# Solving Backtracking Problems with Structure Diagrams

E.S.H. Bulman
D.D. Cowan
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Technical Report CS-85-40
September 1985

# Solving Backtracking Problems with Structure Diagrams

E.S.H. Bulman and D.D. Cowan

## INTRODUCTION

In his book Principles of Program Design, Jackson (1) devotes a whole chapter to backtracking. This is a technique that has to be used in cases where the only way to proceed is by trial and error. He looks at several problems, and in each case draws a structure diagram which he uses to write the program. He identifies three stages in his development of a solution:

1. Trial - picking a given path to start with.

2. Error - making sure that a dead end is recognized.

3. Consideration of the side effects created by taking a path with a dead end. There are three kinds of side effect:

   a. Those that MUST be undone.
   b. Those that need not be undone.
   c. Those that should NOT be undone.

As far as programming stage 3 is concerned, he discusses the side effects, and why they should or should not be undone, and then with much reasoning, describes how his program can be modified to get out of its impasse.

Top-down design assumes that modules exist to do everything that needs to be done, and this allows a programmer to divide his problem into a series of smaller ones for which he assumes (for the time being) a module exists. This division process is continued until the lowest undefined modules consist of only a few simple operations. The contents of all these modules can be derived from data directed design. The solution to Jackson's stage 3 can be found by drawing a structure diagram (2), and this diagram will identity all the information that must be available to undo what has been done. The output structure will be the situation that existed at the point that the program started down the dead end path.

---

(1) Jackson, M.A., Principles of Program Design, 1975, Academic Press.

(2) For a description of structure diagrams, see Cowan, D.D., Graham, J.W., Welch, J.W. and Lucena, C.J.P., A Data-directed Approach to Program Construction. Software Practice and Experience, vol. 10, 355-372 (1980), also appendix A, which explains certain modifications which we have used in the figures in this report.

## EXAMPLES

We shall examine three problems: the Eight Queens, the Knight's Tour and the updating of a direct access file in real time. As will be apparent, all three involve backtracking, and we shall attempt to find one structure diagram that can be used for all of them. We shall examine the first problem in some detail in order to see the development of the solution. In the case of the last two, less detail will be necessary, since the development of their solutions is very similar to that of the first.

### The Eight Queens

To start with, we shall consider the problem of the Eight Queens and show first how the main program is developed, assuming that a backtracking algorithm exists. Then we shall develop this algorithm, again by using a structure diagram.

Eight queens are to be placed on a chess board in such a way that none of them are in check (being in check is defined as one queen being on the same row, column or diagonal as another). Most programs for solving this problem look for all possible solutions. Since we are merely demonstrating a technique for handling backtracking, we shall stop when we have found one solution. The algorithm we shall use is very simple. The first queen will be placed in the first row of the first column. Then for each of the following columns, a queen will be placed on the first uncovered square (uncovered meaning that no queen is already checking that square). When no uncovered square exists for a particular column, we must back up one column, and move its queen to the next uncovered square, and continue from that point as though the previous path had never been tried. Since there may be no uncovered squares remaining in a previous column, it may be necessary to back up more than one column. The input and output structures for this algorithm are shown in Figure 1.

Like the algorithm, the structure diagram for the main program is quite simple. It should be noted that although there are structures for columns and rows, there is nothing for diagonals. It is clear that given a column and a row, we have identified a square which must be associated with exactly two diagonals. We shall in fact keep a table which will identify the two diagonals passing through each square (even if one of them is only of length 1). If a square is available, that means first of all that no queen is on the same row of another column, and next that no queen is on one of the diagonals that passes through that square. It is clear that the output structure is identical, even though the names of some of the structures differ.

The two structures LAST ROW and NOT LAST ROW need a little explanation. Because we can only work by trial and error, if we cannot place a queen on a particular row, then we must try the next one, if there is one. If there is not, that means that we have tried the last row (unsuccessfully) and therefore shall have to back up. This is an action that is an exception to the normal running of the program, and therefore must be represented by an "exception" structure. This is why LAST ROW has to be distinguished from NOT LAST ROW. Writing the program is now very simple, and follows the complete diagram in Figure 2. For every column, one of three things can happen. If there is a row uncovered, the queen is placed on it. If a row we try is covered, but is not the last one, the next one must be tried. Finally, if the row that is causing the problem is the last one of the column, it will be necessary to back up. PLACE QUEEN takes care of the uncovered row situation; the control point B creates a loop which looks after moving from one row to the next; and the black box BACK UP handles the last possibility. We shall now look into this.

We shall use a structure diagram for this purpose and it appears in Figure 3 (3). It should be stressed that even though they are not the same, both this diagram and the one in Figure 1 represent the chess board and the position of the queens. There are other similarities in the two diagrams. In both cases, the board is represented by the structures COLUMN and ROW. Furthermore, in both we are interested in an available row. At the same time, there are important differences. In the main diagram, we need to be able to examine all the rows of a column, whereas in BACK UP, only two are of interest: the one where the queen is, and the next untried row. Moreover, since the actual trial of this second row will take place in the main diagram, that is to say, when BACK UP returns, there is no need to distinguish a covered row from an available one. The structure TABLE OF DIAGONALS COVERED that precedes COLUMN may seem a little odd, however it should not be forgotten that a table created to solve a problem may well be input data for a sub-routine.

When it is necessary to back up, it is because there is no square available in the current column, and therefore the configuration established up to the present column must be changed. This means that the queen in the previous column will have to be moved. To do this, first of all we must go back to that column. This is achieved by decrementing the column counter. Then the queen must be removed so that it can be placed on the next available row. The removal itself obviously belongs to the structure ROW WITH QUEEN. If there is still an untried row in the column, it must be tried. Setting ROW to the value of one more than that of ROW WITH QUEEN and returning to the main program will take care of this. At that time, things will continue as though that point had been naturally from the beginning. As we mentioned earlier, if there are no more

---

(3) There should be diagrams for both PLACE QUEEN and BACK UP, however since the only one we are interested in for generalization purposes is BACK UP, we shall assume that PLACE QUEEN is an operation that has already been properly defined.

untried rows in the column to which we have backed up, it will be necessary to back up another column. This may have to continue all the way back to the first column. What we have here is a recursive action, but one that we can implement as a loop (the structure COLUMN). The control for this loop is: FOR EACH COLUMN (GOING BACKWARDS) UNTIL A NEW ROW IN THE COLUMN CAN BE TRIED. The complete diagram for BACK UP is to be found in Figure 4 (4).

## The Knight's Tour

In the Knight's Tour, the knight is placed in an arbitrary position on the chess board and has to visit all the squares once only (5). Again, we use a simple algorithm. When the knight is placed on any square, a descriptor of that square is modified to show that it is occupied. If the knight is placed on a square whose successors are all occupied, it has found a dead end and must try another path. The easiest thing to do is to go back to its predecessor, and try that square's next successor. If that predecessor has no more untried successors, then the knight must go back another move and try the next successor of its predecessor's predecessor; and so on.

The complete structure diagram appears in Figure 5. A few comments will be helpful. The input structure includes a "successor table" which is used to create a table of square descriptors (6). Each descriptor contains, in addition to the number of successors and their coordinates, an indicator showing whether the square is already occupied, the identity of its predecessor and the identity of the last successor tried, i.e. where the knight was before it moved from this square. Like the table of diagonals covered in the Eight Queens, the descriptor table itself is not input to the main program, neither is it output, however it is used in the operation MOVE KNIGHT and again in BACK UP. (We have only drawn the structure diagram for BACK UP, and in this, the descriptor table appears as input under the name of STATE.) The other input data describes the chess board itself. Since the problem again deals with a chess board, the basic diagram resembles that of the Eight Queens problem, in that the board is again divided into squares (although not in the same way). Another resemblance is the fact that the availability of a square is important and therefore it is necessary to distinguish between avalailable and unavailable successors. The

---

(4) In all future examples, we shall not show a separate input and output diagram since these structures appear clearly in the complete diagram.

(5) One version of this problem requires the knight to finish on the square that it started from.

(6) It would have been possible to formulate an algorithm to calculate the successors of each square, and this would have made it possible to write a general program for chequer boards of an arbitrary size. We used a table in order to simplify programming.

output structure is exactly the same as the input one except for the successor table which is not changed (it is only used to create the descriptor table). When the actions are added to complete the diagram, they are added to the structures which correspond to those in the Eight Queens diagram. Where PLACE QUEEN was included, in the Knight's Tour diagram we find MOVE KNIGHT, and in the same way that BACK UP was only used when the last row was occupied (COVERED) in the Eight Queens, the same operation only appears in the Knight's Tour when there are no more successors available.

When we come to the diagram for the operation BACK UP, we see that there is an interesting difference. In problems involving back up, information has to be kept about the point at which the backing up will stop. Furthermore, the effects of moves made after that point, but which led to the cul de sac, have to be undone. Only when this has been done, can a new path be started from the point which the backing up has reached. In the case of the Eight Queens, because moves were controlled by loops, the appropriate changes could be made by decrementing the column counter or incrementing the row counter; in addition, the special table maintained to look after diagonals had to be corrected. In the Knight's Tour, because all information about successors is kept in the descriptor table, this is the only structure that can be used to make the proper adjustments. The complete diagram for BACK UP appears in Figure 6. The relevant information of the descriptor of the square appears in the structure STATE.

## Updating a File in Real Time from Multiple Transactions

In the final example, we are asked by a business to write a program to process transactions affecting a master file organized to allow direct access. There are four kinds of transaction: open a new account; send an invoice; receive a payment; close an account. Any given customer makes zero or more transactions during the day, and they are to be processed immediately. Obviously, certain transactions are illogical. For example, if a customer already has an account numbered 2222, he cannot open a new account, number 2222 (there is, of course nothing to prevent him from opening another account with a different number). There are several kinds of illogical transaction and the business has asked us not to process ANY transactions for an account for which there is at least one illogical one. Since transactions have to be processed in real time, there is a paradox. Several seemingly good transactions may be processed before the first illogical one occurs, and this means that the ones that were processed ought not to have been. When the illogical transaction is discovered, all the transactions for that customer that have already been processed that day will have to be removed. It will be necessary once again to provide for backing up over those transactions which had been thought to be good. There are two ways of looking at this.

The master file can be thought of as containing a number of "super records", each consisting of a group of transactions and a master part (which may be empty to start with i.e. when a new account is opened). These records are either good or bad. Jackson (7) uses a "posit" construction, which supposes that a record is good until it is proven bad. When that occurs, control "quits" that part of the program that processes good records, and is "admitted" to another part that processes bad ones.

The second way of looking at the problem is in the same manner that we looked at the queens and the knight. We advance along a given path until we are stopped and then we back up to a point that we recognize as being acceptable. Instead of having a process driven by records, we shall use a transaction driven one. All transactions are either good or bad, and will be treated accordingly. The structure diagram for this is to be found at Figure 7.

To back up, it is necessary to put the record in the master file back to the state it was in before any transactions were received for it at all. To do this, a record must be kept of all transactions so that if necessary (and this may not be known until the end of the day) they can be undone. An easier method is to make a copy of the master record before the first transaction is processed. If it is necessary to back up, the current master record can be replaced by the copy of the original. We have used the second method, however as we shall see, the first method conforms to what might be called the "standard diagram" for backtracking. The diagram for the BACK UP we have used is in Figure 8.

## ANALYSIS

In all three of the above examples, the essential part of the main diagram can be modified so that one structure will serve for all three. This structure is shown in Figure 9. There are really only two kinds of transaction, acceptable ones and unacceptable ones. It is the latter that force us to back up. The control points A and B describe what is acceptable and what is not:

```
Eight Queens
     A:  IF ROW AVAILABLE
     B:  IF LAST ROW AND NOT AVAILABLE

Knight's Tour
     A:  IF NEXT SUCCESSOR AVAILABLE
     B:  IF NEXT SUCCESSOR IS LAST AND NOT AVAILABLE

File Update
     A:  IF GOOD TRANSACTION
     B:  IF BAD TRANSACTION
```

---

(7) Ibid.

It should be noted that in the case of the file update, because all bad transactions (for a particular customer) after the first are handled in the main diagram, Figures 7 and 9 do not resemble each other as closely as they could have. Two types of unacceptable transactions have been distinguished: the first (for a customer) and all the others (for the same customer). If the treatment of the latter had been included in BACK UP, the diagrams would have been similar. As we shall see, this would have created a difference in the generalized BACK UP diagram. The problem is caused by the fact that even though we have backed up, we must look at any other transactions that may occur subsequently for the same customer.

## Backing up

In all three problems, the action of backing up implies two things: first, finding the point where the back up can stop and a new path tried; second, removing all trace of the moves from that point which led to the cul de sac. In the case of the Eight Queens, modifying the loop counters locates the square to which we want to back up. It also removes some of the traces of the "bad" moves, in that resetting the column and row counters automatically eliminates moves made past the the back up point. Changing the diagonal table has to be done explicitly. In the Knight's Tour, there is no automatic adjustment. Every time a back up is carried out, it is necessary to change the descriptor of the square or squares on the path leading to the cul de sac. Actually, in both cases, for each square backed up, what was done was simply the opposite of what was done when the original move was made. This is illustrated more clearly in the third example.

In any backtracking problem, it is possible to describe the point or the situation to which the back-up must return. The logically correct way to find this point is to undo all the previous transactions in the reverse order. In the file processing problem, if we keep a record of all the transactions, we can do this. However, since we know this point ahead of time (it is always the master record at the beginning of the day), we can make a copy just before the first transaction for that record is processed, and use that as a short cut. In spite of apparent differences, the basic structure of the three problems is the same. We only use a copy of the opening state as an optimization. Figure 10 shows clearly the essential details of the generalized BACK UP diagram. It must be stressed that although this diagram serves as a foundation, other structures may be added, depending on the circumstances. For example, it is probable that in the file updating problem, a report would be produced, showing all transactions affecting a "bad" record. In fact, this is what we did in Figures 7 and 8. The only changes required to Figure 10 would be the addition of output structures REPORT and DETAIL corresponding to RECORD OF ACTIVITY and TRANSACTION respectively. There would also be an operation PRINT DETAILS associated with the structure TRANSACTION.

If we had decided to handle all transactions after a bad one in BACK UP, the generalized diagram of Figure 10 would have had to be modified. It would have been necessary to distinguish those transactions (if any) which were processed before the first bad one, from those that might arrive after. It is clear that this distinction has to be made in one of the diagrams, but it not so clear which one should be chosen.

## CONCLUSION

We have presented three different problems and developed one main structure diagram and one subsidiary diagram that can serve for all three. In many ways, the Eight Queens and the Knight's Tour are the same problem, however there is a subtle difference. In the first case, the Queen could not be moved until its target square was verified, whereas in the Knight's Tour, the Knight was moved and only then was the successor square checked. The file update problem showed how a sufficient history might have to be maintained in order to undo the previous operations in the reverse order (even though it is possible to short circuit that by using a copy of the starting point).

It is suggested that the two generalized diagrams can be used for all types of backtracking problems. This simplifies the work of the programmer, and enables him to concentrate on the question of what information must be kept in order to undo those operations that led nowhere.

We have deliberately not tackled the question of side effects that should not be undone. This is because these usually have to do with optimization of subsequent operations. Since we are trying to develop a generalized method that will make programming easier for less experienced users, simplicity of the method itself is the prime consideration. Only when a simple, clear program that works has be written should optimization be considered. Even then, this should only be done if the circumstances warrant it.

EIGHT QUEEN'S - INPUT AND OUTPUT

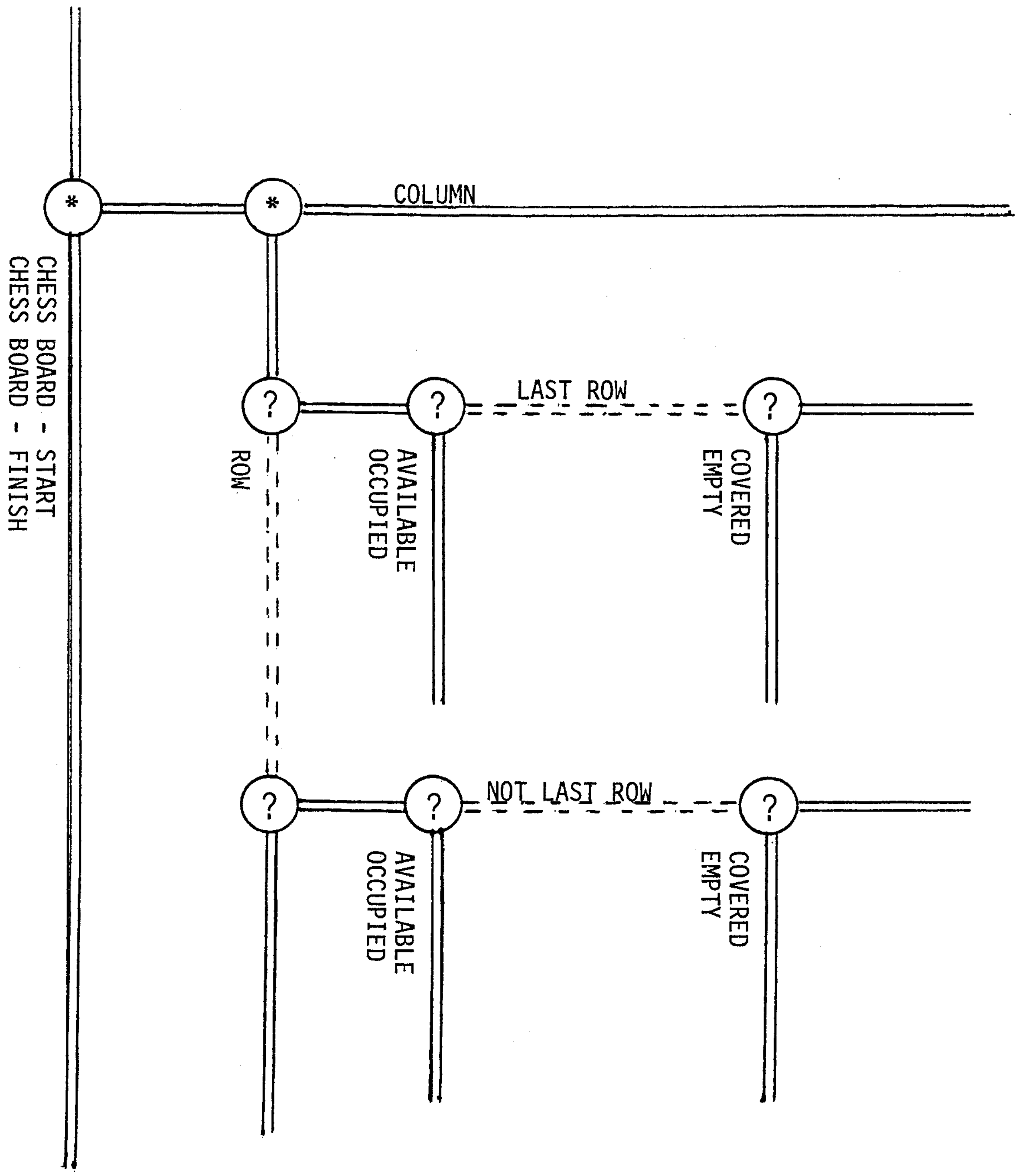
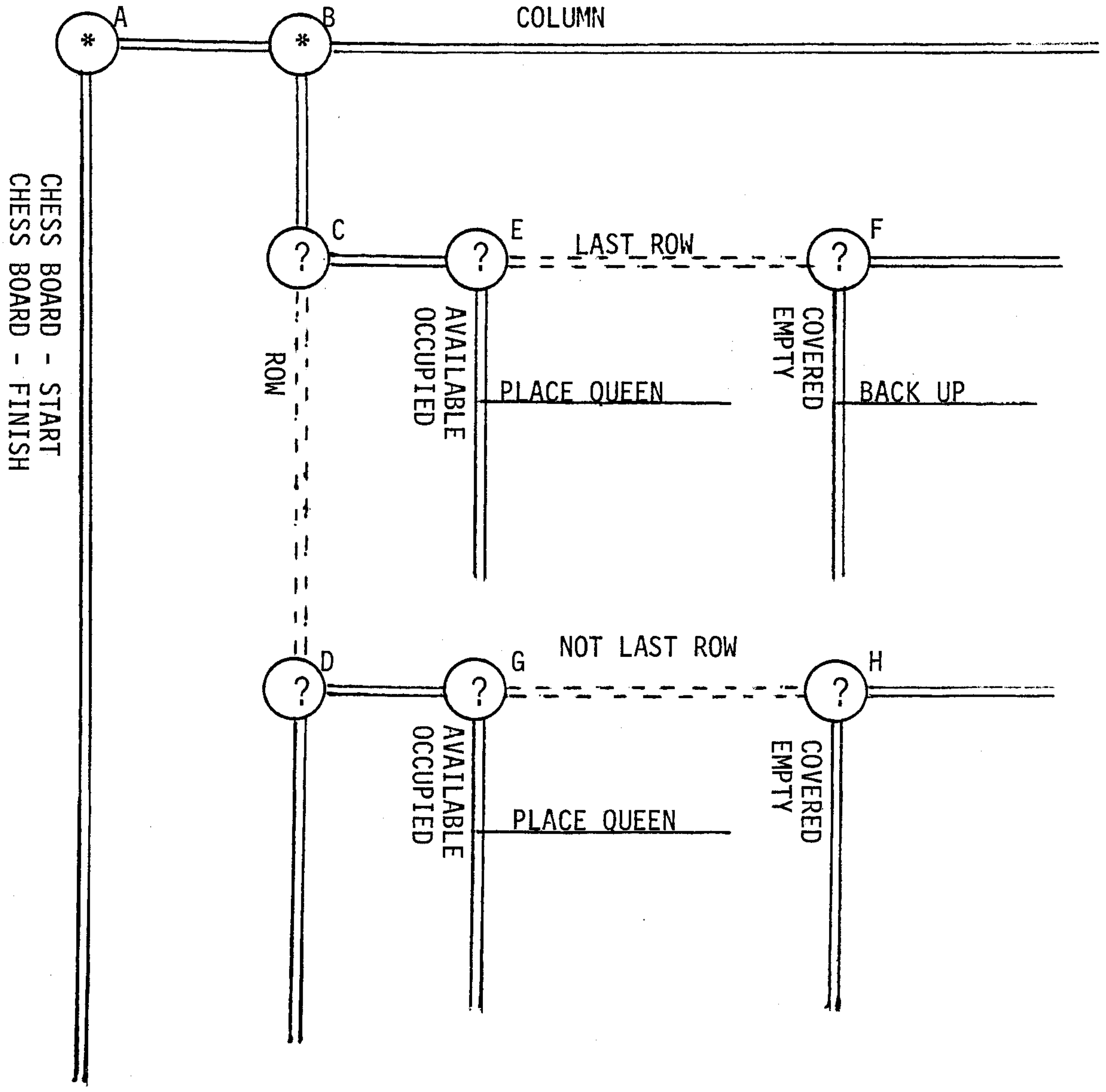


FIGURE 1

EIGHT QUEEN'S - COMPLETE



A: FOR EACH COLUMN
B: FOR EACH ROW
C: IF LAST ROW
D: IF NOT LAST ROW
E: IF SQUARE NOT COVERED
F: IF SQUARE COVERED
G: IF SQUARE NOT COVERED
H: IF SQUARE COVERED

FIGURE 2

# EIGHT QUEEN'S: BACK UP - INPUT AND OUTPUT

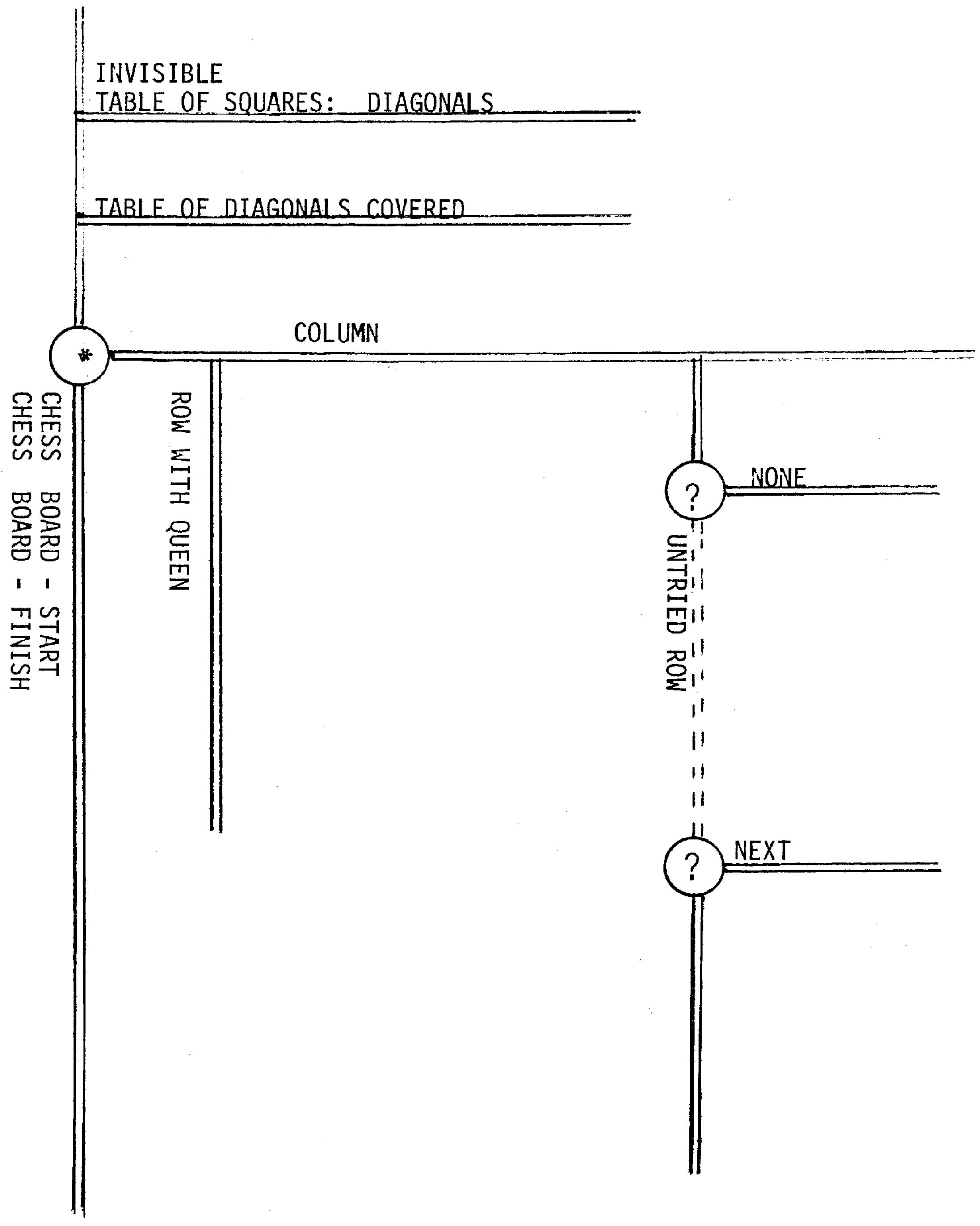
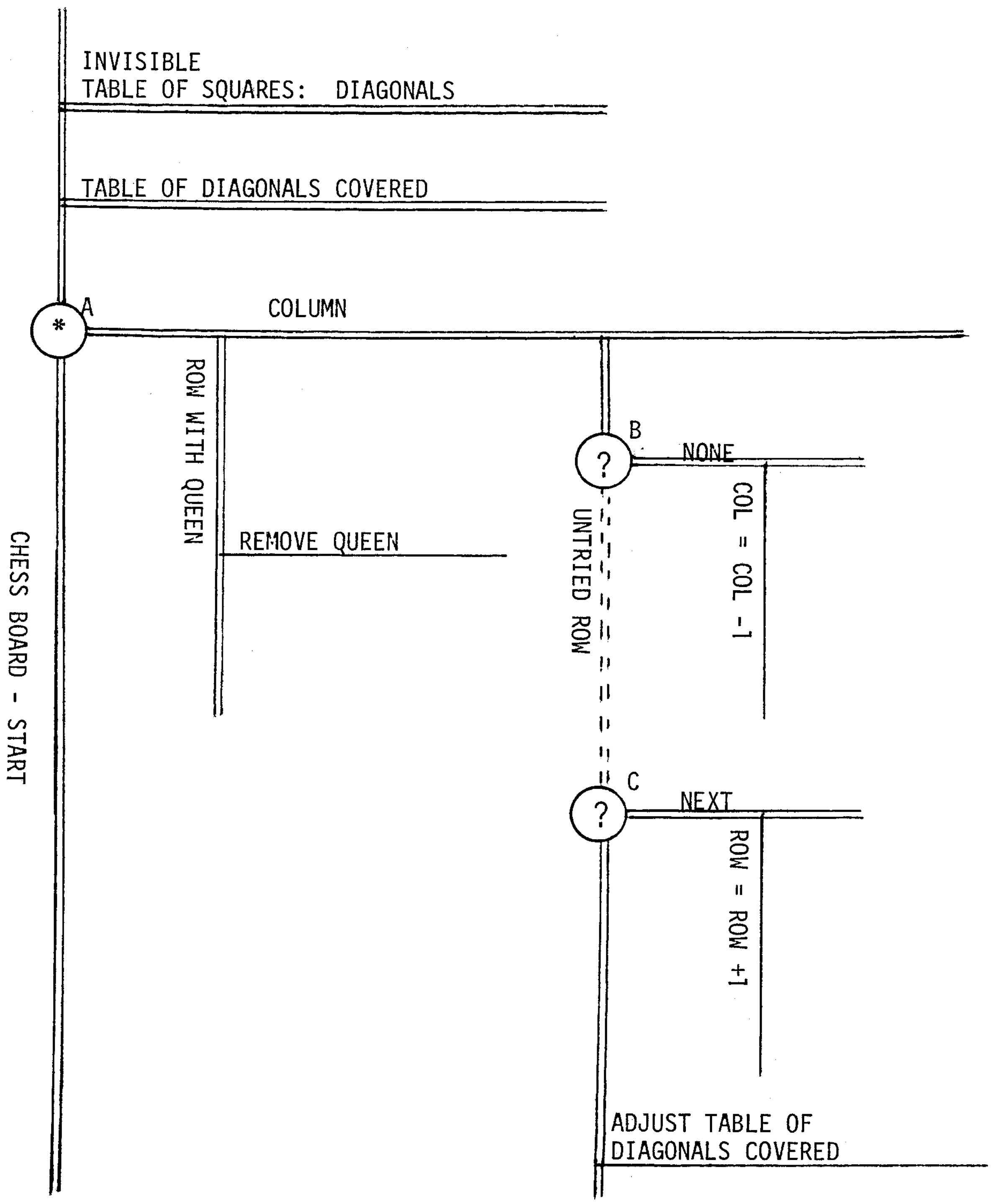


FIGURE 3

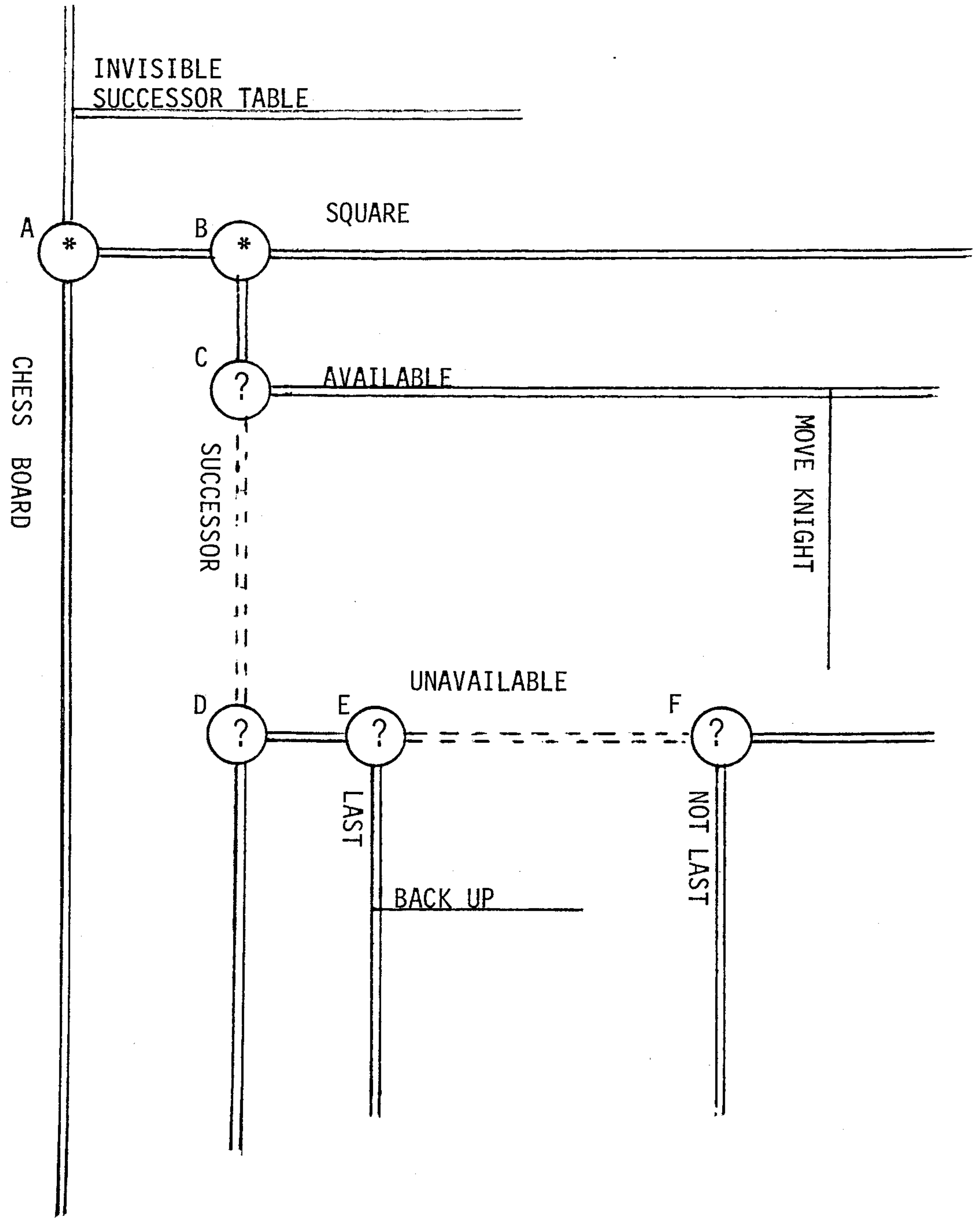
EIGHT QUEEN'S: BACK UP-COMPLETE



A: FOR EACH COLUMN (GOING BACKWARDS) UNTIL
A NEW ROW IN THE COLUMN CAN BE TRIED
B: IF NO UNTRIED ROWS
C: IF AT LEAST ONE UNTRIED ROW

FIGURE 4

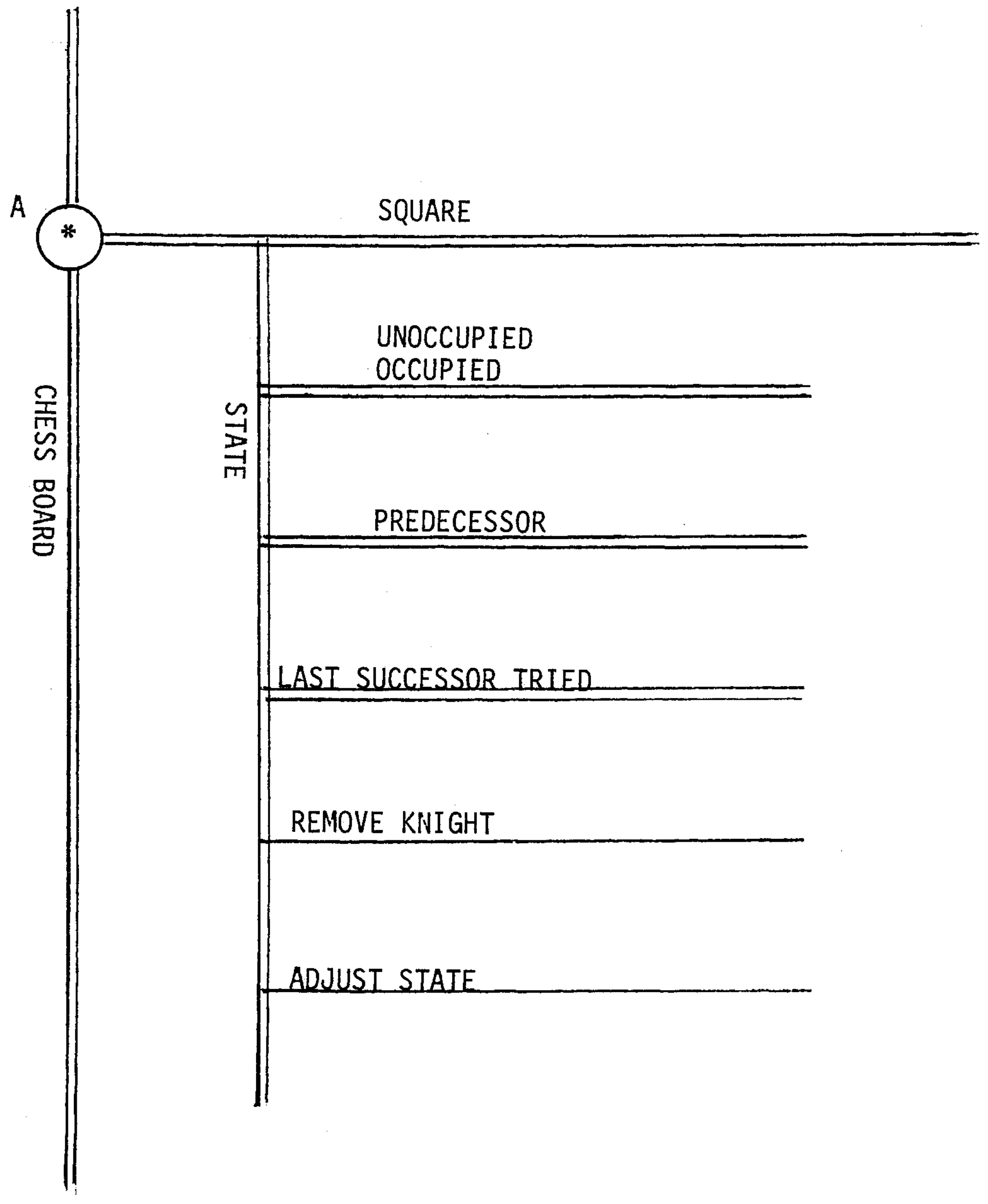
KNIGHT'S TOUR



A: FOR EACH SQUARE IN THE TOUR
B: FOR EACH SUCCESSOR
C: IF SUCCESSOR AVAILABLE
D: IF SUCCESSOR NOT AVAILABLE
E: IF LAST SUCCESSOR
F: IF NOT LAST SUCCESSOR

FIGURE 5

KNIGHT'S TOUR: BACK UP



A: FOR EACH SQUARE ( GOING BACKWARDS) UNTIL
A NEW SUCCESSOR CAN BE TRIED

FIGURE 6

FILE UPDATE - REAL TIME

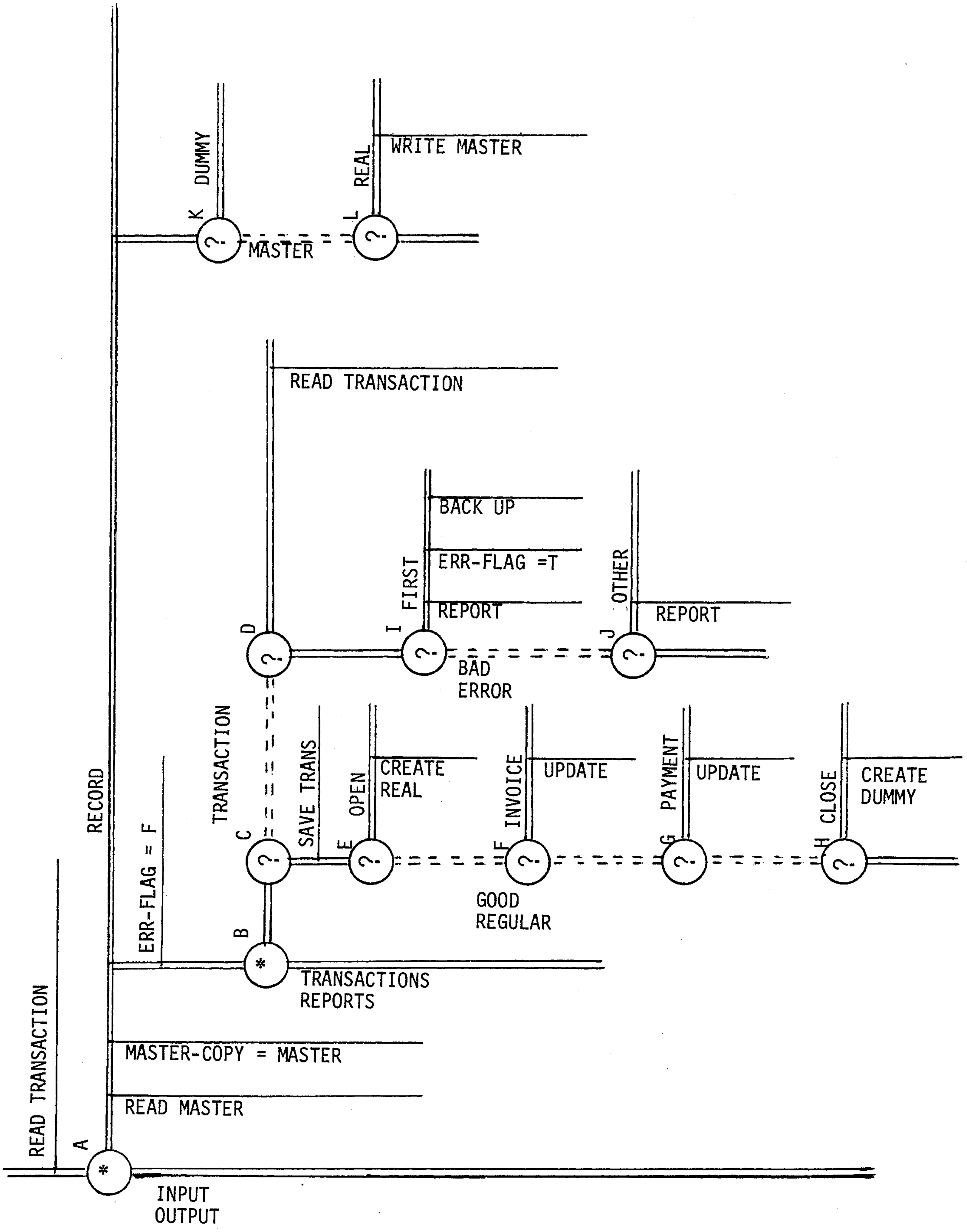


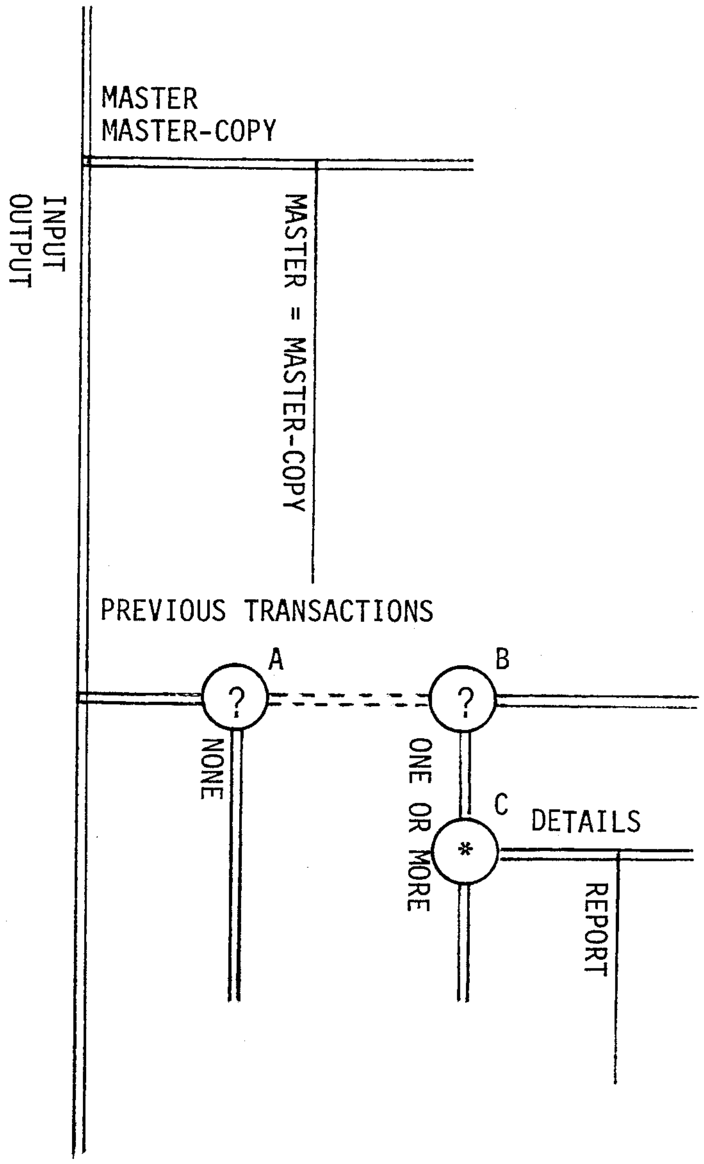
FIGURE 7A

## FILE UPDATE - REAL TIME
## CONTROL POINTS

```
A:  WHILE ID-TRANS ≠ 999
B:  WHILE ID-TRANS = ID-MASTER
C:  IF (MASTER = REAL AND TRANS-TYPE = (INVOICE, PAYMENT OR CLOSE))  OR
       (MASTER = DUMMY AND TRANS-TYPE = OPEN)
D:  NOT C
E:  IF TRANS-TYPE = OPEN
F:  IF TRANS-TYPE = INVOICE
G:  IF TRANS-TYPE = PAYMENT
H:  IF TRANS-TYPE = CLOSE
I:  IF ERR-FLAG = F
J:  IF ERR-FLAG = T
K:  IF MASTER = DUMMY
L:  IF MASTER = REAL
```

FIGURE 7B

FILE UPDATE - REAL TIME:  BACK UP



A:  IF NO PREVIOUS TRANSACTIONS
B:  IF PREVIOUS TRANSACTIONS
C:  FOR EACH PREVIOUS TRANSACTION

FIGURE 8

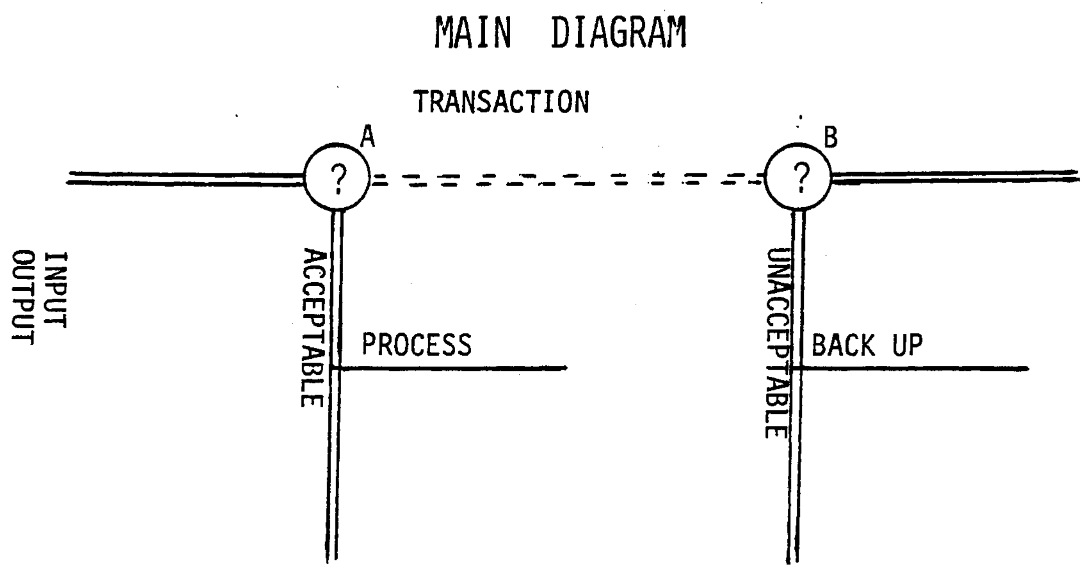
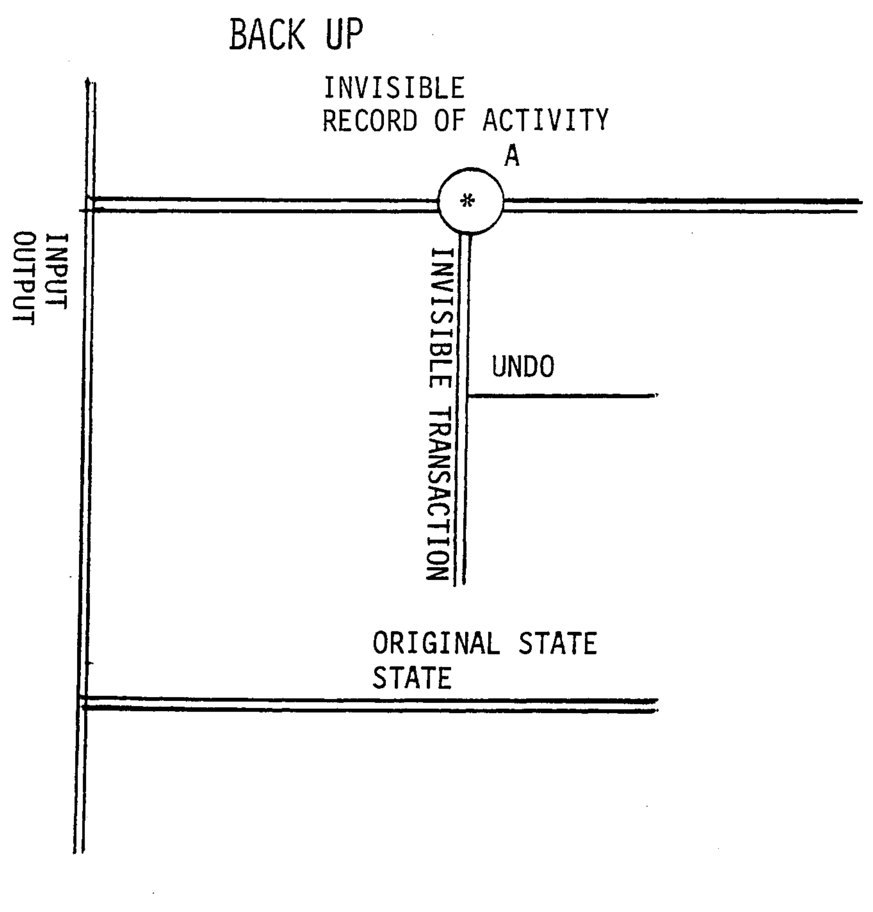


FIGURE 9



A: WHILE STATE UNACCEPTABLE

FIGURE 10

# APPENDIX I

## STRUCTURE DIAGRAMS

Cowan et al. (1) use three types of control point symbol: a circle containing an asterisk (repetition), a circle containing a question mark (alternative) and a circle containing a semi-colon (element of a tuple). In the last case, we merely show the elements of a tuple as straight lines at right angles to the parent structure.

Usually, three diagrams are drawn: the structure of the input, the structure of the output and the programming instructions (operations). We only use two types of diagram: one that combines input and output structures, and a complete diagram i.e. programming instructions superimposed on an input output diagram. In order to differentiate between input and output structures and operations, the former are drawn with a double line, and the latter with a single line. The name of a horizontal input structure appears directly above it, and that of a vertical one, immediately to its left. If the name of the output structure is the same as that of the input name, it is not shown, but if it is different, it appears above the "horizontal" name or to the left of the "vertical" one. In cases where there is no output or input structure corresponding to an input or output structure, an "invisible" structure is drawn and carries the name INVISIBLE. The names of operations appear either above or to the left of them.

---

(1) Ibid.