A taxonomy of finite automata minimization algorithms*

Bruce W. Watson
Faculty of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
e-mail: watson@win.tue.nl
Tel: +31 40 474319

May 18, 1994

Abstract

This paper presents a taxonomy of finite automata minimization algorithms. Brzozowski's elegant minimization algorithm differs from all other known minimization algorithms, and is derived separately. All of the remaining algorithms depend upon computing an equivalence relation on states. We define the equivalence relation, the partition that it induces, and its complement. Additionally, some useful properties are derived. It is shown that the equivalence relation is the greatest fixed point of an equation, providing a useful characterization of the required computation. We derive an upperbound on the number of approximation steps required to compute the fixed point. Algorithms computing the equivalence relation (or the partition, or its complement) are derived systematically in the same framework. The algorithms include Hopcroft's, several algorithms from text-books (including Hopcroft and Ullman's [HU79], Wood's [Wood87], and Aho, Sethi, and Ullman's [ASU86]), and several new algorithms or variants of existing algorithms.

*Reprinted with corrections.

Contents

1 Introduction

The minimization of deterministic finite automata is a problem that has been studied since the late 1950's. Simply stated, the problem is to find the unique (up to isomorphism) minimal deterministic finite automaton that accepts the same language as a given deterministic finite automaton. Algorithms solving this problem are used in applications ranging from compiler construction to hardware circuit minimization. With such a variety of applications, the number of differing presentations also grew: most text-books present their own variation, while the algorithm with the best running time (Hopcroft's) remains obscure and difficult to understand.

This report presents a taxonomy of finite automata minimization algorithms. The need for a taxonomy is illustrated by the following:

- Most text-book authors claim that their minimization algorithm is directly derived from those presented by Huffman [Huff54] and Moore [Moor56]. Unfortunately, most text-books present vastly differing algorithms (for example, compare [AU92], [ASU86], [HU79], and [Wood87]), and only the algorithms presented by Aho and Ullman and by Wood are directly derived from those originally presented in [Huff54, Moor56].
- While most of the algorithms rely on computing an equivalence relation on states, many of the explanations accompanying the algorithm presentations do not explicitly mention whether the algorithm computed the equivalence relation, the partition (of states) that it induces, or its complement.
- Comparison of the algorithms is further hindered by the vastly differing styles of presentation — sometimes as imperative programs, or as functional programs, but frequently only as a descriptive paragraph.

A related taxonomy of finite automata construction algorithms appears in [Wats93].

All except one of the algorithms rely on determining the set of automaton states which are equivalent[1]. The algorithm that does not make use of equivalent states is discussed in Section 2. In Section 3 the definition and some properties of equivalence of states is given. Algorithms that compute equivalent states are presented in Section 4. The main results of the taxonomy are summarized in the conclusions — Section 5. Appendices A and B give the basic definitions required for reading this paper. The definitions related to finite automata are taken from [Wats93]. The minimization algorithm relationships are shown in a "family tree" in Figure 1.

The principal computation in most minimization algorithms is the determination of equivalent (or inequivalent) states — thus yielding an equivalence relation on states. In this paper, we consider the following minimization algorithms:

- Brzozowski's (possibly nondeterministic) finite automaton minimization algorithm as presented in [Brzo62]. This elegant algorithm (Section 2) was originally invented by Brzozowski, and has since been re-invented without credit to Brzozowski. Given a (possibly nondeterministic) finite automaton without ϵ-transitions, this algorithm produces the minimal deterministic finite automaton accepting the same language.
- Layerwise computation of equivalence as presented in [Wood87, Moor56, Brau88, Urba89]. This algorithm (Algorithm 4.2) is a straightforward implementation suggested by the approximation sequence arising from the fixed-point definition of equivalence of states.
- Unordered computation of equivalence. This algorithm (Algorithm 4.3, not appearing in the literature) computes the equivalence relation; pairs of states (for consideration of equivalence) are chosen in an arbitrary order.
- Unordered computation of equivalence classes as presented in [ASU86]. This algorithm (Algorithm 4.4) is a modification of the above algorithm computing equivalence of states.

[1] Equivalence of states is defined later.

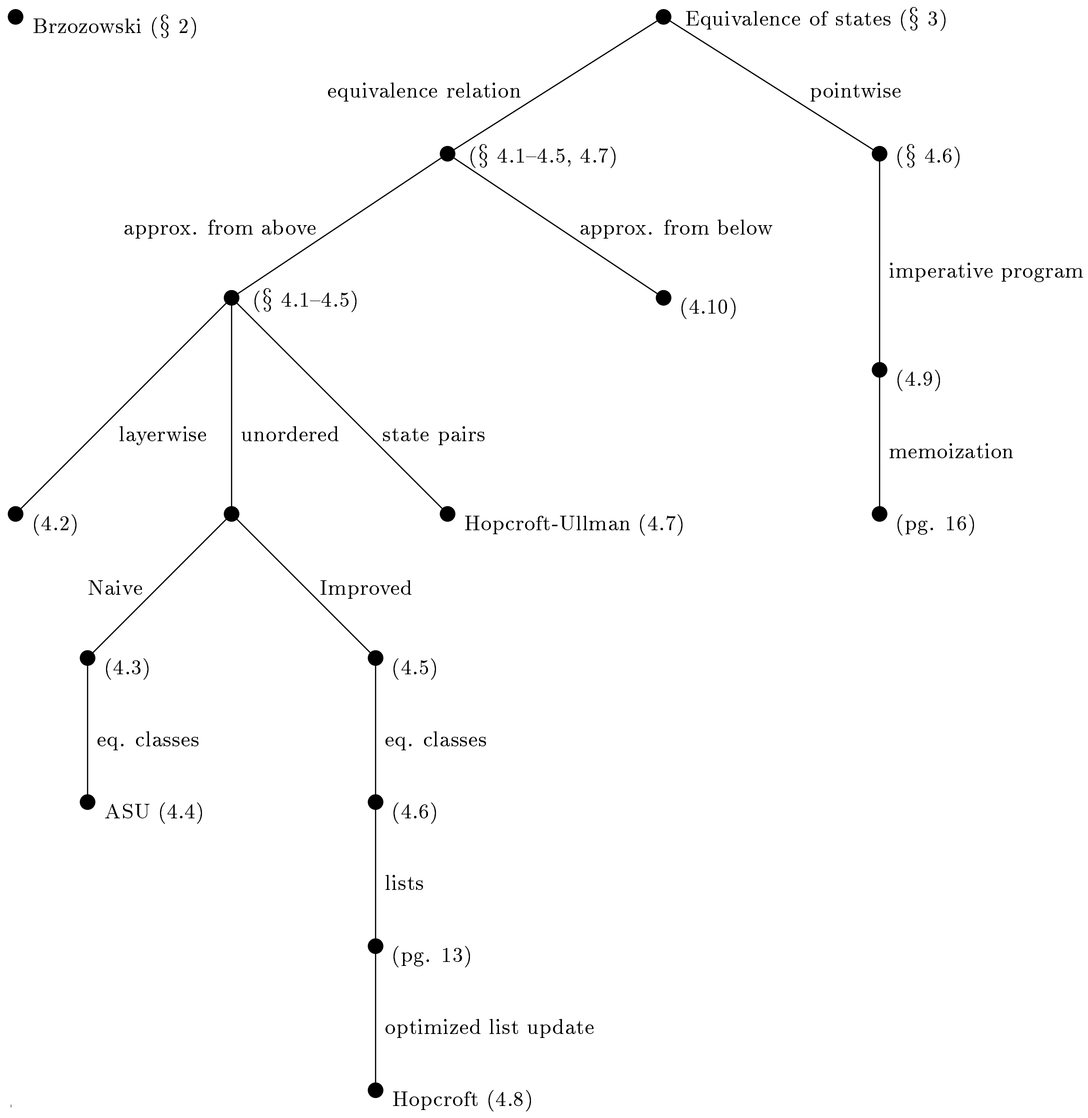

Figure 1: The family trees of finite automata minimization algorithms. Brzozowski's minimization algorithm is unrelated to the others, and appears as a separate (single vertex) tree. Each algorithm presented in this paper appears as a vertex in this tree. For each algorithm that appears explicitly in this paper, the construction number appears in parentheses (indicating where it appears in this paper). For algorithms that do not appear explicitly, a reference to the section or page number is given. Edges denote a refinement of the solution (and therefore explicit relationships between algorithms). They are labeled with the name of the refinement.

- Improved unordered computation of equivalence. This algorithm (Algorithm 4.5, not appearing in the literature) also computes the equivalence relation in an arbitrary order. The algorithm is a minor improvement over the other unordered algorithm.
- Improved unordered computation of equivalence classes. This algorithm (Algorithm 4.6, not appearing in the literature) is a modification of the above algorithm to compute the equivalence classes of states. This algorithm is used in the derivation of Hopcroft's minimization algorithm.
- Hopcroft and Ullman's algorithm as presented in [HU79]. This algorithm (Algorithm 4.7) computes the inequivalence (distinguishability) relation. Although it is based upon the algorithms of Huffman and Moore [Huff54, Moor56], this algorithm uses some interesting encoding techniques.
- Hopcroft's algorithm as presented in [Hopc71, Grie73]. This algorithm (Algorithm 4.8) is the best known algorithm (in terms of running time analysis) for minimization. As the original presentation by Hopcroft is difficult to understand, the presentation in this paper is based upon the one given by Gries.
- Pointwise computation of equivalence. This algorithm (Algorithm 4.9, not appearing in the literature) computes the equivalence of a given pair of states. It draws upon some non-automata related techniques, such as: structural equivalence of types and memoization of functional programs.
- Computation of equivalence from below (with respect to refinement). This algorithm (Algorithm 4.10, not appearing in the literature) computes the equivalence relation from below. Unlike any of the other known algorithms, the intermediate result of this algorithm can be used to construct a smaller (although not minimal) deterministic finite automaton.

2 An algorithm due to Brzozowski

Most minimization algorithms are applied to a *DFA*. In the case of a nondeterministic *FA*, the subset construction is applied first, followed by the minimization algorithm. In this section, we consider the possibility of applying the subset construction (with useless state removal) after an (as yet unknown) algorithm to yield a minimal *DFA*. We now construct such an algorithm. (The algorithm described in this section can also be used to construct the minimal *Complete DFA*, by replacing function *subsetopt* with *subset*.)

Let $M_0 = (Q_0, V, T_0, \emptyset, S_0, F_0)$ be the ϵ*-free FA* to be minimized and $M_2 = (Q_2, V, T_2, \emptyset, S_2, F_2)$ be the minimized *DFA* such that $\mathcal{L}_{FA}(M_0) = \mathcal{L}_{FA}(M_2)$ (and of course $Min(M_2)$ — see Definition B.19). (For the remainder of this section we make use of *Minimal* (Property B.21) as opposed to *Min*.) Since we apply the subset construction last, we have some intermediate finite automaton $M_1 = (Q_1, V, T_1, \emptyset, S_1, F_1)$ such that $M_2 = useful_s \circ subsetopt(M_1)$. We require that M_1 is somehow obtained from M_0, and that $\mathcal{L}_{FA}(M_2) = \mathcal{L}_{FA}(M_1) = \mathcal{L}_{FA}(M_0)$.

From the definition of $Minimal(M_2)$ (Property B.21), we require

$$(\forall\ p, q : p \in Q_2 \wedge q \in Q_2 \wedge p \neq q : \overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q)) \wedge Useful(M_2)$$

For all states $q \in Q_2$ we have $q \in \mathcal{P}(Q_1)$ since $M_2 = useful_s \circ subsetopt(M_1)$. Property B.25 of the subset construction gives

$$(\forall\ p : p \in Q_2 : \overrightarrow{\mathcal{L}}(p) = (\cup\ q : q \in Q_1 \wedge q \in p : \overrightarrow{\mathcal{L}}(q)))$$

We need a sufficient condition on M_1 to ensure $Minimal(M_2)$. The following derivation gives such a condition:

$$\begin{array}{ll}
 & Minimal(M_2) \\
\equiv & \quad \{\ \text{Definition of } Minimal \text{ (Property B.21)}\ \} \\
 & (\forall\ p, q : p \in Q_2 \wedge q \in Q_2 \wedge p \neq q : \overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q)) \wedge Useful(M_2) \\
\Leftarrow & \quad \{\ \text{Property B.25; } M_2 = useful_s \circ subsetopt(M_1)\ \} \\
 & (\forall\ p, q : p \in Q_1 \wedge q \in Q_1 \wedge p \neq q : \overrightarrow{\mathcal{L}}(p) \cap \overrightarrow{\mathcal{L}}(q) = \emptyset) \wedge Useful_f(M_1) \\
\equiv & \quad \{\ \text{Definition of } Det' \text{ (Property B.18) and } Useful_s,\ Useful_f \text{ (Remark B.13)}\ \} \\
 & Det'(M_1^R) \wedge Useful_s(M_1^R) \\
\Leftarrow & \quad \{\ Det'(M) \Leftarrow Det(M)\ \} \\
 & Det(M_1^R) \wedge Useful_s(M_1^R)
\end{array}$$

The required condition on M_1 can be established by (writing reversal as a prefix function) $M_1 = R \circ useful_s \circ subsetopt \circ R(M_0)$.

The complete minimization algorithm (for any ϵ*-free* $M_0 \in FA$) is

$$M_2 = useful_s \circ subsetopt \circ R \circ useful_s \circ subsetopt \circ R(M_0)$$

This algorithm was originally given by Brzozowski in [Brzo62]. The origin of this algorithm was obscured when Jan van de Snepscheut presented the algorithm in his Ph.D thesis [vdSn85]. In this thesis, the algorithm is attributed to a private communication from Prof. Peremans of the Eindhoven University of Technology. Peremans had originally found the algorithm in an article by Mirkin [Mirk65]. Although Mirkin does cite a paper by Brzozowski [Brzo64], it is not clear whether Mirkin's work was influenced by Brzozowki's work on minimization. Jan van de Snepscheut's recent book [vdSn93] describes the algorithm, but provides neither a history nor citations (other than his thesis) for this algorithm.

3 Minimization by equivalence of states

In this subsection, we restrict ourselves to considering minimization of *Complete DFA*'s. This is strictly a notational convenience, as the minimization algorithms can be modified to work for non-*Complete DFA*'s. A *Complete* minimized *DFA* will (in general) have one more state (a sink state) than a non-*Complete* minimized *DFA*, unless the language of the *DFA* is V^*. Let $M = (Q, V, T, \emptyset, S, F)$ be a *Complete DFA*; this particular *DFA* will be used throughout this section. We also assume that all of the states of M are start-reachable, that is $Useful_s(M)$. Since M is deterministic and *Complete*, we will also take the transition relation to be total function $T \in Q \times V \longrightarrow Q$ instead of $T \in Q \times V \longrightarrow \mathcal{P}(Q)$.

In order to minimize the *DFA* M, we compute an equivalence relation $E \subseteq Q \times Q$ defined as:

$$(p, q) \in E \equiv (\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q))$$

Since this is an equivalence relation, we are really interested in unordered pairs of states. It is notationally more convenient to use ordered pairs instead of unordered pairs.

The equivalent states are merged according to equivalence relation E with the *merge* transformation.

Transformation 3.1 (Merging states): For any equivalence relation H such that $H \subseteq E$, the function *merge* can be used to reduce the number of states in the *DFA*[2]. Function *merge* is defined as:

$$\begin{array}{lcll} merge((Q, V, T, \emptyset, \{s\}, F), H) & = & \textbf{let} & T' = \{([p]_H, a, [q]_H) : (p, a, q) \in T\} \\ & & \textbf{in} & \\ & & & ([Q]_H, V, T', \emptyset, \{[s]_H\}, [F]_H) \\ & & \textbf{end} & \end{array}$$

The definition of *merge* is independent of the choice of representatives of the equivalence classes. Function *merge* satisfies the property that

$$\mathcal{L}_{FA}(merge(M, H)) = \mathcal{L}_{FA}(M) \wedge |merge(M, H)| \leq |M| \wedge |merge(M, H)| = \sharp H$$

and it preserves *Complete*, *ϵ-free*, *Useful*, *Det*, and *Minimal*; indeed, *merge* is only defined on *ϵ-free* and deterministic *FA*'s. □

In order to compute relation E, we need a property of function $\overrightarrow{\mathcal{L}}$.

Property 3.2 (Function $\overrightarrow{\mathcal{L}}$): Function $\overrightarrow{\mathcal{L}}$ satisfies

$$\overrightarrow{\mathcal{L}}(p) = (\cup\ a : a \in V : \{a\} \cdot \overrightarrow{\mathcal{L}}(T(p, a))) \cup (\textbf{if}\ (p \in F)\ \textbf{then}\ \{\epsilon\}\ \textbf{else}\ \emptyset\ \textbf{fi})$$

□

This allows us to give an alternate (but equivalent) characterization of equivalence of states.

Definition 3.3 (Equivalence of states): Equivalence relation E is the greatest (under refinement) fixed point of the equivalence

$$(p, q) \in E \equiv (p \in F \equiv q \in F) \wedge (\forall\ a : a \in V : (T(p, a), T(q, a)) \in E)$$

□

Remark 3.4: The greatest fixed point has the least number of equivalence classes of any such fixed point. □

Remark 3.5: Any fixed point of the equivalence in Definition 3.3 can be used. In order to minimize the automaton, the greatest fixed point is desired. □

[2]When H is the identity relation on states, function *merge* will not reduce the number of states.

Property 3.6 (Approximating E): We can compute this greatest fixed point with successive approximations. The successive approximations of E are as follows (for $k \geq 0$):

$$(p,q) \in E_{k+1} \equiv (p,q) \in E_k \wedge (\forall\ a : a \in V : (T(p,a), T(q,a)) \in E_k)$$

where E_0 is defined by

$$(p,q) \in E_0 \equiv (p \in F \equiv q \in F)$$

An equivalent definition of E_0 is $E_0 = (Q \setminus F)^2 \cup F^2$. We also have the property that $E_{k+1} \subseteq E_k$ for all $k \geq 0$. □

Remark 3.7: If E_k is an equivalence relation, then so is E_{k+1}. E_0 is an equivalence relation. □

Remark 3.8: An intuitive explanation of E_k is useful. A pair of states p, q are said to be k-equivalent (written $(p,q) \in E_k$) if and only if there is no string $w : |w| \leq k$ such that $w \in \overrightarrow{\mathcal{L}}(p) \not\equiv w \in \overrightarrow{\mathcal{L}}(q)$. As a consequence, p and q are k-equivalent if and only if

- they are both final or both non-final, and
- for all $a \in V$, $T(p,a)$ and $T(q,a)$ are $(k-1)$-equivalent (by the definitions of $\overrightarrow{\mathcal{L}}$ and T^*).

□

Remark 3.9: An important property of E is that it is also the greatest fixed point, under $\subseteq$ (set containment instead of refinement), of the equivalence in Definition 3.3. As the greatest fixed point, E can be computed with a $\subseteq$-descending sequence of relations, starting with $Q \times Q$. Such a sequence need not consist only of equivalence relations. There may be more steps in such an approximating sequence than in the E_k sequence given above. Fortunately, each such step is usually easier to compute than computing E_{k+1} from E_k. Some algorithms that compute these cheaper (but longer) sequences are given in Sections 4.2–4.5 and 4.7. □

All previously known algorithms compute E by successive approximation from above (with respect to $\subseteq$). A new algorithm in Section 4.7 computes E by successive approximation from below. In that section, the practical importance of this is explained.

3.1 Distinguishability

It is also possible to compute E by first computing its complement $D = \neg E$. Relation D (called the distinguishability relation on states) is defined as:

$$(p,q) \in D \equiv (\overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q))$$

Definition 3.10 (Distinguishability of states): D is the least (under $\subseteq$, set containment[3]) fixed point of an equation

$$(p,q) \in D \equiv (p \in F \not\equiv q \in F) \vee (\exists\ a : a \in V : (T(p,a), T(q,a)) \in D)$$

□

Property 3.11 (Approximating D): As with equivalence relation E, relation D can be computed by successive approximations (for $k \geq 0$)

$$(p,q) \in D_{k+1} \equiv (p,q) \in D_k \vee (\exists\ a : a \in V : (T(p,a), T(q,a)) \in D_k)$$

with $D_0 = \neg E_0 = ((Q \setminus F) \times F) \cup (F \times (Q \setminus F))$. For all $k \geq 0$ we have $D_k = \neg E_k$. We also have the property that $D_{k+1} \supseteq D_k$ for $k \geq 0$. □

[3]Here, $\subseteq$ denotes normal set containment; refinement does not apply since D is not necessarily an equivalence relation.

Remark 3.12: As with E_k, an intuitive explanation of D_k is useful. A pair of states p, q are said to be k-distinguished (written $(p,q) \in D_k$) if and only if there is a string $w : |w| \leq k$ such that $w \in \overrightarrow{\mathcal{L}}(p) \not\equiv w \in \overrightarrow{\mathcal{L}}(q)$. As a consequence, p and q are k-distinguished (some authors say k-distinguishable) if and only if

- one is final and the other is non-final, or
- there exists $a \in V$ such that $T(p,a)$ and $T(q,a)$ are $(k-1)$-distinguished.

□

3.2 An upperbound on the number of approximation steps

We can easily place an upperbound on the number of steps in the computation of E.

Let E_j be the greatest fixed point of the equation defining E. We have the sequence of approximations (where I_Q is the identity relation on states):

$$E_0 \supset E_1 \supset \cdots \supset E_j \supseteq I_Q$$

The indices of some of the equivalence relations in the approximation sequence are known: $\sharp I_Q = |Q|$ and $\sharp E_0 \leq 2$. We can deduce that:

$$\sharp E_0 < \sharp E_1 < \cdots < \sharp E_j \leq \sharp I_Q = |Q|$$

In the case that $\sharp E_0 = 0$, we have that E_0 is the greatest fixed point. In the case that $\sharp E_1 = 1$, either all states are final states, or all states are non-final ones; in both cases E_0 is the greatest fixed point. In the case that $\sharp E_0 = 2$, we have $i + 2 \leq \sharp E_i$. Since $j + 2 \leq \sharp E_j \leq \sharp I_Q = |Q|$ we get $j \leq |Q| - 2$. This gives an upperbound of $(|Q| - 2) \,\mathbf{max}\, 0$ steps for the computation (starting at E_0) of the greatest fixed point E_j (using the approximating sequence given in Property 3.6).

A consequence of this upperbound is that $E = E_{(|Q|-2)\,\mathbf{max}\,0}$. As we shall see later, this can lead to some efficiency improvements to algorithms computing E. This result is also noted by Wood [Wood87, Lemma 2.4.1]. This upperbound also holds for computing D and $[Q]_E$ by approximation.

3.3 Characterizing the equivalence classes of E

It is also practical to compute $[Q]_E$: the set of equivalence classes of E. In order to characterize partition $[Q]_E$, we begin our derivation with Definition 3.3, the characterization of E as the largest equivalence relation (under $\subseteq$) such that

$$\begin{aligned}
& (\forall\, p,q : (p,q) \in E : (p \in F \equiv q \in F) \land (\forall\, a : a \in V : (T(p,a), T(q,a)) \in E)) \\
\equiv \quad & \{\text{Definition of membership in } E\text{; move } a \text{ to outer quantification}\} \\
& (\forall\, p,q,a : (p,q) \in E \land a \in V : (p \in F \equiv q \in F) \land [T(p,a)]_E = [T(q,a)]_E) \\
\equiv \quad & \{\text{Introduce equivalence classes } Q_0, Q_1 \text{ explicitly}\} \\
& (\forall\, Q_0, Q_1, a : Q_0 \in [Q]_E \land Q_1 \in [Q]_E \land a \in V : \\
& \quad (\forall\, p,q : p \in Q_0 \land q \in Q_0 : (p \in F \equiv q \in F) \land (T(p,a) \in Q_1 \equiv T(q,a) \in Q_1)))
\end{aligned}$$

Definition 3.13 (Function *Splittable*): In order to make this quantification more concise, we define

$$\mathit{Splittable}(Q_0, Q_1, a) \equiv (\exists\, p,q : p \in Q_0 \land q \in Q_0 : (T(p,a) \in Q_1 \not\equiv T(q,a) \in Q_1)))$$

□

Using *Splittable*, $[Q]_E$ is the largest partition (under $\sqsubseteq$) such that $[Q]_E \sqsubseteq [Q]_{E_0}$ and

$$(\forall\, Q_0, Q_1, a : Q_0 \in [Q]_E \land Q_1 \in [Q]_E \land a \in V : \neg\mathit{Splittable}(Q_0, Q_1, a))$$

This characterization will be used in the computation of $[Q]_E$.

4 Algorithms computing E, D, or $[Q]_E$

In this section, we consider several algorithms that compute D, E, or $[Q]_E$. Some of the algorithms are presented in general terms: computing D and E. Since only one of D or E is needed (and not both), such a general algorithm would be modified for practical use to compute only one of the two.

4.1 Computing D and E by layerwise approximations

The definition of E_{k+1} in terms of E_k (and likewise for D) leads naturally to the following algorithm computing D and E (where variable k is a ghost variable, used only for specifying the invariant)

Algorithm 4.1:

$G, H := D_0, E_0;$
$G_{old}, H_{old}, k := \emptyset, Q \times Q, 0;$
$\{\text{invariant: } G = D_k \wedge H = E_k\}$
do $G \neq G_{old} \longrightarrow$
  $\{G \neq G_{old} \wedge H \neq H_{old}\}$
  $G_{old}, H_{old} := G, H;$
  $G := (\cup\ p, q : (p,q) \in G_{old} \vee (\exists\ a : a \in V : (T(p,a), T(q,a)) \in G_{old}) : \{(p,q)\});$
  $H := (\cup\ p, q : (p,q) \in H_{old} \wedge (\forall\ a : a \in V : (T(p,a), T(q,a)) \in H_{old}) : \{(p,q)\});$
  $\{G = \neg H\}$
  $k := k + 1$
od$\{G = D \wedge H = E\}$

This algorithm is said to compute D and E layerwise, since it computes the sequences D_k and E_k. The update of G and H in the repetition can be made with another repetition as shown in the program now following.

Algorithm 4.2 (Layerwise computation of D and E):

$G, H := D_0, E_0;$
$G_{old}, H_{old}, k := \emptyset, Q \times Q, 0;$
$\{\text{invariant: } G = D_k \wedge H = E_k\}$
do $G \neq G_{old} \longrightarrow$
  $\{G \neq G_{old} \wedge H \neq H_{old}\}$
  $G_{old}, H_{old} := G, H;$
  for $(p,q) : (p,q) \in H_{old}$ **do**
    if $(\exists\ a : a \in V : (T(p,a), T(q,a)) \in G_{old}) \longrightarrow G, H := G \cup \{(p,q)\}, H \setminus \{(p,q)\}$
    $[\!]$ $(\forall\ a : a \in V : (T(p,a), T(q,a)) \in H_{old}) \longrightarrow$ **skip**
    fi
  rof;
  $\{G = \neg H\}$
  $k := k + 1$
od$\{G = D \wedge H = E\}$

The algorithm can be split into two: one computing only D, and the other computing only E. The algorithm computing only E is essentially the algorithm presented by Wood in [Wood87, pg. 132]. According to Wood, it is based on the work of Moore [Moor56]. Its running times is $\mathcal{O}(|Q|^3)$. Brauer uses some encoding techniques to provide an $\mathcal{O}(|Q|^2)$ version of this algorithm in [Brau88], while Urbanek improves upon the space requirements of Brauer's version in [Urba89]. None of these variants is given here. The algorithm computing only D does not appear in the literature.

With a little effort this algorithm can be modified to compute $[Q]_E$.

4.2 Computing D, E, and $[Q]_E$ by unordered approximation

Instead of computing each E_k (computing E layerwise), we can compute E by considering pairs of states in an arbitrary order (as outlined in Remark 3.9). This is done in the following algorithm (which also computes D):

Algorithm 4.3:

$G, H := D_0, E_0;$
$\{\text{invariant: } G = \neg H \wedge G \subseteq D\}$
do $(\exists\ p, q, a : a \in V \wedge (p, q) \in H : (T(p, a), T(q, a)) \in G) \longrightarrow$
  let $p, q : (p, q) \in H \wedge (\exists\ a : a \in V : (T(p, a), T(q, a)) \in G);$
  $\{(p, q) \in D\}$
  $G, H := G \cup \{(p, q)\}, H \setminus \{(p, q)\}$
od$\{G = D \wedge H = E\}$

This algorithm can be split into one computing only D, and one computing only E. At the end of each iteration step, it may be that H is not an equivalence relation (that is, $H \neq H^*$) — see Remark 3.9. A slight modification to this algorithm can be made by adding the following assignment before the **od**:

$$H := (\mathbf{MAX}_{\subseteq}\ J : J \subseteq H \wedge J = J^* : J); G := \neg H$$

Addition of this assignment makes the algorithm compute the refinement sequence E_k (see Remark 3.9). This assignment may improve the running time of the algorithm if a cheap method of computing the quantified **MAX** is used. This algorithm does not appear in the literature.

When we convert the above algorithm to compute $[Q]_E$, the resulting algorithm is the following one, given by Aho, Sethi, and Ullman in [ASU86, Alg. 3.6]:

Algorithm 4.4:

$P := [Q]_{E_0};$
$\{\text{invariant: } [Q]_E \sqsubseteq P \sqsubseteq [Q]_{E_0}\}$
do $(\exists\ Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V : \mathit{Splittable}(Q_0, Q_1, a)) \longrightarrow$
  let $Q_0, Q_1, a : \mathit{Splittable}(Q_0, Q_1, a);$
  $Q_0' := \{p : p \in Q_0 \wedge T(p, a) \in Q_1\};$
  $\{\neg\mathit{Splittable}(Q_0 \setminus Q_0', Q_1, a) \wedge \neg\mathit{Splittable}(Q_0', Q_1, a)\}$
  $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q_0', Q_0'\}$
od
$\{(\forall\ Q_0, Q_1, a : Q_0 \in P \wedge Q_1 \in P \wedge a \in V : \neg\mathit{Splittable}(Q_0, Q_1, a))\}$
$\{P = [Q]_E\}$

This algorithm has running time $\mathcal{O}(|Q|^2)$.

4.3 More efficiently computing D and E by unordered approximation

We present another algorithm that considers pairs of states in an arbitrary order. This algorithm (which also computes D) consists of two nested repetitions:

Algorithm 4.5:

$G, H := D_0, E_0;$
$\{\text{invariant: } G = \neg H \wedge G \subseteq D\}$
do $(\exists\ p, q, a : a \in V \wedge (p, q) \in H : (T(p, a), T(q, a)) \in G) \longrightarrow$
  let $p, a : p \in Q \wedge a \in V \wedge (\exists\ q : (p, q) \in H : (T(p, a), T(q, a)) \in G);$
  for $q : (p, q) \in H \wedge (T(p, a), T(q, a)) \in G$ **do**
    $G, H := G \cup \{(p, q)\}, H \setminus \{(p, q)\}$
  rof
od$\{G = D \wedge H = E\}$

As with Algorithm 4.3, at the end of each outer iteration step, it may be that $H \neq H^*$. This can be solved with an assignment to H as can be done in Algorithm 4.3. This algorithm does not appear in the literature. It can also be modified to compute only D or only E.

Modifying the above algorithm to compute $[Q]_E$ is particularly interesting; the modified algorithm will be used in Section 4.5 to derive an algorithm (by Hopcroft) which is the best known algorithm for *FA* minimization. The algorithm is (where variable P_{old} is used only for the invariant):

Algorithm 4.6:

$P := [Q]_{E_0};$
$\{\text{invariant: } [Q]_E \sqsubseteq P \sqsubseteq [Q]_{E_0}\}$
do $(\exists\ Q_1, a : Q_1 \in P \wedge a \in V : (\exists\ Q_0 : Q_0 \in P : \mathit{Splittable}(Q_0, Q_1, a))) \longrightarrow$
  let $Q_1, a : (\exists\ Q_0 : Q_0 \in P : \mathit{Splittable}(Q_0, Q_1, a));$
  $P_{old} := P;$
  $\{\text{invariant: } [Q]_E \sqsubseteq P \sqsubseteq P_{old}\}$
  for $Q_0 : Q_0 \in P_{old} \wedge \mathit{Splittable}(Q_0, Q_1, a)$ **do**
    $Q_0' := \{p : p \in Q_0 \wedge T(p, a) \in Q_1\};$
    $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q_0', Q_0'\}$
  rof
  $(\forall\ Q_0 : Q_0 \in P : \neg\mathit{Splittable}(Q_0, Q_1, a))\}$
od
$\{(\forall\ Q_1, a : Q_1 \in P \wedge a \in V : (\forall\ Q_0 : Q_0 \in P : \neg\mathit{Splittable}(Q_0, Q_1, a)))\}$
$\{P = [Q]_E\}$

The inner repetition "splits" each eligible equivalence class Q_0 with respect to pair (Q_1, a). (In actuality, some particular Q_0 will not be split by (Q_1, a) if $\neg\mathit{Splittable}(Q_0, Q_1, a)$.)

4.4 An algorithm due to Hopcroft and Ullman

From the definition of D, we see that a pair (p, q) is in D if and only if $p \in F \not\equiv q \in F$ or there is some $a \in V$ such that $(T(p, a), T(q, a)) \in D$. This forms that basis of the algorithm considered in this subsection. With each pair of states (p, q) we associate a set of pairs of states $L(p, q)$ such that

$$(r, s) \in L(p, q) \Rightarrow ((p, q) \in D \Rightarrow (r, s) \in D)$$

For each pair (p, q) (such that $(p, q) \notin D_0$ — p and q are not already known to be distinguished) we do the following:

- If there is an $a \in V$ such that we know that $(T(p, a), T(q, a)) \in D$ then $(p, q) \in D$. We add (p, q) to our approximation of D, along with $L(p, q)$, and for each $(r, s) \in L(p, q)$ add $L(r, s)$, and for each $(t, u) \in L(r, s)$ add $L(t, u)$, etc.

- If there is no $a \in V$ such that $T((p,a), T(q,a)) \in D$ is known to be true, then for all $b \in V$ we put (p,q) in the set $L(T(p,b), T(q,b))$ since $(T(p,b), T(q,b)) \in D \Rightarrow (p,q) \in D$. If later it turns out that for some $b \in V$, $(T(p,b), T(q,b)) \in D$, then we will also put $L(T(p,b), T(q,b))$ (including (p,q)) in D.

In our presentation of the algorithm, the invariants given are not sufficient to prove the correctness of the algorithm, but are used to illustrate the method in which the algorithm works. The algorithm is:

Algorithm 4.7:

---

for $(p,q) : (p,q) \in (Q \times Q)$ **do**
  $L(p,q) := \emptyset$
rof;
$G := D_0$;
$\{$invariant: $G \subseteq D \wedge (\forall\ p,q : (p,q) \notin D_0 : (\forall\ r,s : (r,s) \in L(p,q) : (p,q) \in D \Rightarrow (r,s) \in D))\ \}$
for $(p,q) : (p,q) \notin D_0$ **do**
  if $(\exists\ a : a \in V : (T(p,a), T(q,a)) \in G) \longrightarrow$
    $A, B := \{(p,q)\}, \emptyset$;
    $\{$invariant: $A \subseteq D \wedge B \subseteq G \wedge A \cap B = \emptyset$
      $\wedge\ A \cup B = (\cup\ p,q : (p,q) \in B : L(p,q))\}$
    do $A \neq \emptyset \longrightarrow$
      let $(r,s) : (r,s) \in A$;
      $G := G \cup \{(r,s)\}$;
      $A, B := A \setminus \{(r,s)\}, B \cup \{(r,s)\}$;
      $A := A \cup (L(r,s) \setminus B)$
    od
  $[\!]$ $(\forall\ a : a \in V : (T(p,a), T(q,a)) \notin G) \longrightarrow$
    for $a \in V : T(p,a) \neq T(q,a)$ **do**
      $\{(T(p,a), T(q,a)) \in D \Rightarrow (p,q) \in D\}$
      $L(T(p,a), T(q,a)) := L(T(p,a), T(q,a)) \cup \{(p,q)\}$
    rof
  fi
rof$\{G = D\}$

This algorithm has running time $\mathcal{O}(|Q|^2)$ and is given by Hopcroft and Ullman [HU79, Fig. 3.8]. In [HU79] it is attributed to Huffman [Huff54] and Moore [Moor56]. In their description, Hopcroft and Ullman describe L as mapping each pair of states to a list of pairs of states. The list data-type is not required here, and a set is used instead.

It is possible to modify the above algorithm to compute E. Such an algorithm does not appear in the literature.

4.5 Hopcroft's algorithm to compute $[Q]_E$ efficiently

We now derive an efficient algorithm due to Hopcroft [Hopc71]. This algorithm has also been derived by Gries [Grie73]. This algorithm presently has the best known running time analysis of all *DFA* minimization algorithms.

We begin with Algorithm 4.6. Recall that the inner repetition "splits" each equivalence class Q_0 with respect to pair (Q_1, a). An observation (due to Hopcroft) is that once all equivalence classes have been split with respect to a particular (Q_1, a), no equivalence classes need to be split with respect to the same (Q_1, a) on any subsequent iteration step of the outer repetition [Hopc71, pp. 190–191], [Grie73, Lemma 5]. The observation is simple to prove: the equivalence classes never grown in size, and we need only prove that (for all equivalence classes $\hat{Q}$):

$$\neg \mathit{Splittable}(\hat{Q}, Q_1, a) \Rightarrow (\forall\ Q'_0 : Q'_0 \subseteq \hat{Q} : \neg \mathit{Splittable}(\hat{Q}, Q_1, a))$$

We can use this fact to maintain a set L of such (equivalence class, alphabet symbol) pairs. We will then split the equivalence classes with respect to elements of L. In the original presentations of this algorithm [Hopc71, Grie73], L is a list. As this is not necessary, we retain L's type as a set.

$P := [Q]_{E_0}$;
$L := P \times V$;
$\{$invariant: $[Q]_E \sqsubseteq P \sqsubseteq [Q]_{E_0} \wedge L \subseteq (P \times V)$
$\quad \wedge\, L \supseteq \{(Q_1, a) : (Q_1, a) \in (P \times V) \wedge (\exists\, Q_0 : Q_0 \in P : \mathit{Splittable}(Q_0, Q_1, a))\}$
$\quad \wedge\, (\forall\, Q_0, Q_1, a : Q_0 \in Q \wedge (Q_1, a) \in L : \neg \mathit{Splittable}(Q_0, Q_1, a)) \Rightarrow (P = [Q]_E)\}$
do $L \neq \emptyset \longrightarrow$
  let $Q_1, a : (Q_1, a) \in L$;
  $P_{old} := P$;
  $L := L \setminus \{(Q_1, a)\}$;
  $\{$invariant: $[Q]_E \sqsubseteq P \sqsubseteq P_{old}\}$
  for $Q_0 : Q_0 \in P_{old} \wedge \mathit{Splittable}(Q_0, Q_1, a)$ **do**
    $Q_0' := \{p : p \in Q_0 \wedge T(p, a) \in Q_1\}$;
    $P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q_0', Q_0'\}$;
    for $b : b \in V$ **do**
      if $(Q_0, b) \in L \longrightarrow L := L \setminus \{(Q_0, b)\} \cup \{(Q_0', b), (Q_0 \setminus Q_0', b)\}$
      $[\!]$ $(Q_0, b) \notin L \longrightarrow L := L \cup \{(Q_0', b), (Q_0 \setminus Q_0', b)\}$
      fi
    rof
  rof
  $\{(\forall\, Q_0 : Q_0 \in P : \neg \mathit{Splittable}(Q_0, Q_1, a))\}$
od$\{P = [Q]_E\}$

The innermost update of L is intentionally clumsy and will be used to arrive at the algorithm given by Hopcroft and Gries. In the update of set L, if $(Q_0, b) \in L$ (for some $b \in V$) and Q_0 has been split into $Q_0 \setminus Q_0'$ and Q_0' then (Q_0, b) is replaced (in L) by $(Q_0 \setminus Q_0', b)$ and (Q_0', b).

Another observation due to Hopcroft is that splitting an equivalence class with respect to any two of (Q_0, b), (Q_0', b), and $(Q_0 \setminus Q_0', b)$ is the same as splitting the equivalence class with respect to all three [Hopc71, pp. 190–191], [Grie73, Lemma 6]. This is shown in the following intermezzo.

> We only prove that: if an equivalence class $\hat{Q}$ has been split with respect to (Q_0, b) and (Q_0', b), then it need not be split with respect to $(Q_0 \setminus Q_0', b)$. The two remaining cases can be proven analogously.

$$
\begin{array}{ll}
 & \neg \mathit{Splittable}(\hat{Q}, Q_0, b) \wedge \neg \mathit{Splittable}(\hat{Q}, Q_0', b) \\
\equiv & \quad \{\text{ De Morgan }\} \\
 & \neg(\mathit{Splittable}(\hat{Q}, Q_0, b) \vee \mathit{Splittable}(\hat{Q}, Q_0', b)) \\
\equiv & \quad \{\text{ Definition of } \mathit{Splittable} \} \\
 & \neg((\exists\, p, q : p, q \in \hat{Q} : T(p, b) \in Q_0 \not\equiv T(q, b) \in Q_0) \\
 & \quad \vee\, (\exists\, p, q : p, q \in \hat{Q} : T(p, b) \in Q_0' \not\equiv T(q, b) \in Q_0')) \\
\equiv & \quad \{\text{ Combine existential quantifiers }\} \\
 & \neg(\exists\, p, q : p, q \in \hat{Q} : (T(p, b) \in Q_0 \not\equiv T(q, b) \in Q_0) \vee (T(p, b) \in Q_0' \not\equiv T(q, b) \in Q_0')) \\
\Rightarrow & \quad \{\, Q_0' \subseteq Q_0 \,\} \\
 & \neg(\exists\, p, q : p, q \in \hat{Q} : T(p, b) \in Q_0 \setminus Q_0' \not\equiv T(q, b) \in Q_0 \setminus Q_0') \\
\equiv & \quad \{\text{ Definition of } \mathit{Splittable} \} \\
 & \neg \mathit{Splittable}(\hat{Q}, Q_0 \setminus Q_0', b)
\end{array}
$$

> The two remaining cases can be proven analogously.

For efficiency reasons we therefore choose the smallest two of the three (comparing $|Q_0|$, $|Q'_0|$, and $|Q_0 \setminus Q'_0|$) in the update of set L. If $(Q_0, b) \notin L$, then splitting has already been done with respect to (Q_0, b) and we add either (Q'_0, b) or $(Q_0 \setminus Q'_0, b)$ (whichever is smallest) to L. On the other hand, if $(Q_0, b) \in L$, then splitting has not yet been done and we remove (Q_0, b) from L and add (Q'_0, b) and $(Q_0 \setminus Q'_0, b)$ instead.

Lastly, we observe that by starting with $P = [Q]_{E_0} = \{Q \setminus F, F\}$ we have already split Q. As a result, we need only split with respect to either $(Q \setminus F, b)$ or (F, b) (for all $b \in V$) [Hopc71, pp. 190–191], [Grie73, Lemma 7].

This gives the algorithm[4]:

Algorithm 4.8 (Hopcroft):

$$
\begin{array}{l}
P := [Q]_{E_0};\\
L := (\mathbf{if}\ (|F| \leq |Q \setminus F|)\ \mathbf{then}\ \{F\}\ \mathbf{else}\ \{Q \setminus F\}\ \mathbf{fi}) \times V;\\
\{\text{invariant: } [Q]_E \sqsubseteq P \sqsubseteq [Q]_{E_0} \wedge L \subseteq (P \times V)\\
\qquad \wedge\, (\forall\, Q_0, Q_1, a : Q_0 \in Q \wedge (Q_1, a) \in L : \neg \mathit{Splittable}(Q_0, Q_1, a)) \Rightarrow (P = [Q]_E)\}\\
\mathbf{do}\ L \neq \emptyset \longrightarrow\\
\qquad \mathbf{let}\ Q_1, a : (Q_1, a) \in L;\\
\qquad P_{old} := P;\\
\qquad L := L \setminus \{(Q_1, a)\};\\
\qquad \{\text{invariant: } [Q]_E \sqsubseteq P \sqsubseteq P_{old}\}\\
\qquad \mathbf{for}\ Q_0 : Q_0 \in P_{old} \wedge \mathit{Splittable}(Q_0, Q_1, a)\ \mathbf{do}\\
\qquad\qquad Q'_0 := \{p : p \in Q_0 \wedge T(p, a) \in Q_1\};\\
\qquad\qquad P := P \setminus \{Q_0\} \cup \{Q_0 \setminus Q'_0, Q'_0\};\\
\qquad\qquad \mathbf{for}\ b : b \in V\ \mathbf{do}\\
\qquad\qquad\qquad \mathbf{if}\ (Q_0, b) \in L \longrightarrow L := L \setminus \{(Q_0, b)\} \cup \{(Q'_0, b), (Q_0 \setminus Q'_0, b)\}\\
\qquad\qquad\qquad [\!]\ (Q_0, b) \notin L \longrightarrow\\
\qquad\qquad\qquad\qquad L := L \cup (\mathbf{if}\ (|Q'_0| \leq |Q_0 \setminus Q'_0|)\ \mathbf{then}\ \{(Q'_0, b)\}\ \mathbf{else}\ \{(Q_0 \setminus Q'_0, b)\}\ \mathbf{fi})\\
\qquad\qquad\qquad \mathbf{fi}\\
\qquad\qquad \mathbf{rof}\\
\qquad \mathbf{rof}\\
\qquad \{(\forall\, Q_0 : Q_0 \in P : \neg \mathit{Splittable}(Q_0, Q_1, a))\}\\
\mathbf{od}\{P = [Q]_E\}
\end{array}
$$

Unfortunately, the running time analysis of this algorithm is complicated and is not discussed here. It is shown by both Hopcroft and Gries that it is $\mathcal{O}(|Q| \log |Q|)$, [Grie73, Hopc71].

[4]Part of the invariant has been omitted, being rather complicated to derive.

4.6 Computing $(p, q) \in E$

From the problem of deciding the structural equivalence of two types, it is known that equivalence of two states can be computed recursively by turning the fixed point definition of E into a functional program. If the (unmodified) definition were to be used directly as a functional program, there is the possibility of non-termination. In order for the functional program to work, it takes a third parameter along with the two states.

The following program, similar to the one presented in [t-Ei91], computes relation E pointwise; an invocation $equiv(p, q, \emptyset)$ determines whether states p and q are equivalent. It assumes that two states are equivalent (by placing the pair of states in S, the third parameter) until shown otherwise.

```
function equiv(p, q, S) is
    if {p, q} ∈ S ⟶ eq := true
    ▯ {p, q} ∉ S ⟶
        eq := (p ∈ F ≡ q ∈ F);
        eq := eq ∧ (∀ a : a ∈ V : equiv(T(p, a), T(q, a), S ∪ {{p, q}}))
    fi;
    return eq
```

The $\forall$ quantification can be implemented using a repetition

---

```
function equiv(p, q, S) is
    if {p, q} ∈ S ⟶ eq := true
    ▯ {p, q} ∉ S ⟶
        eq := (p ∈ F ≡ q ∈ F);
        for a : a ∈ V do
            eq := eq ∧ equiv(T(p, a), T(q, a), S ∪ {{p, q}})
        rof
    fi;
    return eq
```

The correctness of this program is shown in [t-Ei91]. Naturally, the guard eq can be used in the repetition (to terminate the repetition when $eq \equiv false$) in a practical implementation. This optimization is omitted here for clarity.

There are a number of methods for making this program more efficient. From Section 3.2 recall that $E = E_{(|Q|-2)\,\mathbf{max}\,0}$. We add a parameter k to function $equiv$ such that an invocation $equiv(p, q, \emptyset, k)$ returns $(p, q) \in E_k$ as its result. The recursion depth is bounded by $(|Q|-2)\,\mathbf{max}\,0$. The new function is

---

```
function equiv(p, q, S, k) is
    if k = 0 ⟶ eq := (p ∈ F ≡ q ∈ F)
    ▯ k ≠ 0 ∧ {p, q} ∈ S ⟶ eq := true
    ▯ k ≠ 0 ∧ {p, q} ∉ S ⟶
        eq := (p ∈ F ≡ q ∈ F);
        for a : a ∈ V do
            eq := eq ∧ equiv(T(p, a), T(q, a), S ∪ {{p, q}}, k − 1)
        rof
    fi;
    return eq
```

The third parameter S is made a global variable, improving the efficiency of this algorithm in practice. As a result, *equiv* is no longer a functional program in the sense that it now makes use of a global variable. The correctness of this transformation is shown in [t-Ei91]. We assume that S is initialized to $\emptyset$. When $S = \emptyset$, an invocation $equiv(p, q, (|Q| - 2)\,\mathbf{max}\,0)$ returns $(p,q) \in E$; after such an invocation $S = \emptyset$.

Algorithm 4.9 (Pointwise computation of E):

$$
\begin{array}{l}
\textbf{function}\ equiv(p,q,k)\ \ \textbf{is} \\
\quad \textbf{if}\ k = 0 \longrightarrow eq := (p \in F \equiv q \in F) \\
\quad [\!]\ k \neq 0 \wedge \{p,q\} \in S \longrightarrow eq := true \\
\quad [\!]\ k \neq 0 \wedge \{p,q\} \notin S \longrightarrow \\
\qquad eq := (p \in F \equiv q \in F); \\
\qquad S := S \cup \{\{p,q\}\}; \\
\qquad \textbf{for}\ a : a \in V\ \textbf{do} \\
\qquad\quad eq := eq \wedge equiv(T(p,a), T(q,a), k-1) \\
\qquad \textbf{rof}; \\
\qquad S := S \setminus \{\{p,q\}\} \\
\quad \textbf{fi}; \\
\quad \textbf{return}\ eq
\end{array}
$$

The procedure *equiv* can be memoized to further improve the running time in practice.

This algorithm does not appear in the literature.

4.7 Computing E by approximation from below

This latest version of function *equiv* can be used to compute E and D (assuming I_Q is the identity relation on states, and S is the global variable used in Algorithm 4.9):

Algorithm 4.10 (Computing E from below):

$$
\begin{array}{l}
S, G, H := \emptyset, \emptyset, I_Q; \\
\{\text{invariant: } (G \cup H) \subseteq (Q \times Q) \wedge G \subseteq D \wedge H \subseteq E\} \\
\textbf{do}\ (G \cup H) \neq Q \times Q \longrightarrow \\
\quad \textbf{let}\ p,q : (p,q) \in ((Q \times Q) \setminus (G \cup H)); \\
\quad \textbf{if}\ equiv(p,q,(|Q|-2)\,\mathbf{max}\,0) \longrightarrow H := H \cup \{(p,q)\} \\
\quad [\!]\ \neg equiv(p,q,(|Q|-2)\,\mathbf{max}\,0) \longrightarrow G := G \cup \{(p,q)\} \\
\quad \textbf{fi} \\
\textbf{od}\{G = D \wedge H = E\}
\end{array}
$$

Further efficiency improvements can be made as follows:

- We change the initialization of G to $G := ((Q \setminus F) \times F) \cup (F \times (Q \setminus F))$.
- We make use of the fact that $E = E^*$; obviously E is symmetrical, halving the required amount of computation. H can be updated at each iteration step by $H := H^*$ (provided the data-structures in the implementation are such that $*$-closure is easily implemented).
- Make use of the facts that

$$
\begin{array}{lll}
(p,q) \notin E & \Rightarrow & (\forall\ r,s : r \in Q \wedge s \in Q \\
 & & \qquad \wedge\ (\exists\ w : w \in V^* : T^*(r,w) = p \wedge T^*(s,w) = q) : ((r,s) \notin E) \\
(p,q) \in E & \Rightarrow & (\forall\ w : w \in V^* : (T^*(p,w), T^*(q,w)) \in E)
\end{array}
$$

  The first (respectively second) implication states that if p, q are two distinguished (respectively equivalent) states, and r, s are two states such that there is $w \in V^*$ and $T(r,w) = p \wedge T(s,w) = q$ (respectively $T(p,w) = r \wedge T(q,w) = s$), then r, s are also distinguished (respectively equivalent).

This algorithm has worse running time than the $\mathcal{O}(|Q| \log |Q|)$ of Hopcroft's algorithm [Hopc71, Grie73]. This algorithm has a significant advantage over all of the known algorithms: although function *equiv* computes E pointwise from above (with respect to $\sqsubseteq$, refinement), the main program computes E from below (with respect to $\subseteq$, normal set inclusion[5]). As such, any intermediate result H in the computation of E is usable in (at least partially) reducing the size of an automaton; all of the other algorithms presented have unusable intermediate results. This property has use in reducing the size of automata when the running time of the minimization algorithm is restricted for some reason (for example, in real-time applications).

[5]This is set inclusion, as opposed to refinement, since the intermediate result H may not be an equivalence relation during the computation.

5 Conclusions

The conclusions about minimization algorithms are:

- A derivation of Brzozowski's minimization algorithm was presented. This derivation proved to be easier to understand than either the original derivation (by Brzozowski), or the derivation given by van de Snepscheut. A brief history of the minimization algorithm was presented, hopefully resolving some misattributions of its discovery.
- The definition of equivalence (relation E) and distinguishability (relation D) as fixed points of certain equations proved easier to understand than many text-book presentations.
- The fixed point characterization of E made it particularly easy to calculate an upperbound on the number of approximation steps required to compute E (or D). This upperbound later proved useful in determining the running time of some of the algorithms, and also in making efficiency improvements to the pointwise algorithm.
- The definition of E as a greatest fixed point helped to identify the fact that all of the (previously) known algorithm computed E from above (with respect to refinement). As such, all of these algorithms have intermediate results that are not usable in minimizing the finite automaton.
- We successfully presented all of the well-known text-book algorithms in the same framework. Most of them were shown to be essentially the same, with minor differences in their loop structures. One exception was Hopcroft and Ullman's algorithm [HU79], which has a distinctly different loop structure. The presentation of that algorithm (with invariants) in this paper is arguably easier to understand than the original presentation. Our presentation highlights the fact that the main data-structure in the algorithm need not be a list — a set suffices.
- Hopcroft's minimization algorithm [Hopc71] was originally presented in a style that is not very understandable. As with Gries's paper [Grie73], we strive to derive this algorithm in a clear and precise manner. The presentation in this paper highlights two important facts: the beginning point for the derivation of this algorithm is one of the easily understood straightforward algorithms; and, the use of a list data-structure in both Hopcroft's and Gries's presentation of this algorithm is not necessary — a set can be used instead.
- This paper presented several new minimization algorithms, many of which were variations on the well-known algorithms. Two of the new algorithms (presented in Sections 4.6 and 4.7) are not derived from any of the well-known algorithms, and are significant in their own right.
  - An algorithm was presented that computes the relation E in a pointwise manner. This algorithm was refined from an algorithm used to determine the structural equivalence of types. Several techniques played important roles in the refinement:
    - the upperbound on the number of steps required to compute E was used to improve the algorithm by limiting the number of pairs of states that need to be considered in computing E pointwise;
    - memoization of the functional-program portion of the algorithm was used to reduce the amount of redundant computation.
  - A new algorithm was presented, that computes E from below. This algorithm makes use of the pointwise computation of E to construct and refine an approximation of E. Since the computation is from below, the intermediate results of this algorithm are usable in (at least partially) reducing the size of the *DFA*. This can be useful in applications where the amount of time available for minimization of the *DFA* is limited (as in real-time applications). In contrast, all of the (previously) known algorithms have unusable intermediate results.

A Some basic definitions

Convention A.1 (Powerset): For any set A we use $\mathcal{P}(A)$ to denote the set of all subsets of A. $\mathcal{P}(A)$ is called the *powerset* of A; it is sometimes written 2^A. $\Box$

Convention A.2 (Sets of functions): For sets A and B, $A \longrightarrow B$ denotes the set of all total functions from A to B, while $A \not\rightarrow B$ denotes the set of all partial functions from A to B. $\Box$

Remark A.3: For sets A, B and relation $C \subseteq A \times B$ we can interpret C as a function $C \in A \longrightarrow \mathcal{P}(B)$. $\Box$

Convention A.4 (Tuple projection): For an n-tuple $t = (x_1, x_2, \ldots, x_n)$ we use the notation $\pi_i(t)$ $(1 \le i \le n)$ to denote tuple element x_i; we use the notation $\bar{\pi}_i(t)$ $(1 \le i \le n)$ to denote the $(n-1)$-tuple $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots x_n)$. Both π and $\bar{\pi}$ extend naturally to sets of tuples. $\Box$

Convention A.5 (Relation composition): Given sets A, B, C (not necessarily different) and two relations, $E \subseteq A \times B$ and $F \subseteq B \times C$, we define relation composition (infix operator $\circ$) as:

$$E \circ F = \{(a, c) : (\exists\ b : b \in B : (a, b) \in E \wedge (b, c) \in F)\}$$

$\Box$

Convention A.6 (Equivalence classes of an equivalence relation): For any equivalence relation E on set A we denote the set of equivalence classes of E by $[A]_E$; that is

$$[A]_E = \{[a]_E : a \in A\}$$

Set $[A]_E$ is also called the *partition* of A induced by E. $\Box$

Definition A.7 (Index of an equivalence class): For equivalence relation E on set A, define $\sharp E = |[A]_E|$. $\sharp E$ is called the *index* of E. $\Box$

Definition A.8 (Alphabet): An *alphabet* is a non-empty set of finite size. $\Box$

Definition A.9 (Refinement of an equivalence relation): For equivalence relations E and E' (on set A), E is a *refinement* of E' if and only if $E \subseteq E'$. $\Box$

Definition A.10 (Refinement ($\sqsubseteq$) relation on partitions): For equivalence relations E and E' (on set A), $[A]_E$ is said to be a refinement of $[A]_{E'}$ (written $[A]_E \sqsubseteq [A]_{E'}$) if and only if $E \subseteq E'$. An equivalent statement is that $[A]_E \sqsubseteq [A]_{E'}$ if and only if every equivalence class (of A) under E is entirely contained in some equivalence class (of A) under E'. $\Box$

Definition A.11 (Tuple and relation reversal): For an n-tuple $(x_1, x_2, \ldots, x_n)$ define reversal as (postfix and superscript) function R:

$$(x_1, x_2, \ldots, x_n)^R = (x_n, \ldots, x_2, x_1)$$

Given a set A of tuples, we define $A^R = \{x^R : x \in A\}$. $\Box$

B Finite automata

In this section we define finite automata, some of their properties, and some transformations on finite automata. Most of these definitions are taken directly from [Wats93].

Definition B.1 (Finite automaton): A finite automaton (an *FA*) is a 6-tuple (Q, V, T, E, S, F) where

- Q is a finite set of states,
- V is an alphabet,
- $T \in \mathcal{P}(Q \times V \times Q)$ is a transition relation,
- $E \in \mathcal{P}(Q \times Q)$ is an ϵ-transition relation
- $S \subseteq Q$ is a set of start states, and
- $F \subseteq Q$ is a set of final states.

The definitions of an alphabet and function $\mathcal{P}$ are in Definition A.8 and Convention A.1 respectively. □

Remark B.2: We will take some liberty in our interpretation of the signatures of the transition relations. For example, we also use the signatures $T \in V \longrightarrow \mathcal{P}(Q \times Q)$, $T \in Q \times Q \longrightarrow \mathcal{P}(V)$, $T \in Q \times V \longrightarrow \mathcal{P}(Q)$, $T \in Q \longrightarrow \mathcal{P}(V \times Q)$, and $E \in Q \longrightarrow \mathcal{P}(Q)$. In each case, the order of the Q's from left to right will be preserved; for example, the function $T \in Q \longrightarrow \mathcal{P}(V \times Q)$ is defined as $T(p) = \{(a, q) : (p, a, q) \in T\}$. The signature that is used will be clear from the context. See Remark A.3. The definition of $\longrightarrow$ appears in Convention A.2. □

Since we only consider finite automata in this paper, we will frequently simply use the term *automata*.

B.1 Properties of finite automata

In this subsection we define some properties of finite automata. To make these definitions more concise, we introduce particular finite automata $M = (Q, V, T, E, S, F)$, $M_0 = (Q_0, V_0, T_0, E_0, S_0, F_0)$, and $M_1 = (Q_1, V_1, T_1, E_1, S_1, F_1)$.

Definition B.3 (Size of an *FA*): Define the size of an *FA* as $|M| = |Q|$. □

Definition B.4 (Isomorphism ($\cong$) of *FA*'s): We define isomorphism ($\cong$) as an equivalence relation on *FA*'s. M_0 and M_1 are isomorphic (written $M_0 \cong M_1$) if and only if $V_0 = V_1$ and there exists a bijection $g \in Q_0 \longrightarrow Q_1$ such that

- $T_1 = \{(g(p), a, g(q)) : (p, a, q) \in T_0\}$,
- $E_1 = \{(g(p), g(q)) : (p, q) \in E_0\}$,
- $S_1 = \{g(s) : s \in S_0\}$, and
- $F_1 = \{g(f) : f \in F_0\}$.

□

Definition B.5 (Extending the transition relation T): We extend transition relation $T \in V \longrightarrow \mathcal{P}(Q \times Q)$ to $T^* \in V^* \longrightarrow \mathcal{P}(Q \times Q)$ as follows:

$$T^*(\epsilon) = E^*$$

and (for $a \in V, w \in V^*$)

$$T^*(aw) = E^* \circ T(a) \circ T^*(w)$$

Operator $\circ$ (composition) is defined in Convention A.5. This definition could also have been presented symmetrically. $\Box$

Remark B.6: We also sometimes use the signature $T^* \in Q \times Q \longrightarrow \mathcal{P}(V^*)$. $\Box$

Definition B.7 (Left and right languages): The left language of a state (in M) is given by function $\overleftarrow{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(V^*)$, where

$$\overleftarrow{\mathcal{L}}_M(q) = (\cup\ s : s \in S : T^*(s, q))$$

The right language of a state (in M) is given by function $\overrightarrow{\mathcal{L}}_M \in Q \longrightarrow \mathcal{P}(V^*)$, where

$$\overrightarrow{\mathcal{L}}_M(q) = (\cup\ f : f \in F : T^*(q, f))$$

The subscript M is usually dropped when no ambiguity can arise. $\Box$

Definition B.8 (Language of an *FA*): The language of a finite automaton (with alphabet V) is given by the function $\mathcal{L}_{FA} \in FA \longrightarrow \mathcal{P}(V^*)$ defined as:

$$\mathcal{L}_{FA}(M) = (\cup\ s, f : s \in S \wedge f \in F : T^*(s, f))$$

$\Box$

Definition B.9 (*Complete*): A *Complete* finite automaton is one satisfying the following:

$$Complete(M) \equiv (\forall\ q, a : q \in Q \wedge a \in V : T(q, a) \neq \emptyset)$$

$\Box$

Definition B.10 (ϵ-*free*): Automaton M is ϵ-*free* if and only if $E = \emptyset$. $\Box$

Definition B.11 (Start-useful automaton): A $Useful_s$ finite automaton is defined as follows:

$$Useful_s(M) \equiv (\forall\ q : q \in Q : \overleftarrow{\mathcal{L}}(q) \neq \emptyset)$$

$\Box$

Definition B.12 (Final-useful automaton): A $Useful_f$ finite automaton is defined as:

$$Useful_f(M) \equiv (\forall\ q : q \in Q : \overrightarrow{\mathcal{L}}(q) \neq \emptyset)$$

$\Box$

Remark B.13: $Useful_s$ and $Useful_f$ are closely related by *FA* reversal (to be presented in Transformation B.22). For all $M \in FA$ we have $Useful_f(M) \equiv Useful_s(M^R)$. $\Box$

Definition B.14 (Useful automaton): A *Useful* finite automaton is one with only reachable states:

$$Useful(M) \equiv Useful_s(M) \wedge Useful_f(M)$$

$\Box$

Property B.15 (Deterministic finite automaton): A finite automaton M is deterministic if and only if

- it does not have multiple start states,
- it is ϵ-*free*, and
- transition function $T \in Q \times V \longrightarrow \mathcal{P}(Q)$ does not map pairs in $Q \times V$ to multiple states.

Formally,

$$Det(M) \equiv (|S| \leq 1 \wedge \epsilon\text{-}free(E) \wedge (\forall\ q, a : q \in Q \wedge a \in V : |T(q,a)| \leq 1))$$

□

Definition B.16 (Deterministic *FA*'s): *DFA* denotes the set of all deterministic finite automata. We call *FA* \ *DFA* the set of *nondeterministic finite automata.* □

Convention B.17 (Transition function of a *DFA*): For $(Q, V, T, \emptyset, S, F) \in DFA$ we can consider the transition function to have signature $T \in Q \times V \not\longrightarrow Q$. (A definition of $\not\longrightarrow$ appears in Convention A.2.) The transition function is total if and only if the *DFA* is *Complete*. □

Property B.18 (Weakly deterministic automaton): Some authors use a definition of a deterministic automaton that is weaker than *Det*; it uses left languages and is defined as follows:

$$Det'(M) \equiv (\forall\ q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overleftarrow{\mathcal{L}}(q_0) \cap \overleftarrow{\mathcal{L}}(q_1) = \emptyset)$$

$Det(M) \Rightarrow Det'(M)$ is easily proved. □

Definition B.19 (Minimality of a *DFA*): An $M \in DFA$ is minimal as follows:

$$Min(M) \equiv (\forall\ M' : M' \in DFA \wedge \mathcal{L}_{FA}(M) = \mathcal{L}_{FA}(M') : |M| \leq |M'|)$$

Predicate *Min* is defined only on *DFA*'s. Some definitions are simpler if we define a minimal, but still *Complete*, *DFA* as follows:

$$Min_C(M) \equiv (\forall\ M' : M' \in DFA \wedge Complete(M') \wedge \mathcal{L}_{FA}(M) = \mathcal{L}_{FA}(M') : |M| \leq |M'|)$$

Predicate Min_C is defined only on *Complete* *DFA*'s. □

Property B.20 (Minimality of a *DFA*): An M, such that $Min(M)$, is the unique (modulo $\cong$) minimal *DFA*, due to the Myhill-Nerode theorem. Introductory presentations of the theorem appear in [HU79, Wats93]. □

Property B.21 (An alternate definition of minimality of a *DFA*): For the purposes of minimizing a *DFA*, we use the definition (defined only on *DFA*'s):

$$\begin{aligned} Minimal(Q, V, T, \emptyset, S, F) \quad \equiv \quad & (\forall\ q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overrightarrow{\mathcal{L}}(q_0) \neq \overrightarrow{\mathcal{L}}(q_1)) \\ & \wedge\ Useful(Q, V, T, \emptyset, S, F) \end{aligned}$$

We have the property that (for all $M \in DFA$) $Minimal(M) \equiv Min(M)$. It is easy to prove that $Min(M) \Rightarrow Minimal(M)$. The reverse direction follows from the Myhill-Nerode theorem.

A similar definition that relates to Min_C is (also defined only on *DFA*'s):

$$\begin{aligned} Minimal_C(Q, V, T, \emptyset, S, F) \quad \equiv \quad & (\forall\ q_0, q_1 : q_0 \in Q \wedge q_1 \in Q \wedge q_0 \neq q_1 : \overrightarrow{\mathcal{L}}(q_0) \neq \overrightarrow{\mathcal{L}}(q_1)) \\ & \wedge\ Useful_s(Q, V, T, \emptyset, S, F) \end{aligned}$$

We have the property that (for all $M \in DFA$ such that $Complete(M)$) $Minimal_C(M) \equiv Min_C(M)$. The contrapositive of $Min_C(M) \Rightarrow Minimal_C(M)$ is easily proved, and the reverse direction also follows from the Myhill-Nerode theorem. □

B.2 Transformations on finite automata

Transformation B.22 (*FA* reversal): *FA* reversal is given by postfix (superscript) function $R \in FA \longrightarrow FA$, defined as:

$$(Q, V, T, E, S, F)^R = (Q, V, T^R, E^R, F, S)$$

Function R satisfies

$$(\forall\ M : M \in FA : (\mathcal{L}_{FA}(M))^R = \mathcal{L}_{FA}(M^R)).$$

□

Transformation B.23 (Removing start state unreachable states): Transformation $useful_s \in FA \longrightarrow FA$ removes those states that are not start-reachable:

$$\begin{array}{lcl} useful_s(Q, V, T, E, S, F) & = & \textbf{let} \quad U = SReachable(Q, V, T, E, S, F) \\ & & \textbf{in} \\ & & \qquad (U, V, T \cap (U \times V \times U), E \cap (U \times U), S \cap U, F \cap U) \\ & & \textbf{end} \end{array}$$

Function $useful_s$ satisfies

$$(\forall\ M : M \in FA : Useful_s(useful_s(M)) \wedge \mathcal{L}_{FA}(useful_s(M)) = \mathcal{L}_{FA}(M))$$

□

Transformation B.24 (Subset construction): The function *subset* transforms an ϵ-*free FA* into a *DFA* (in the **let** clause $T' \in \mathcal{P}(Q) \times V \longrightarrow \mathcal{P}(\mathcal{P}(Q))$)

$$\begin{array}{lcl} subset(Q, V, T, \emptyset, S, F) & = & \textbf{let} \quad T'(U, a) = \{(\cup\ q : q \in U : T(q, a))\} \\ & & \qquad\ \ F' = \{U : U \in \mathcal{P}(Q) \wedge U \cap F \neq \emptyset\} \\ & & \textbf{in} \\ & & \qquad (\mathcal{P}(Q), V, T', \emptyset, \{S\}, F') \\ & & \textbf{end} \end{array}$$

In addition to the obvious property that (for all $M \in FA$) $\mathcal{L}_{FA}(subset(M)) = \mathcal{L}_{FA}(M)$, function *subset* satisfies

$$(\forall\ M : M \in FA \wedge \epsilon\text{-}free(M) : Det(subset(M)) \wedge Complete(subset(M)))$$

It is also known as the "powerset" construction. □

Property B.25 (Subset construction): Let $M_0 = (Q_0, V, T_0, \emptyset, S_0, F_0)$ and $M_1 = subset(M_0)$ be finite automata. By the subset construction, the state set of M_1 is $\mathcal{P}(Q_0)$. We have the following property:

$$(\forall\ p : p \in \mathcal{P}(Q_0) : \overrightarrow{\mathcal{L}}_{M_1}(p) = (\cup\ q : q \in p : \overrightarrow{\mathcal{L}}_{M_0}(q)))$$

□

Definition B.26 (Optimized subset construction): The function *subsetopt* transforms an ϵ-*free FA* into a *DFA*. This function is an optimized version of *subset*.

$$\begin{array}{lcl} subsetopt(Q, V, T, \emptyset, S, F) & = & \textbf{let} \quad T'(U, a) = \{(\cup\ q : q \in U : T(q, a))\} \\ & & \qquad\ \ Q' = \mathcal{P}(Q) \setminus \{\emptyset\} \\ & & \qquad\ \ F' = \{U : U \in \mathcal{P}(Q) \wedge U \cap F \neq \emptyset\} \\ & & \textbf{in} \\ & & \qquad (Q', V, T' \cap (Q' \times V \times Q'), \emptyset, \{S\}, F') \\ & & \textbf{end} \end{array}$$

In addition to the property that (for all $M \in FA$) $\mathcal{L}_{FA}(subsetopt(M)) = \mathcal{L}_{FA}(M)$, function *subsetopt* satisfies

$$(\forall\ M : M \in FA \wedge \epsilon\text{-}free(M) : Det(subsetopt(M)))$$

□

References

[ASU86] AHO, A.V., R. SETHI, AND J.D. ULLMAN. *Compilers: Principles, Techniques, and Tools,* Addison-Wesley Publishing Co., Reading, M.A., 1988.

[AU92] AHO, A.V. AND J.D. ULLMAN. *Foundations of Computer Science,* Computer Science Press, New York, N.Y. 1992.

[Brau88] BRAUER, W. "On minimizing finite automata," EATCS Bulletin 35, June 1988.

[Brzo62] BRZOZOWSKI, J.A. "Canonical regular expressions and minimal state graphs for definite events," in *Mathematical theory of Automata,* Vol. 12 of MRI Symposia Series, pp. 529–561, Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1962.

[Brzo64] BRZOZOWSKI, J.A. "Derivatives of regular expressions," J. ACM 11(4): 481–494, 1964.

[Dijk76] DIJKSTRA, E.W. *A discipline of programming,* Prentice-Hall Inc., N.J., 1976.

[t-Ei91] TEN EIKELDER, H.M.M. "Some algorithms to decide the equivalence of recursive types," Computing Science Note 91/31, Eindhoven University of Technology, The Netherlands, 1991.

[Grie73] GRIES, D. "Describing an algorithm by Hopcroft," Acta Inf. 2: 97–109, 1973.

[HU79] HOPCROFT, J.E. AND J.D. ULLMAN. *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley Publishing Co., Reading, M.A., 1979.

[Hopc71] HOPCROFT, J.E. "An $n \log n$ algorithm for minimizing the states in a finite automaton," in *The Theory of Machines and Computations* (Z. Kohavi, ed.), pp. 189–196, Academic Press, New York, 1971.

[Huff54] HUFFMAN, D.A. "The synthesis of sequential switching circuits," J. Franklin Institute, 257(3): 161–190 and 257(4): 275–303, 1954.

[Mirk65] MIRKIN, B.G. "On dual automata," Kibernetika 2(1): 7–10, 1966.

[Moor56] MOORE, E.F. "Gedanken-experiments on sequential machines," in *Automata Studies,* (C.E. Shannon and J. McCarthy, eds.), pp. 129–153, Princeton University Press, Princeton, N.J., 1956.

[Myhi57] MYHILL, J. "Finite automata and the representation of events," WADD TR-57-624, pp. 112–137, Wright Patterson AFB, Ohio, 1957.

[Nero58] NERODE, A. "Linear automaton transformations," Proc. AMS 9: 541–544, 1958.

[RS59] RABIN, M.O AND D. SCOTT. "Finite automata and their decision problems," IBM J. Res. 3(2): 115–125, 1959.

[vdSn85] VAN DE SNEPSCHEUT, J.L.A. "Trace theory and VLSI design," PhD Thesis, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1985. Also available as Lecture Notes in Computer Science 200, Springer-Verlag, Berlin, 1985.

[vdSn93] VAN DE SNEPSCHEUT, J.L.A. *What computing is all about,* Springer-Verlag, New York, N.Y., 1993.

[Urba89] URBANEK, F. "On minimizing finite automata," EATCS Bulletin 39, Oct. 1989.

[Wats93] WATSON, B.W. "A taxonomy of finite automata constructions," Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993.

[Wood87] WOOD, D. *Theory of Computation,* Harper & Row, Publishers, New York, N.Y., 1987.