B. Watson, A taxonomy of finite automata minimization algorithms, Technical Report Computer Science Note 93/44, Eindhoven University of Technology, Faculty of Mathematics and Computing Science, 1994.