AN N LOG N ALGORITHM FOR MINIMIZING STATES IN A FINITE AUTOMATON

BY

JOHN HOPCROFT

STAN-CS-71-190
January, 1971

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

AN N LOG N ALGORITHM FOR MINIMIZING STATES IN A FINITE AUTOMATON

John Hopcroft

Abstract

An algorithm is given for minimizing the number of states in a finite automaton or for determining if two finite automata are equivalent. The asymptotic running time of the algorithm is bounded by knlogn where k is some constant and n is the number of states. The constant k depends linearly on the size of the input alphabet.

This research was supported by the National Science Foundation under grant number NSF-GJ-96, and the Office of Naval Research under grant number N-00014-67-A-0112-0057 NR 044-402. Reproduction in whole or in part is permitted for any purpose of the United States Government.

AN n log n ALGORITHM FOR MINIMIZING STATES IN A FINITE AUTOMATON

John Hopcroft

Stanford University

Introduction

Most basic texts on finite automata give algorithms for minimizing the number of states in a finite automaton [1, 2]. However, a worst case analysis of these algorithms indicate that they are n^2 processes where n is the number of states. For finite automata with large numbers of states, these algorithms are grossly inefficient. Thus in this paper we describe an algorithm for minimizing the states in which the asymptotic running time in a worst case analysis grows as $n \log n$. The constant of proportionality depends linearly on the number of input symbols. Clearly the same algorithm can be used to determine if two finite automata are equivalent.

The essence of previously published algorithms was to first partition the states according to their outputs. The blocks of the partitions are then repeatedly refined by examining the successor state on a given input for each state in the block. States whose successor states on a given input are in different blocks are placed in separate blocks. When no further refinement is possible, all states in the same block of the partition can be shown to be equivalent. Consider the example in Figure 1. The initial partition is (1,2,3,4,5)(6). Since on input 0, the successor states of states 1, 2, 3 and 4 are in the first block of the partition and the successor of state 5 is in the second block, the first iteration refines the partition into the blocks (1,2,3,4)(5) and (6). Successive refinements yield (1,2,3)(4)(5)(6); (1,2)(3)(4)(5)(6) and (1)(2)(3)(4)(5)(6). Thus, in this example all pairs of states are inequivalent. For this example it is seen that as many as n iterations may be required and the total number of steps needed to execute the algorithm if implemented in a straightforward fashion on a digital computer is n^2.

State	Input 0	Input 1	Output
1	2	1	0
2	3	2	0
3	4	3	0
4.	55	4	0
5	6	5	0
6	6	6	1
	Next State		

Figure 1

The algorithm proposed in this paper may also require n iterations but the work per iteration summed over all iterations yields only $n \log n$. We illustrate the algorithm by an example before specifying it in detail. Extensive use of list processing is employed to reduce the computation time. First the state table is inverted to obtain the table shown in Figure 2. The states are partitioned according to their outputs (1,2,3,4,5)(6). Next a block and an input symbol on which the partition is refined are selected. Assume the block (6) and input 0 are selected. The states in each block are further partitioned depending on whether on input 0 their next state is in block (6) or not. Thus the next partition is (1,2,3,4)(5)(6). Note that had we partitioned on the block (1,2,3,4,5) and input 0 we would have obtained the same result.

More generally, once we have partitioned on a block and an input symbol, we need never partition on that block and input symbol again until the block is split and then we need only partition on one of the two subblocks. Since the time needed to partition on a block is proportional to the transitions into the block and since we can always select the half with fewer transitions, the total number of steps in the algorithm is bounded by n log n .

States	Input 0	Input 1	Output
1	-	1	0
2	1	2	0
3	2	3	0
4	3	4	0
5	4	5	0
6	5,6	6	1

previous state

Figure 2

Formal description of the algorithm

Let $A = (S,I,\delta,F)$ be a finite automaton where S is a finite set of states, I is a finite set of inputs, 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 length strings of symbols from I . States s and t are said to be equivalent if for each x in I^*, $\delta(s,x)$ is in F if and only if $\delta(t,x)$ is in F .

The algorithm for finding the equivalence classes of S is described below.

Step 1. $\forall s \in S$ and $a \in I$ construct

$$\delta^{-1}(s,a) = \{t \mid \delta(t,a) = s\} .$$

Step 2. Construct $B(1) = F$, $B(2) = S-F$ and for each a in I and $1 \le i \le 2$ construct

$$a(i) = \{s \mid s \in B(i) \text{ and } \delta^{-1}(s,a) \ne \emptyset\} .$$

Step 3. Set $k = 3$

Step 4. $\forall a \in I$ construct

$$L(a) = \begin{cases} \{1\} & \text{if } |a(1)| \le |a(2)| \\ \{2\} & \text{otherwise} \end{cases}$$

Step 5. Select a in I and i in $L(a)$. The algorithm terminates when $L(a) = \emptyset$ for each a in I .

Step 6. Delete i from $L(a)$.

Step 7. $\forall j < k$ st $\exists t$ in $B(j)$ with $\delta(t,a) \in a(i)$ perform steps 7a, 7b, 7c, and 7d.

Step 7a. Partition $B(j)$ into $B'(j) = \{t \mid \delta(t,a) \in a(i)\}$ and $B''(j) = B(j)-B'(j)$.

Step 7b Replace $B(j)$ by $B'(j)$ and construct $B(k) = B''(j)$. Construct corresponding $a(j)$ and $a(k)$ for each a in I .

Step 7c. $\forall a \in I$ modify L(a) as follows.

$$L(a) = \begin{cases} L(a) \cup \{j\} & \text{if } j \notin L(a) \text{ and } 0 < |a(j)| \leq |a(k)| \\ L(a) \cup \{k\} & \text{otherwise} \end{cases}$$

Step 7d. Set $k = k+1$.

Step 8. Return to Step 5.

Correctness of algorithm

The claim is made that on termination of the algorithm two states are equivalent if and only if they are in the same block. The algorithm must terminate since the only times that an index is added to L(a) for some a in I are in Step 4 which is executed only once and in Step 7c. An index is added at Step 7c only after a refinement of a block of the partition. Each time Step 6 is executed, an index is removed from L(a) for some a . Thus the algorithm must terminate.

It is easily shown by induction on the number of times Step 7a is executed that if s is in B(i) and t is in B(j) , $i \neq j$, then s is not equivalent to t . Clearly, it is true the first time Step 7a is executed since only two blocks exist, B(1) containing only final states and B(2) containing only nonfinal states. Blocks are refined at Step 7a only when successor states on a given input have previously been shown to be inequivalent.

To see that two inequivalent states cannot be in the same block when the algorithm terminates, assume that states s and t are in B(i) and that s and t are not equivalent. Without loss of generality, assume $\delta(s,a)$ is in B(j) and $\delta(t,a)$ is in B(k) where $j \neq k$. (If $\delta(s,a)$ and $\delta(t,a)$ are in the same block then there exists a shortest x such that $\delta(s,x)$ and $\delta(t,x)$ are in distinct blocks. Clearly an x exists and hence a shortest x since for some x one or the other of $\delta(s,x)$ and $\delta(t,x)$ but not both is in a final state and each block consists solely of final or solely of nonfinal states. Let a be the last symbol of x and write x = ya . Then $\delta(s,y)$ and $\delta(t,y)$ are in the same block, $\delta(s,y)$ and $\delta(t,y)$ are not equivalent and $\delta(\delta(s,y),a)$ and $\delta(\delta(t,y),a)$ are in different blocks. Replace s by $\delta(s,y)$ and replace t by $\delta(t,y)$.) Consider the point at which the block containing $\delta(s,a)$ and $\delta(t,a)$ was partitioned so that $\delta(s,a)$ and $\delta(t,a)$ first appeared in separate subblocks. At that point one of the two subblocks was placed in L(a) . When this subblock is removed from L(a) , the block containing s and t is partitioned with s and t going into separate subblocks. Thus s and t cannot both be in B(i) , a contradiction.

Analysis of the running time

The running time of the algorithm is clearly dependent on the implementation. The algorithm has been programmed in ALGOL. Since the implementation consists of approximately 300 ALGOL statements we shall simply indicate how the various steps were implemented and discuss informally their running time.

The sets such as $\delta^{-1}(s,a)$, L(a) , etc. were represented by linked lists in such a way that an item could be added or deleted at the beginning of the list in a fixed number of steps. Vectors were also maintained to indicate if a state was or was not on a given list. This eliminates searching a list simply to determine if the item is on the list and is essential in Step 7c. The sets B(i) and a(i) were represented as doubly linked lists so that an item could be added or deleted anywhere in the list in a fixed number of steps once the position is given. The structure was such that given a state s , its position in B(i) and a(i) could be determined in a fixed number of steps.

Steps 1, 2, and 4 are executed only once and require time proportional to the product of the number of states times the number of input symbols. Steps 5 through 8 form a simple loop. The time necessary to traverse the loop for given a in I and i in L(a) is proportional to the number of state transitions on input a terminating on states in B(i) . (To see this, note that Steps 5,6 and 8 are finite. In Step 7 we do not need to examine B(j) for each $j < k$ to see if there exists a t in B(j) with $\delta(t,a)$ in a(i) . Rather we look at each state in a(i) and then consult the inverse state table to find each t such that $\delta(t,a)$ is in a(i) . Each time a new t is found, the block containing t is located and t placed on a list of states to be split off from the block. The block is then placed on a list of blocks which have been refined if it is not already on the list. Finally we go down the list of blocks which have been refined and actually partition them. The number of blocks we must look at must be less than the number of state transitions on input a terminating on states in B(i) . The time necessary to actually partition a block is proportional to the number of states to be split off. When the number is summed over all blocks which are partitioned it adds up to the number of state transitions on input a terminating on states in B(i) .) Let k be the constant of proportionality.

Consider the time spent in the loop of Step 5 through 8 for a given input symbol a . Assume that at step 5 the blocks of the partition are B(1),B(2),...,B(m) and that $L(a) = \{i_1,i_2,\ldots,i_r\}$. Let $\{i_{r+1},i_{r+2},\ldots,i_m\} = \{1,2,\ldots,m\} - L(a)$. The claim is made that the total time spent on traversals of the loop for which input symbol a is selected in Step 5 until the program terminates is bounded by

$$T = k\left(\sum_{j=1}^{r} a_{i_j} \log a_{i_j} + \sum_{j=r+1}^{m} a_{i_j} \log(a_{i_j}/2)\right) .$$

Clearly the bound is valid if the algorithm has terminated. If the loop is traversed for an input symbol other than a , then the time spent is not included in T . However, since blocks get split and the set L(a) modified we must show that the new value of T , call it $\hat{T}$, is less than or equal to the old value of T . If a block whose index is in L(a) is partitioned, then a term of the form b log b is replaced by the expression c log c + (b−c) log(b−c) which decreases the value of T . If a block whose index is not in L(a) is partitioned, then a term of the form b log b/2 is replaced by the expression

$$c \log c + (b-c) \log(b-c)/2$$

where $c \le b-c$. Since $c \le b/2$ and $(b-c)/2 \le b/2$,

$$\begin{aligned} c \log c + (b-c)\log(b-c)/2 &\le c \log b/2 + (b-c)\log b/2 \\ &\le b \log b/2 . \end{aligned}$$

ither case $\hat{T}$ is less than T . Finally, assume $r \ne 0$ and a and some ℓ in L(a) has been selected at Step 5. As we showed earlier, the time around the loop is bounded by ka_ℓ . Thus by induction, the total time is bounded by

$$k[a_\ell + a_\ell \log(a_\ell/2) + \sum_{\substack{j=1\\ j\neq\ell}}^{r} a_{i_j} \log a_{i_j} + \sum_{j=r+1}^{m} a_{i_j} \log(a_{i_j}/2)] .$$

We must show that this expression is less than or equal to T . That is, we need show

$$a_\ell + a_\ell \log a_\ell/2 \le a_\ell \log a_\ell .$$

Clearly $a_\ell + a_\ell \log a_\ell/2 = a_\ell(\log a_\ell/2 + \log 2) = a_\ell \log a_\ell$. This completes the proof of the claim.

The first time Step 5 is executed the formula for T is bounded by knlog n . Multiply by the number of input symbols and adding in the time for Steps 1 through 4 yields a total bound proportional to n log n .

Experimental results and conclusions

In order to obtain timing information, the algorithm was applied to two classes of finite automata. Automata in the first class are given by $A(n) = (\{1,2,\ldots,n\},\{0,1\},\delta,\{1\})$ where $\delta(1,0) = \delta(1,1) = 1$ and $\delta(i,0) = i-1$ and $\delta(i,1) = i$ for $2 \le i \le n$. Automata in the second class are given (for even n) by $B(n) = (\{1,2,\ldots,n\}, \{0,1\},\delta, \{i|1 \le i \le n/2\})$ where $\delta(i,0) = \delta(i,1) = n/2 + 2i-1$ and $\delta(n/4+i,0) = \delta(n/4+i,1) = 2i-1$ for $1 \le i \le n/4$ and $\delta(n/2+i,0) = \delta(n/2+i,1) = 2i-1$ for $n/2 < i \le n$. The running times on an IBM 360/67 for the two classes are listed in Table 1.

n	A(n)	B(n)
100	$\frac{37}{60}$	$\frac{2}{3}$
1000	$5\frac{50}{60}$	$6\frac{3}{4}$
2003	$11\frac{38}{60}$	$13\frac{2}{3}$

Table 1. time in seconds

Note that A(n) is the example which required n^2 steps for previous algorithms.

Our algorithm is particularly suited for A(n) and a detailed analysis shows that the running time should grow linearly-with the number of states as the experimental evidence indicates. The worst case for our algorithm is typified by B(n) in which blocks are always partitioned equally. The running time for B(n) should grow as n log n for both the current algorithm and for previously published algorithms. The results seem to indicate that the algorithm is practical for minimizing states in finite automata (or testing equivalence of finite automata) of up to several thousand states.

References

1. Harrison, M. A., Introduction to Switching and Automata Theory, McGraw-Hill, New York, 1965.
2. McCluskey, E. J., Introduction to the Theory of Switching Circuits, McGraw-Hill, New York, 1965.

```
COMMENT*****************************************************************
*     ALGORITHM TO MINIMIZE STATES IN FINITE AUTOMATON
      *****************************************************************;
BEGIN
   INTEGER N;
   READ(N);
      BEGIN
         INTEGER ARRAY NEXTONZERO,NEXTONONE,PREVONZERO,PREVONONE(1::N);
         INTEGER ARRAY FINAL(1::N);
         INTEGER ARRAY DATA,NEXT(1::6*N);
         INTEGER ARRAY STATENEXT,STATELAST,ONENEXT,ONELAST,ZERONEXT,
            ZEROLAST,SPLIT(1::2*N);
         INTEGER ARRAY BLOCK,NUMINBLOCK,NUMINSPLIT(1::N);
         INTEGER ARRAY BLOCKSPLIT(1::N);
         INTEGER ARRAY NUMZERO,NUMONE(1::N);
         INTEGER ARRAY ONBLOCKLISTZERO,ONBLOCKLISTONE(1::N);
         INTEGER ARRAY NZEROTR,NONETR(1::N);
         INTEGER BLOCKLISTZERO,BLOCKLISTONE;
         INTEGER FREE,CELL;
         INTEGER LIST;
         INTEGER SPLITBLOCKS;
         INTEGER K,J,STATE,LASTSTATE,FREEBLOCK,BSPLIT;
         PROCEDURE ADD(INTEGER VALUE RESULT X;INTEGER VALUE Y);
            BEGIN
               INTEGER TEMP;
               TEMP:=FREE;
               FREE:=NEXT(FREE);
               NEXT(TEMP):=X;
               X:=TEMP;
               DATA(X):=Y;
            END;
         PROCEDURE REMOVEFROM(INTEGER VALUE RESULT X);
            BEGIN
               INTEGER TEMP;
               TEMP:=FREE;
               FREE:=X;
               X:=NEXT(X);
               NEXT(FREE):=TEMP;
            END;
         PROCEDURE INSERTSTATE(INTEGER VALUE X,Y);
            BEGIN
               STATENEXT(N+Y):=STATENEXT(X);
               STATELAST(N+Y):=X;
               IF (STATENEXT(X)¬=0) THEN
               STATELAST(STATENEXT(X)):=N+Y;
               STATENEXT(X):=N+Y;
            END;
         PROCEDURE INSERTZERO(INTEGER VALUE X,Y);
            BEGIN
               ZERONEXT(N+Y):=ZERONEXT(X);
               ZEROLAST(N+Y):=X;
               IF (ZERONEXT(X)¬=0) THEN
               ZEROLAST(ZERONEXT(X)):=N+Y;
               ZERONEXT(X):=N+Y;
            END;
         PROCEDURE INSERTONE(INTEGER VALUE X,Y);
            BEGIN
               ONENEXT(N+Y):=ONENEXT(X);
```

```
                    ONELAST(N+Y):=X;
                    IF (ONENEXT(X)¬=0) THEN
                    ONELAST(ONENEXT(X)):=N+Y;
                    ONENEXT(X):=N+Y;
                 END;
    PROCEDURE INITIALIZE;
    BEGIN
             COMMENT*************************************************************
             *      INITIALIZE
             ****************************************************************;
             FOR I:=1 UNTIL 2*N DO
                BEGIN
                   STATENEXT(I):=STATELAST(I):=0;
                   ONENEXT(I):=ONELAST(I):=0;
                   ZERONEXT(I):=ZEROLAST(I):=0;
                END;
             FOR I:=1 UNTIL N DO
                BEGIN
                   PREVONZERO(I):=PREVONONE(I):=0;
                   NUMZERO(I):=NUMONE(I):=0;
                   SPLIT(I):=BLOCKSPLIT(I):=0;
                   NUMINBLOCK(I):=NUMINSPLIT(I):=0;
                   ONBLOCKLISTZERO(I):=ONBLOCKLISTONE(I):=0;
                   NZEROTR(I):=NONETR(I):=0;
                END;
             FOR I:=1 UNTIL 6*N DO NEXT(I):=I+1;
             NEXT(6*N):=0;
             BLOCKLISTZERO:=BLOCKLISTONE:=0;
             FREE:=1;
             SPLITBLOCKS:=0;
             WRITE("STATE TABLE");
             WRITE("            STATE            INPUT 0        INPUT 1
     OUTPUT");
             FOR I:=1 UNTIL N DO
                BEGIN
                   READ(NEXTONZERO(I),NEXTONONE(I),FINAL(I));
                   WRITE(I,NEXTONZERO(I),NEXTONONE(I),FINAL(I));
                END;
             COMMENT*************************************************************
             *      INVERT STATE TABLE
             ****************************************************************;
             FOR I:=1 UNTIL N DO
                BEGIN
                   ADD(PREVONZERO(NEXTONZERO(I)),I);
                   NZEROTR(DATA(PREVONZERO(NEXTONZERO(I)))):=
                      NZEROTR(DATA(PREVONZERO(NEXTONZERO(I))))+1;
                   ADD(PREVONONE(NEXTONONE(I)),I);
                   NONETR(DATA(PREVONONE(NEXTONONE(I)))):=
                      NONETR(DATA(PREVONONE(NEXTONONE(I))))+1;
                END;
             COMMENT*************************************************************
             *      SET UP INITIAL BLOCKS
             ****************************************************************;
             FOR I:=1 UNTIL N DO
                BEGIN
                   IF FINAL(I)=1 THEN
                      BEGIN
                         INSERTSTATE(1,I);
```

```
              IF NEXTONZERO(I)¬=0 THEN
                BEGIN
                  INSERTZERO(1,I);
                  NUMZERO(1):=NUMZERO(1)+NZEROTR(I);
                END;
              IF NEXTONONE(I)¬=0 THEN
                BEGIN
                  INSERTONE(1,I);
                  NUMONE(1):=NUMONE(1)+NONETR(I);
                END;
              BLOCK(I):=1;
              NUMINBLOCK(1):=NUMINBLOCK(1)+1;
            END
          ELSE
            BEGIN
              INSERTSTATE(2,I);
              IF NEXTONZERO(I)¬=0 THEN
                BEGIN
                  INSERTZERO(2,I);
                  NUMZERO(2):=NUMZERO(2)+NZEROTR(I);
                END;
              IF NEXTONONE(I)¬=0 THEN
                BEGIN
                  INSERTONE(2,I);
                  NUMONE(2):=NUMONE(2)+NONETR(I);
                END;
              BLOCK(I):=2;
              NUMINBLOCK(2):=NUMINBLOCK(2)+1;
            END;
        END;
      IF (NUMZERO(1)>=0) AND (NUMZERO(1)<=NUMZERO(2)) THEN
        BEGIN
          ADD(BLOCKLISTZERO,1);
          ONBLOCKLISTZERO(1):=1;
        END
      ELSE
        BEGIN
          ADD(BLOCKLISTZERO,2);
          ONBLOCKLISTZERO(2):=2;
        END;
      IF (NUMONE(1)>=0) AND ( NUMONE(1)<=NUMONE(2) ) THEN
        BEGIN
          ADD(BLOCKLISTONE,1);
          ONBLOCKLISTONE(1):=1;
        END
      ELSE
        BEGIN
          ADD(BLOCKLISTONE,2);
          ONBLOCKLISTONE(2):=2;
        END;
      FREEBLOCK:=3;
END;
INITIALIZE;
      COMMENT*****************************************************
      *      SELECT BLOCK AND PARTITION ON IT
      ************************************************************;
REPEAT:  IF BLOCKLISTZERO¬=0 THEN
        BEGIN
```

```
                J:=CATA(BLCCKLISTZERO);
                REMCVEFRCM(BLCCKLISTZERO);
                CNBLCCKLISTZERC(J):=0;
                GC TO DIV ZERO;
             END
          ELSE IF BLCCKLISTCNE¬=0 THEN
             BECIN
                J:=CATA(BLCCKLISTCNE);
                REMCVEFRCM(BLCCKLISTCNE);
                CNBLCCKLISTCNE(J):=0;
                GCT ODIVCNE;
             END
          ELSF CC TCFINISHED;
CIVZEFC:  STATE:=ZERONEXT(J);
          WHILE STATE¬=0 DC
             BEGIN
                CELL:=PREVCNZERO(STATE-N);
                WHILECELL¬=0 C O
                   BEGIN
                      LASTSTATE:=CATA(CELL);
                      K:=BLCCK(LASTSTATE);
                      ACD(SPLIT(K),LASTSTATE);
                      IF "LCCKSPL IT(K)=0THENACD(SPLITBLCCKS,K);
                      NUMINSPLIT(K):=NUMINSPLIT(K)+1;
                      BLOCKSPLIT(K):=1;
                      CELL:=NEXT(CELL);
                   END;
                STATE:=ZERCNEXT(STATE);
             END;
          GCT C RETURN;
DIVCNE:   STATE:=CNENEXT(J);
          WHILESTATE¬=0 D O
             BEGIN
                CELL:=PREVCNONE(STATE-N);
                WHILECELL¬=0 C C
                   BECIN
                      LASTSTATE:=CATA(CELL);
                      K:=BLCCK(LASTSTATE);
                      ACC(SPLIT(K),LASTSTATE);
                      I  F BLCCKSPLIT(K)=0 THEN ACD(SPLITBLCCKS,K);
                      NUMINSPLIT(K):=NUMINSPLIT(K)+1;
                      BLCCKSPLIT(K):=1;
                      CELL:=NEXT(CELL);
                   END;
                STATE:=CNENEXT(STATE);
             END;
          CCMMENT**********************************************************
          *     REFINE BLCCKS
          ****************************************************************;
RETURN:   I F(SPLITBLCCKS=0)THEN GOT  CREPEAT;
          BSPLIT:=CATA(SPLITBLCCKS);
          REMCVEFRCM(SPLITBLCCKS);
          BLCCKSPLIT(BSPLIT):=C;
          I  F NUMINBLOCK(BSPLIT)=NUMINSPLIT(BSPLIT) THEN
             BECIN
                NUMINSPLIT(BSPLIT):=C;
                WHILE (SPLIT(BSPLIT)¬=0) CC REMOVEFROM(SPLIT(BSPLIT));
                GCT ORETURN;
          END;
      NUMINBLOCK(BSPLIT):=NUMINBLOCK(BSPLIT)-NUMINSPLIT(BSPLIT);
      NUMINBLOCK(FREEBLOCK):=NUMINSPLIT(BSPLIT);
      NUMINSPLIT(BSPLIT):=0;
      WHILE SPLIT(BSPLIT)¬=0 DO
          BEGIN
              STATE:=DATA(SPLIT(BSPLIT));
              REMOVEFROM(SPLIT(BSPLIT));
              STATENEXT(STATELAST(N+STATE)):=STATENEXT(N+STATE);
              IF STATENEXT(N+STATE)¬=0 THEN
              STATELAST(STATENEXT(N+STATE)):=STATELAST(N+STATE);
              INSERTSTATE(FREEBLOCK,STATE);
              BLOCK(STATE):=FREEBLOCK;
      IF PREVONZERO(STATE)¬=0 THEN
          BEGIN
              ZERONEXT(ZEROLAST(N+STATE)):=ZERONEXT(N+STATE);
              IF ZERONEXT(N+STATE)¬=0 THEN
              ZEROLAST(ZERONEXT(N+STATE)):=ZEROLAST(N+STATE);
              INSERTZERO(FREEBLOCK,STATE);
              NUMZERO(BSPLIT):=NUMZERO(BSPLIT)-NZEROTR(STATE);
              NUMZERO(FREEBLOCK):=NUMZERO(FREEBLOCK)+NZEROTR(STATE);
          END;
      IF PREVONONE(STATE)¬=0 THEN
          BEGIN
              ONENEXT(ONELAST(N+STATE)):=ONENEXT(N+STATE);
              IF ONENEXT(N+STATE)¬=0 THEN
              ONELAST(ONENEXT(N+STATE)):=ONELAST(N+STATE);
              INSERTONE(FREEBLOCK,STATE);
              NUMONE(BSPLIT):=NUMONE(BSPLIT)-NONETR(STATE);
              NUMONE(FREEBLOCK):=NUMONE(FREEBLOCK)+ NONETR(STATE);
          END;
          END;
      COMMENT**********************************************************
      *     ADD NEW BLOCK TO BLOCKLIST
      ****************************************************************;
      IF (ONBLOCKLISTZERO(BSPLIT)¬=1) AND (NUMZERO(BSPLIT)>0) AND
          (NUMZERO(BSPLIT)<=NUMZERO(FREEBLOCK)) THEN
          BEGIN
              ADD(BLOCKLISTZERO,BSPLIT);
              ONBLOCKLISTZERO(BSPLIT):=1;
          END
      ELSE
          BEGIN
              ADD(BLOCKLISTZERO,FREEBLOCK);
              ONBLOCKLISTZERO(FREEBLOCK):=1;
          END;
      IF (ONBLOCKLISTONE(BSPLIT)¬=1) AND (NUMONE(BSPLIT)>0) AND
          (NUMONE(BSPLIT)<=NUMONE(FREEBLOCK)) THEN
          BEGIN
              ADD(BLOCKLISTONE,BSPLIT);
              ONBLOCKLISTONE(BSPLIT):=1;
          END
      ELSE
          BEGIN
              ADD(BLOCKLISTONE,FREEBLOCK);
              ONBLOCKLISTONE(FREEBLOCK):=1;
          END;
      FREEBLOCK:=FREEBLOCK+1;
```

```
C291 --              GC TC RETURN;
0292 - -             CCMMENT***************************************************************
0293 --              *     CUTPUT
C294 - -             ****************************************************************;
0245 --         FINISHEC:FCR I:=1 UNTIL N DC
0296 3-                  BECIN
C297 - -                    CELL:=STATENEXT(I);
C298 - -                    IF CELL=0 THEN GC TC EXIT;
C299 - -                    WRITE("BLCCK",I);
03CC - -                    WRITE(CELL-N);
C3C1 - -                    CELL:=STATENEXT(CELL);
03C2 - -                    WHILE CELL¬=0 DO
03C3 4 -                       BEGIN
0304 --                           WRITECN(CELL-N);
03C5 - -                          CELL:=STATENEXT(CELL);
0306 - 4                       ENC;
C3C7 - 3                 ENC;
03C8 - 2        EXIT: END;
C309 - 1        ENC.

002.83 SECCNCS IN CCMPILATICN, 10832 BYTES CF C C D E GENERATEC
```