DESCRIBING AN ALGORITHM BY HOPCROFT

David Gries

TR 72-151

December 1972

Computer Science Department
Cornell University
Ithaca, New York 14850

Describing an Algorithm by Hopcroft

David Gries[†]
Cornell University
Ithaca, New York

Abstract

We give an algorithm, its correctness proof, and its proof of execution time bound, for finding the sets of equivalent states in a deterministic finite state automaton. The time bound is $K \cdot m \cdot n \cdot \log n$ where K is a constant, m the number of input symbols, and n the number of states. Hopcroft [3] has already published such an algorithm. The main reason for this paper is to illustrate the use of communicating an algorithm to others using a structured, top-down approach. We have also been able to improve on Hopcroft's algorithm by reducing the size of the algorithm and correspondingly complicating the proof of the running time bound.

[†]This research was supported by NSF Grant No. GJ-28176.

Describing an Algorithm by Hopcroft

David Gries
Cornell University
Ithaca, New York

Introduction

In [3], Hopcroft gives an algorithm for minimizing the number of states in a finite automaton. The running time has a bound $K \cdot m \cdot n \cdot \log(n)$ where K is a constant, m is the number of input symbols, and n is the number of states. Previous algorithms ran in time proportional to mn^2 or worse. Hopcroft's algorithm is thus a significant achievement.

Unfortunately the algorithm, its proof of correctness and the proof of running time, are all very difficult to understand. We present here a "structured", top-down approach to the presentation of the algorithm which makes it much clearer. One technique in achieving this is to present the correctness proof and algorithm hand-in-hand. In fact, once a few details of the proof are known, the algorithm is obvious. The correctness proof itself consists of a few simple lemmas each of which can be proved in a few lines.

Such a structured approach to presenting an algorithm seems to be longer and require more discussion than the conventional way. If the reader wishes to complain about this, he is challenged to first read Hopcroft's original paper and see whether he can understand it easily. The advantages of our approach will then be clear.

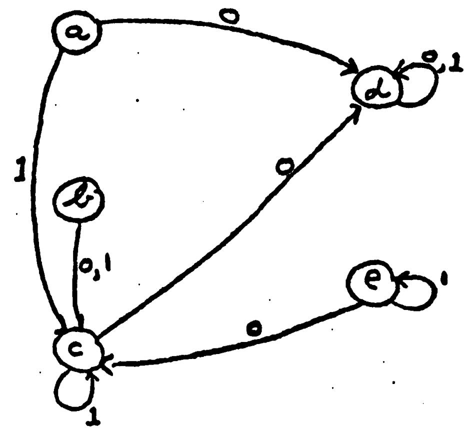
While we present the algorithm in top-down fashion, the reader should realize that it wasn't fully developed in this manner. A proof of a theorem may be neat and elegant, but this does not say that the proof was actually thought of in the same manner as it was presented. The same holds for programs and their descriptions.

The proof of the time bound is based on Hopcroft's work. However, it is important to notice that part of the proof given here depends on proving that a relation between values of variables remains invariant during execution of the algorithm, and this proof turns out to be significantly easier to follow than Hopcroft's. We have also been able to significantly cut down on the size of the program and the data structures needed, by complicating the proof of the running time. Our algorithm is therefore different in a few aspects from Hopcroft's.

Problem Definition

Let $A = (S,I,\delta,F)$ be a deterministic finite automaton, where S is a finite set of states, I is a finite set of input symbols, δ is a mapping from $S \times I$ into S, and $F \subseteq S$ is the set of final states. No initial state is specified since it is of no importance in what follows. The mapping δ is extended to $S \times I^*$ in the usual manner where I^* denotes the set of all finite strings (including the empty string ϵ) of symbols from I. States s and t are said to be <u>equivalent</u> if for each $x \in I^*$, $\delta(s,x) \in F$ if and only if $\delta(t,x) \in F$. We denote the empty set by ϕ. We want an algorithm which finds equivalent states of a finite automaton.

<u>Example</u>: Consider the automaton with $S = \{a,b,c,d,e\}$, $I = \{0,1\}$, $F = \{d,e\}$, and δ given by the arcs of diagram of Figure 1 in the conventional manner. a is not equivalent to b since $\delta(a,0) \in F$, $\delta(b,0) \notin F$. The final states d and e are <u>not</u> equivalent since $\delta(d,0) \in F$ but $\delta(e,0) \notin F$.



$S = \{a,b,c,d,e\}$

$I = \{0,1\}$

$F = \{d,e\}$

Sets of equivalent states:

$\{a,c\},\{b\},\{d\},\{e\}$

Figure 1 Finite state automaton

<u>The basic algorithm</u>

We are now ready to begin our discussion of the algorithm. The following discussion may seem slightly circuitous. We present it in this way so that the final algorithm is easy to understand and easy to prove correct.

<u>Definition 1</u>. <u>A</u> partititioning <u>of the states into blocks</u> B_1, $B_2, \ldots, B_p$ <u>is</u> acceptable <u>if</u> (a) <u>no block contains both a final and a nonfinal state</u>, and (b) <u>if s and t are equivalent states then they are in the same block</u>.

As an example, we present the following lemma:

<u>Lemma 2</u>. <u>The partitioning</u> $B_1 = F$, $B_2 = S-F$ <u>is acceptable</u>.

<u>Proof</u>: Part (a) of definition 1 is clearly satisfied. Secondly, since a final and a nonfinal state cannot be equivalent, (b) is satisfied. Q.E.D.

We wish to write an algorithm which generates the single partitioning such that two states s and t are equivalent if and only if they are in the same block. The following lemma characterizes this partitioning.

<u>Lemma 3</u>. <u>A partitioning</u> $B_1, B_2, \ldots, B_p$ <u>gives the blocks of equivalent states if and only if</u> (a) <u>the partitioning is acceptable and</u> (b) <u>for each pair of blocks</u> B_i, B_j <u>and symbol</u> $a \in I$,

(1) $s, t \in B_i$, $\delta(s,a) \in B_j$ <u>implies</u> $\delta(t,a) \in B_j$.

<u>Proof</u>: Suppose a partitioning is not acceptable. Then either a block contains a final and a nonfinal state (which can't be equivalent), or two equivalent states are in different blocks. In either case, the partitioning obviously doesn't yield the blocks of equivalent states.

Suppose (b) doesn't hold and the partitioning is acceptable. Thus we have (for some s, t, B_i, B_j and a)

$$s, t \in B_i \ , \ \delta(s,a) \in B_j \ , \ \delta(t,a) \notin B_j$$

Since $\delta(s,a)$ and $\delta(t,a)$ are in different blocks, they cannot be equivalent. Hence there exists a string $x \in I$ such that

$$\delta(\delta(s,a),x) \in F \ , \quad \text{and} \quad \delta(\delta(t,a),x) \notin F \quad \text{(or vica versa)}$$

Hence $\delta(s,ax) \in F$, $\delta(t,ax) \notin F$ and s and t are not equivalent but are in the same block.

To prove sufficiency, suppose that both (a) and (b) hold. Consider any two states s and t in a block B_i . We need only show that s and t are equivalent. We can easily prove by induction on the length of the string x, that $\delta(a,x)$ and $\delta(t,x)$ are always in the same block. Since a block cannot contain both a nonfinal and final state, the theorem is proved.

Q.E.D.

We can restate Lemma 3 slightly to describe how one can refine an acceptable partitioning to get another one:

<u>Lemma 4</u>: <u>Let</u> $B_1, \ldots, B_p$ <u>be an acceptable partitioning</u>. <u>Suppose there are two blocks</u> B_i <u>and</u> B_j <u>and a symbol</u> a <u>such that</u>

(2) $\quad s,t \in B_i$, $\delta(s,a) \in B_j$, <u>but</u> $\delta(t,a) \notin B_j$.

<u>Then</u> s <u>and</u> t <u>are not equivalent states</u>, <u>and we get a new acceptable partitioning by replacing</u> B_i <u>by the two blocks</u>

(3) $\quad \{s \in B_i | \delta(s,a) \in B_j\}$ <u>and</u> $\{s \in B_i | \delta(s,a) \notin B_j\}$.

We leave the proof to the reader. This splitting of B_i as described is called <u>splitting</u> B_i <u>with respect to the pair</u> (B_j,a) or simply <u>splitting</u> B_i <u>wrt</u> (B_j,a). We now write the following algorithm:

(4) $B_1 \leftarrow F;$ $B_2 \leftarrow S - F;$ [initially there are two blocks]

<u>while</u> $\exists$ a, B_i, B_j such that (2) holds <u>do</u>

SPLIT: split B_i wrt (B_j, a)

<u>end</u>

The algorithm must terminate since each execution of the loop statement SPLIT adds a new block, and the number of blocks in an acceptable partition can be no greater than the number of states of the finite automaton.

To show that after execution the partitioning yields the block of equivalent states, we use the following theorem about loops of the form "<u>while</u> B <u>do</u> S <u>end</u>" where S is a sequence of statements and B and P are relations (see Hoare[1]):

(5) $P \wedge B\ \{S\}\ P$ implies $P\ \{$<u>while</u> B <u>do</u> S end$\}\ P \wedge \neg B$

which means: If the truth of relations P and B before execution of sequence S implies the truth of P after execution of S , then the truth of P before executing "<u>while</u> B <u>do</u> S <u>end</u>" implies the truth of P and falsity of B after execution, providing the loop terminates.

For algorithm (4), let P be the relation "the partitioning is acceptable". By Lemma 2, P is true just before execution of the <u>while</u> loop, while by Lemma 4 execution of S always yields an acceptable partitioning. The relation B is "$\exists$ a, B_i, B_j such that (2) holds". Thus, P and $\neg$ B hold after execution of the loop, but these are the sufficient requirements described in Lemma 3 for the final partitioning to be the desired one.

Note that execution of algorithm (4) says nothing about the order in which the triples (a,B_i,B_j) are chosen for splitting. Hence the order does not matter. Let us get closer to our final algorithm by describing something about this ordering. Our refinement (6) determines all splittings wrt a pair (B_j,a) and then performs all these splittings at the same time.

(6) $B_1 \leftarrow F$; $B_2 \leftarrow S - F$;

<u>while</u> $\exists$ a, B_i, B_j such that (2) holds <u>do</u>

Determine the splittings of all blocks wrt (B_j,a);

Split each block as just determined.

<u>end</u>

The algorithm is not very efficient, since we must at least check every triple (a,B_i,B_j) for a pair s,t such that $s,t \in B_i$, $\delta(s,a) \in B_j$ but $\delta(t,a) \notin B_j$. In fact it looks like at least an mn^2 algorithm. Let us consider the possibility of maintaining a list L of all pairs (B_j,a) wrt which some blocks may have to be split. Another way to put it is that if we <u>know</u> it is not necessary to split any B (including B_j itself) wrt a pair (B_j,a) we <u>won't</u> put that pair on the list. We can then keep splitting until the list L becomes empty.

We must of course prove that we can correctly maintain such a list L. But let us first look more closely at what splitting does, and some consequences we can draw from this.

(7) The result of splitting all blocks wrt (B_j,a) is that any future block B (including the final blocks) satisfies <u>one</u>

of the following:

(a) for all $s \in B$ $\delta(s,a) \in B_j$, <u>or</u>

(b) for all $s \in B$ $\delta(s,a) \notin B_j$

Assertion 7 certainly holds for the blocks resulting from the splitting, and since each future block is a subset of one of these, (7) must continue to be true as the algorithm progresses. It should also be quite clear that this is the only result that splitting accomplishes. This discussion yields the following simple lemma.

<u>Lemma 5</u>. <u>Suppose all blocks have been split wrt</u> (B_j,a). <u>Then there is no need to split any future block wrt</u> (B_j,a).

This next lemma is important; without it, we would not be able to have running time proportional to $m \cdot n \cdot \log(n)$.

<u>Lemma 6</u>. <u>Suppose a block</u> B_j <u>is split into blocks</u> $\bar{B}_j$ <u>and</u> $\tilde{B}_j$. <u>Consider a symbol a. Splitting all blocks wrt to any</u> two <u>of the three pairs</u> (B_j,a), $(\bar{B}_j,a)$, <u>and</u> $(\tilde{B}_j,a)$ <u>performs the same function as splitting</u> all <u>blocks wrt all three pairs</u>.

<u>Proof</u>. Suppose we split all blocks wrt (B_j,a) and $(\bar{B}_j,a)$. This implies that each future block B satisfies <u>one</u> of the following:

$s \in B$ implies $\delta(s,a) \in B_j$ and $\delta(s,a) \in \bar{B}_j$, or

$s \in B$ implies $\delta(s,a) \in B_j$ and $\delta(s,a) \notin \bar{B}_j$, or

$s \in B$ implies $\delta(s,a) \notin B_j$ and $\delta(s,a) \notin \bar{B}_j$, or

$s \in B$ implies $\delta(s,a) \notin B_j$ and $\delta(s,a) \in \bar{B}_j$

Since $\bar{B}_j \cup \tilde{B}_j = B_j$ and $\bar{B}_j \cap \tilde{B}_j = \emptyset$, we infer that one of the following holds:

$s \in B$ implies $\delta(s,a) \in \tilde{B}_j$, or

$s \in B$ implies $\delta(s,a) \notin \tilde{B}_j$

This is precisely what splitting all blocks wrt $(\tilde{B}_j,a)$ accomplishes (see(7)). We leave to the reader to prove the rest of the theorem (in the same fashion) -- that splitting wrt $(\bar{B}_j,a)$ and $(\tilde{B}_j,a)$ accomplishes the task of splitting wrt (B_j,a); and that splitting wrt (B_j,a) and $(\tilde{B}_j,a)$ accomplishes the task of splitting wrt $(\bar{B}_j,a)$.

Q.E.D.

As an example of the use of lemma 6, we have the following:

Lemma 7. Let the two initial blocks be $B_1=F$ and $B_2=S-F$. For a given symbol a, it is necessary to split all blocks wrt only one of the pairs (B_1,a) and (B_2,a).

Proof. Consider lemma 6, with $B_j=S$, $\bar{B}_j=F$, and $\tilde{B}_j=S-F$. We already know that $\delta(s,a) \in B_j$ for any symbol a, so it is not necessary to split wrt (B_j,a). Hence we need only split wrt either $(\bar{B}_j,a)$ or $(\tilde{B}_j,a)$, but not both.

Q.E.D.

Now let us give the modified algorithm using a list L, and also state precisely the meaning of this list.

```
(8)  B1 ← F;  B2 ← S - F;  L = Ø;
     for each c ∈ I do†
         if B1 is smaller than B2 then add (B1,c) to L
         else add (B2,c) to L;
     end
     while L ≠ Ø do
       b:  Pick one pair (Bj,a) ∈ L;
       c:  Determine splittings of all blocks wrt (Bj,a);
       d:  L ← L - (Bj,a);
       e:  Split each block as determined in c;
       f:  /*Fix L according to the splits that occurred in step e*/
              for each block B just split into B̄ and B̃ (say) do †
                 for each c ∈ I do
                  if (B,c) ∈ L then
                     L ← L + (B̄,c) + (B̃,c) - (B,c)
                  else if B̄ is smaller than B̃ then add (B̄,c) to L
                       else add (B̃,c) to L
                 end
              end
     end
```

†The for each statement can be interpreted as follows. Let I contain elements $I_1, I_2, \ldots, I_k$. Then we can rewrite the statement for each $c \in I$ do S end as

$J \leftarrow 1$; while $J \leq k$ do $c \leftarrow I_J$; S; $J \leftarrow J + 1$ end

(9) <u>Meaning of List L</u>: L is a list of pairs (B_j,a) wrt which we must attempt to split all blocks so that either (7a) or (7b) will hold for each block. If B_j is a block and $(B_j,a) \notin L$ for some a, then either (7a) or (7b) already holds, or we are assured by other means that either (7a) or (7b) will hold when the algorithm terminates.

Now compare the modified algorithm (8) with its predecessor (6). The only change is the introduction of the list L and the statements to manipulate L. If we can show that the main <u>while</u> loop of (8) terminates, and that (9) is invariantly true before and after execution of this loop, then indeed the algorithm performs the desired task.

The list L must become empty since (1) each time we use a pair (B_j,a) it is deleted in statement d; (2) pairs are added to L in statement f only if a block is actually split into two distinct, nonempty blocks; and (3) the number of splits is bounded by n, the total number of states. Hence the algorithm terminates.

We must show that (9) is an invariantly true relation of the main loop. To do this, we again make use of the axiom

$$P \wedge B \{S\} P \qquad \text{implies} \qquad P\{\underline{\text{while}}\ B\ \underline{\text{do}}\ S\}\ P \wedge \neg B$$

where P is assertion (9). First of all (9) is true before execution of the loop, by lemma 7. Thus we need only show that executing statements b through f leave (a) true. L is changed in statements d and f, so we must look at them more closely.

Statement d deletes from L the pair (B_j,a). Since we are splitting all blocks wrt (B_j,a), Lemma 5 gives us the right to delete it. Statement f processes each block B (say) that is split into $\bar{B}$ and $\tilde{\hat{B}}$. By Lemma 6, for a given symbol c we need only split wrt <u>two</u> of the three pairs (B,c), $(\bar{B},c)$ and $(\tilde{\hat{B}},c)$. If (B,c) is in L, then statement f replaces it by $(\bar{B},c)$ and $(\tilde{\hat{B}},c)$. Suppose however that (B,c) is not in L. Statement (9) tells us we can assume that the result of splitting wrt (B,c) will be accomplished by other means before termination; hence we need only split wrt <u>one</u> of the pairs $(\bar{B},c)$ and $(\tilde{\hat{B}},c)$. Hence executing b-f leaves (9) true.

Algorithm (8) is thus correct; it performs the desired function. It remains to show that execution time is no worse than proportional to m·n·log(n). This requires some lemmas, a discussion of data structures needed to implement the states, blocks, and the list L, and some further refinements of statements b through f. While there is nothing really difficult, there is a lot of detail. We will try to structure the discussion so that the reader can stop reading as soon as he feels he understands the main thrust of the argument.

Proof of worst case running time $O(m \cdot n \cdot \log(n))$

In Table 1 we give the time necessary to execute the various components of the program (we exclude initialization for the moment). Note that we give the total time spent in executing each of the substatements b-f of the main loop, and not the time required for one execution. The k_i are constants. Since the total time is the sum of these individual times, total execution time is of the order $m \cdot n \cdot \log(n)$.

One basic point is that the loop iterates a maximum of $2 \cdot m \cdot n$ times before L becomes empty, which is proved in Lemma 8. With suitable data structures, then, statements b and d can be executed in constant time k_b and k_d, so that the total time spent in these is no more than $2 \cdot k_b \cdot m \cdot n$ and $2 \cdot k_d \cdot m \cdot n$, respectively.

Statement f need only look at blocks that are actually split. The proof of lemma 8 indicates that at most $2 \cdot n$ blocks can be created, which shows that at most $2 \cdot n$ blocks can be split. Assuming that the execution time of the conditional statement "if $(B,c) \in L$ ..." of statement f is bounded by a constant k_b we see that the total time spent in f is no worse proportional to $k_b \cdot 2 \cdot n \cdot m$.

We need only analyze statements c and e, which determine splittings and then make the splits. These are the only statements whose total execution time may be proportional to $m \cdot n \cdot \log(n)$. And indeed the analysis gives us more trouble. To perform the analysis we must look closer at just how to execute c and e.

Statement	Maximum total time spent	Proof and discussion
main <u>while</u> loop	$2 \cdot m \cdot n$ iterations	Lemma 8
b	$2 \cdot k_b \cdot m \cdot n$	Lemma 8, and above
c	$k_c \cdot m \cdot n \cdot \log(n)$	Lemma 9,10, below
d	$2 \cdot k_d \cdot m \cdot n$	Lemma 8, and above
e	$k_e \cdot m \cdot n \cdot \log(n)$	Lemma 10, and below
f	$2 \cdot k_f \cdot m \cdot n$	Lemma 8, and above
total	$O(m \cdot n \cdot \log(n))$	

Table 1. Worst case running time analysis for the algorithm

Let us now look closer at splitting. Splitting a block B_i wrt (B_j,a) replaces B_i by two blocks $\bar{B}_i$ and $\tilde{B}_i$ which satisfy

$$s \in \bar{B} \text{ implies } \delta(s,a) \in B_j$$
$$s \in \tilde{B} \text{ implies } \delta(s,a) \notin B_j$$

Given block B_i, let us split it by <u>removing</u> from it those states s such that $\delta(s,a) \in B_j$, and putting these states in a new block B_k, called B_i's twin. Thus B_i is split into B_i and B_k.

In order to determine the splitting of all blocks wrt (B_j,a), we need to make a list D (say) of all states which must be removed from blocks -- which satisfy the property $\delta(s,a) \in B_j$. Statement c thus looks like:

(10): c: **Determine the splittings of all blocks wrt (B_j,a):**

$D \leftarrow \phi$;
<u>for</u> <u>each</u> $s \in B_j$ <u>do</u>
   <u>if</u> $\delta^{-1}(s,a) \neq \phi$ <u>then</u> $D \leftarrow D \cup \delta^{-1}(s,a)$
<u>end</u>

Statement c seems to have an extra test "if $\delta^{-1}(s,a) \neq \phi$". This test has been inserted just to make the proof of running time a bit clearer.

Statement e, which actually splits blocks, could be written as

e: <u>for</u> <u>each</u> block B_i in the partition <u>do</u>
   $B_k \leftarrow B_i \cap D$; ($B_k$ is a newly generated block - B_i's twin)
   $B_i \leftarrow B_i - B_k$;
   <u>end</u>

While correct, statement e is too inefficient since each time it is executed it must manipulate <u>each</u> block, and this would lead to a $m \cdot n^2$ algorithm. Hence we must refine e further to look only at blocks which have a chance of being partitioned — which contain states in D. We can also recognize a case where removing states is unnecessary. If for <u>all</u> $s \in B$, $\delta(s,a) \in B_j$ then $\{s \in B_i | \delta(s,a) \notin B_j\}$ is empty. We end up with the following algorithm e:

(11) e: Split each block as just determined:

```
for each s ∈ D do
    BI ← block number in which  s  appears;
    if all s ∈ BI have δ(s,a) ∈ Bj
    then [no need to split block — do nothing]
    else begin if BI has no twin BK yet then
                        generate BI's twin BK and set BK ← ϕ;
                  Move s from BI to its twin BK
          end
end
```

We are now ready to perform a worst case analysis of the total time spent in executing statements c and e. We first of all prove in lemma 9 that the total number of times the condition "if $\delta^{-1}(s,a) \neq 0$" of statement c is executed has a bound of $m \cdot n \cdot \log(n)$. This is the most difficult part of the running time proof. The total time spent in the rest of statement c (the statement $D \leftarrow D \cup \delta^{-1}(s,a)$) and the total time spent in statement e are both proportional to the total number of states stored into D throughout the execution of the algorithm. We show in Lemma 10 that the total number of states put into D is bounded from above by $m \cdot n \cdot \log(n)$, which completes our analysis.

Two important ideas helped us in reducing the running time to $m \cdot n \cdot \log(n)$. The first was that if a block B is split into $\bar{B}$ and $\tilde{B}$ we need only split wrt two of the three pairs (B,a), $(\bar{B},a)$, and $(\tilde{B},a)$. The second was that in case we need put only one of $(\bar{B},a)$ and $(\tilde{B},a)$ in L, we should put the one whose block ($\bar{B}$ or $\tilde{B}$) contains the fewest number of states.

Lemma 8: The maximum number of iterations in the main loop of algorithm (8) is $2 \cdot m \cdot n$.

Proof: We show that the maximum number of pairs put into L is $2 \cdot m \cdot n$. Since each iteration deletes one pair from L, the lemma follows.

For each different block B created and for each symbol a, the pair (B,a) is put in L at most once. We need only show that at most $2 \cdot n$ blocks can be created. Consider the binary tree consisting of the blocks created. The root node is the block consisting of all states; its two sons are the blocks $B_1=F$ and $B_2=S-F$. For each block B its sons are the blocks $\bar{B}$ and $\breve{B}$ into which it is split. The number of end nodes of this tree is bounded by n, the maximum number of blocks possible at any one time. Hence, the binary tree can contain at most $2 \cdot n$ nodes. Q.E.D.

<u>Lemma 9</u>. <u>The total number of times the condition "if $\delta^{-1}(s,a) \neq \phi$ then" of statement c (11) is executed is bounded from above by</u> $m \cdot n \cdot \log(n)$.

<u>Proof</u>: Consider a particular symbol $\bar{a}$. We prove below that the number of times the above condition is executed with $a = \bar{a}$ is bounded by $n \cdot \log(n)$. Since there are m symbols, the lemma will follow immediately.

Let us introduce a new variable COUNT. Initially we set COUNT to 0, and we change statement d to

(12) d: <u>begin</u> <u>if</u> $a = \bar{a}$ <u>then</u> COUNT ← COUNT + b_j;
L ← L − (B_j, a);
<u>end</u>;

where b_j is the number of states in B_j. This has no effect on the output, since COUNT is not used elsewhere in the algorithm. But note that since b is the number of states in B_j, after termination COUNT contains the number of times the condition "<u>if</u> $\delta^{-1}(s,a) \neq 0$ <u>then</u>" is executed with $a = \bar{a}$. Thus we need only prove that $n \cdot \log(n) \geq$ COUNT.

At each point of execution, let the blocks be called $B_1, B_2, \ldots$ and let $K = \{B_i \mid (B_i,\bar{a}) \in L\}$, $\bar{K} = \{B_i \mid (B_i,a) \notin L\}$. Consider the expression

$$T = n \cdot \log(n) - \sum_{B_i \in K} b_i \log b_i - \sum_{B_i \in \bar{K}} b_i \log \frac{b_i}{2}$$

where $\log c$ means $\log_2 c$. Remember, b_i is the number of states in B_i . Since the total number of states is n , we always have $n \log(n) \geq T \geq 0$. We claim that the relation P:

(13) $\qquad T \geq \text{COUNT}$

holds at the end of the algorithm. Using our theorem

$$P \wedge B \{S\} P \text{ implies } P\{\underline{\text{while}}\ B\ \underline{\text{do}}\ S\ \underline{\text{end}}\}\ P \wedge \neg B\ ,$$

we need only show that P holds before execution of the main loop "<u>while</u> $L \neq \phi$ <u>do</u> ... <u>end</u>", and that executing S does not change the truth of P . P certainly holds before execution of the loop, since we initialized COUNT to 0.

Consider now an execution of S. Relation (13) may change only if either T or COUNT is changed. Since only statements d and f change either one, let us analyze these carefully.

Statement d (12) changes both T and COUNT. If (and only if) (d) adds b_j to COUNT, it changes T by deleting $(B_j,\bar{a})$ from L . If we show that this latter change also increases T by b_j , then (13) remains invariant. Deleting $(B_j,\bar{a})$ from L has the effect of replacing (in T) a term

$- b_j \log b_j$ by $- b_j \log \frac{b_j}{2}$. But

$$b_j \log b_j - b_j \log \frac{b_j}{2} = b_j(\log b_j - \log b_j + \log_2 2) = b_j .$$

Hence (13) remains invariant.

Statement f changes T by adding to and deleting pairs (B_j,a) from L. We show that T never decreases. Suppose a block B is split into blocks $\bar{B}$ and $\tilde{B}$. Then $b = \bar{b} + \tilde{b}$ where we assume without loss of generality that $\bar{b} \leq \tilde{b}$. Suppose $(B,\bar{a})$ was in L. Then $-b \log b$ is replaced by $-\bar{b} \log \bar{b} - \tilde{b} \log \tilde{b}$. We have

$$\bar{b} \log \bar{b} + \tilde{b} \log \tilde{b} \leq (\bar{b} + \tilde{b}) \log \tilde{b} < b \log b$$

So the change increases T. Now suppose $(B,\bar{a})$ was not in L. Then $(\bar{B},\bar{a})$ is put in L. Hence a term $- b \log \frac{b}{2}$ is replaced by $- \bar{b} \log \bar{b} - \tilde{b} \log \frac{\tilde{b}}{2}$. Since

$$\bar{b} \log \bar{b} + \tilde{b} \log \frac{\tilde{b}}{2} \leq \bar{b} \log \frac{b}{2} + \tilde{b} \log \frac{b}{2} = b \log \frac{b}{2} ,$$

T again does not decrease. Hence (13) remains invariant.

Q.E.D.

<u>Lemma 10</u>: <u>The</u> total <u>number of states put into</u> D <u>during executions of statement</u> c (10) <u>is bounded by</u> $m \cdot n \cdot \log(n)$.

<u>Proof</u>: Consider <u>one particular</u> transition $\delta(s,a) = t$. We show that the number of times a pair (B_j,a) with $t \in B_j$ can be in L, is bounded by $\log(n)$. Hence the number of times that

the particular state s is added to D because of this transition $\delta(s,a) = t$ is bounded by $\log(n)$. Since there are at most $m \cdot n$ transitions, the lemma will follow immediately.

Suppose t appears in B_j , $(B_j,a) \in L$, and that the pair (B_j,a) is chosen in statement b. Then s is added to D in statement c. We claim that the <u>next</u> time a pair (B,a) with $t \in B$ is put in L , that $b \le b_j/2$. Why? We never have to split wrt (B_j,a) again. If B_j itself is split into $\bar{B}$ and $\tilde{B}$ with $\bar{b} \le b_j/2 \le \tilde{b}$, then only $(\bar{B},a)$ is put in L .

The fact that if (B,a) with $t \in B$ is put into L then $b \le b_j/2$, together with $b_j \le n/2$, means that such a pair (B,a) can be chosen in step b at most $\log_2 n$ times.

Q.E.D.

<u>The Complete Algorithm</u>

It may be advantageous for the reader to see the whole algorithm in one piece. We present it here, with a few changes needed because of the refinements of c and e. Following the algorithm we present the data structures used in the actual PL/I implementation.

(14) Initialize: $B_1 \leftarrow F$; $B_2 \leftarrow S - F$; $L = \phi$;

<u>for</u> <u>each</u> $c \in I$ <u>do</u>

<u>if</u> $b_1 \le b_2$ <u>then</u> add (B_1,c) to L <u>else</u> add (B_2,c) to L

<u>end</u>;

<u>while</u> $L \ne \phi$ <u>do</u>

```
b: Pick one pair (Bj,a) ∈ L;
c: Determine splittings of all blocks wrt (Bj,a):
   D ← φ;
   for each s ∈ Bj do D ← D ∪ δ⁻¹(s,a) end;
d: L ← L - {(Bj,a)};
e: Split each block as determined in c:
   for each s ∈ D do
        BI ← block number in which s appears;
        if all s ∈ BI have δ(s,a) ∈ Bj then
        else begin if BI has no twin BK yet then
                           generate BI's twin BK and set BK ← φ;
                     Move s from BI to its twin BK
             end
   end;
f: Fix L according to splits that occurred:
   for each block BI split into BI and its twin BK do
        for each c ∈ I do
            if (BI,c) ∈ L then add (BK,c) to L
           else if bi ≤ bk then add (BI,c) to L
               else add (BK,c) to L
       end
   end f;
end
```

```
EQUIVSTATE: PROCEDURE(DELTA,F,N,M);
     /*THE FA HAS N STATES 1,2,3,....,N AND M SYMBOLS 1,2,3,....,M.        */
     /*DELTA IS THE MAPPING FUNCTION: DELTA(S,C) = T (OR 0 IF NO SUCH */
     /*ARC EXISTS) WHERE S AND T ARE STATES AND C A SYMBOL. F(S)=1(0) */
     /*MEANS THAT S IS A FINAL (NONFINAL) STATE. THE PROCEDURE PRINTS */
     /*OUT THE BLOCKS OF EQUIVALENT STATES.                               */
     /* TIME REQUIRED IS PROPORTIONAL TO M*N*LOG(N).                      */
     /* SPACE IS ROUGHLY 10*N + 4*M*N.                                     */

     DECLARE (DELTA(*,*), F(*), N, M) FIXED BINARY;

     DECLARE (B(2*N), BF(2*N), BB(2*N), BFREE) FIXED BINARY;
      /*REPRESENTS THE BLOCKS. THE FIRST N ELEMENTS DESCRIBE THE          */
      /*STATES WHILE ELEMENTS N+1,N+2,... DESCRIBE THE BLOCKS.            */
      /*FOR STATE S, B(S) IS THE BLOCK IN WHICH IT IS (ITS INDEX).        */
      /*FOR BLOCK I, B(I) IS THE NUMBER OF STATES CURRENTLY IN IT.        */
      /*                                                                   */
      /*THE BF (BB) ARRAY IS A CIRCULAR FORWARD (BACKWARD) CHAIN          */
      /*LINKING THE STATES IN A BLOCK WITH THAT BLOCK. FOR EXAMPLE, IF*/
      /*BLOCK I HAS STATES S1 AND S2, THEN                                 */
      /*    B(I) = 2    BF(I) = S1    BB(I) = S2                           */
      /*    B(S1)= I    BF(S1)= S2    BB(S1)= I                            */
      /*    B(S2)= I    BF(S2)= I     BB(S2)= S1                           */
      /*                                                                   */
      /*THE BLOCKS IN USE ARE N+1, N+2,...,BFREE.                          */

     DECLARE (LF(M:  N*M+M), LB(M:  N*M+M), LST) FIXED BINARY;
      /*REPRESENTS THE LIST OF BLOCK-SYMBOL PAIRS L. PAIR (BI,C) IS IN*/
      /*L IFF LF((BI-N)*M+C-I) > 0. LF AND LB ARE FORWARD AND BACKWARD*/
      /*CHAINS LINKING ALL PAIRS IN L WITH THE ENTRY LF(LST) WHERE       */
      /*LST=N*M+M. SEE DESCRIPTION OF BLOCKS FOR THIS CONCEPT.           */


     DECLARE    (DM(N,M), DMLIST(N*M)) FIXED BINARY;
      /*THE SETS DELTA INVERSE(S,C). LET T=DM(S,C). IF T=0, THE          */
      /*INVERSE SET IS EMPTY. IF T>0, THE SET CONSISTS OF DMLIST(T),    */
      /*DMLIST(T+1), ..., WITH THE LAST ONE IN THE SET BEING NEGATIVE.*/

     DECLARE (TWIN(2*N),   SD(N+1:2*N), SPLITNO) FIXED BINARY;
      /*DURING STEP C, TWIN(1),...,TWIN(SPLITNO) IS A LIST OF THOSE      */
      /*BLOCKS BEING SPLIT, WHILE FOR EACH SUCH BLOCK BI, TWIN(BI)        */
      /*YIELDS BI'S TWIN AND SD(BI) IS THE NUMBER OF STATES IN BI         */
      /*WITH DELTA(S,A) IN BJ. IF SD(BI) = B(BI), THERE IS NO NEED TO */
      /*SPLIT BI. THESE ARRAYS MUST BE SET TO 0 BY STEP C.                 */

     DECLARE (D(N), ND) FIXED BINARY;
      /*FOR A FIXED (BJ,A), D(1),...,D(ND) HOLDS THE DELTA INV(BJ,A). */
```

A more complicated algorithm with an easier proof.

Half the difficulty of the proof of running time is in Lemma 9. Let us suppose that, for each block B_i and symbol c, we maintain a set

$$\hat{B}_i(c) = \{s \in B_i \mid \delta^{-1}(s,c) \neq \phi\}$$

We can then change statement c (10) to

```
D ← φ;
For each s ∈ B̂_j(a) do
    D ← D ∪ δ^{-1}(s,a);
end
```

This eliminates the need for Lemma 9. Of course the program is more complicated because we must maintain the sets $\hat{B}_i(c)$, which change whenever B_i changes. These sets can be maintained as doubly-linked lists so that the operations of insertion and deletion of states, and determining the size of each $\hat{B}_i(c)$ can be performed in a fixed amount of time. This has the added advantage that if $\hat{B}_i(c) = \phi$, there is no need to add (B_i,c) to L. The running time is still proportional to $m \cdot n \cdot \log(n)$ in the worst case. This is Hopcroft's original algorithm.

The Program

The algorithm was written in PL/I, using the data structures described, and consists of 170 PL/I statements. It was not completed and run until this paper was finished (except for this paragraph and revisions). The program was tested using the

After syntax errors due to unfamiliarity with PL/I were fixed, three further errors were discovered: (1) a mistake in the input statement used to read in test data, (2) a mistake in the output statement used to print results, and (3) two cards were out of order in the minimization procedure itself. Other than that, no errors were detected.

Acknowledgements

I would like to thank John Hopcroft for numerous discussions about his algorithm, and Steven Brown, Donald Johnson, Tom Szymanski, and John Williams, for critically reading the manuscript and suggesting improvements. Thanks are also due to the referee, Tony Hoare, who made numerous improvements. All referees should referee the way he does.

REFERENCES

[1] Hoare, C.A.R., An axiomatic basis for computer programming, CACM 12 (October 1969), p. 576-583.

[2] Dijkstra, E.W., Notes on structured programming, EWD 249, Technical University Eindhoven, The Netherlands, 1969.

[3] Hopcroft, J., An n log n algorithm for minimizing states in a finite automaton, in Theory of Machines and Computations, Academic Press, New York, 1971, p. 189-196.