Non-Deterministic Algorithms*

JACQUES COHEN

Physics Department, Brandeis University, Waltham, Massachusetts 02154

Primitive commands representing the concepts of choice, failure, and success are used to describe non-deterministic algorithms for solving a variety of problems. First, the role of the primitives is explained in a manner appealing to the reader's intuition. Then, a solution to the classical 8-queens problem is presented as a non-deterministic program, and its implementation is described. Two examples follow, showing the usefulness of the primitives in computer-aided problem solving: the first is a simple question-answering program, the other is a parser for a context-sensitive language. Finally, a brief survey of current and related work is presented which includes: additional desirable primitives, implementation, correctness, efficiency, and theoretical implications

Keywords and Phrases: backtracking, language constructs, implementation, interpreters, problem solving, efficiency.

CR Categories: 3.64, 4.13, 4.22, 5.23

INTRODUCTION

A common connotation for the word *deterministic* is the *absence of choice.* In a deterministic event one knows precisely what will happen under each set of antecedent circumstances. In contrast, the term *non-deterministic* is used to denote an event whose outcome depends on the choices made by its participants.

The term *non-deterministic* was first used in computer science in connection with automata theory (see, for example, HOPC69). In that field, the term refers to a certain type of abstract machine. A non-deterministic machine has to obey a set of instructions which is ambiguous in the sense that two or more instructions may be applicable at some states of the execution.

Floyd has used the expression *non-deterministic algorithms* [FLOY67] to denote those which contain special commands, obeyed by choosing one parameter among several.

In some cases, non-deterministic algorithms have equivalent deterministic, and often more efficient, counterparts. Nevertheless, non-deterministic algorithms are often more succinctly and clearly described, and can always be *simulated* on a deterministic machine such as a digital computer. The translation of a non-deterministic algorithm into the program which simulates it can be performed automatically. These features parallel the advantages of recursive algorithms, which can be translated automatically into their equivalent, but perhaps less succinct, non-recursive counterparts.

Non-deterministic algorithms have also been referred to as "backtrack programs." We shall see that backtrack programs are actually the result of translating non-deterministic algorithms so that they can be run

* This work was supported by the National Science Foundation under Grant Number MCS-74-24569-A01.

CONTENTS

in a computer. The word "backtracking" was first used in this context by Walker [WALK60] in 1960.

The first reported simulation of non-deterministic algorithms was done by Golomb and Baumert in 1965 [GOLO65]. They proposed a useful technique for searching for specific configuration vectors in multidimensional spaces. Two years later Floyd [FLOY67] proposed a number of language features which would enable programmers to achieve succinctness and clarity in describing combinatorial algorithms.

More recently, the underlying concepts behind non-deterministic algorithms have become important in fields as varied as artificial intelligence and complexity theory.

Although numerous papers have been published on the various facets of non-determinism, there has been no tutorial presentation summarizing the work done in this area. This paper attempts to fill this gap, while emphasizing the aspects of non-determinism related to language constructs, implementation, and problem solving. Reference will also be made to the importance of non-determinism in classifying algorithms according to their efficiency.

The paper is directed, not to the expert, but to the reader who would like to become acquainted with the basic notions of non-determinism, as presently used in computer science.

Only an elementary knowledge of programming is necessary for reading the following two sections. A reader with a good programming background should have no difficulty understanding the remaining sections. (The example in Section 5 may be skipped on a first reading.) The references in Section 6, although not exhaustive, should provide directions for further reading.

1. AN INTUITIVE EXAMPLE

The following event, which actually took place during the exhibition of an avant-garde movie, illustrates particularly well the non-deterministic situations which interest us. The spectators have gathered in a hall to watch "a non-deterministic movie." The projection starts, and the film progresses until, say, the hero and the villain are about to duel. At this point, which we call a *choice point,* the show is interrupted by an announcement requesting those in the audience favoring the villain's victory to move to a nearby projection hall. Once people have *chosen* to stay or to move, projections are resumed in both halls. Following a choice point each resumed projection is called an *episode.* It is easy to imagine that similar interruptions could take place at different points at the various projections, part of the audience always being directed to other halls according to their preference. The process could continue indefinitely, limited only by the number of spectators and halls.

Obviously a movie of the above kind has several plots, each of which can be represented graphically by a path from the root to a leaf of a tree (see Figure 1). This tree will be referred to as a *tree of choices.* A node (other than the root) corresponds to a choice point, a branch corresponds to an episode, and a leaf corresponds to "THE END" being flashed on the screen. Notice that the tree is not necessarily binary since

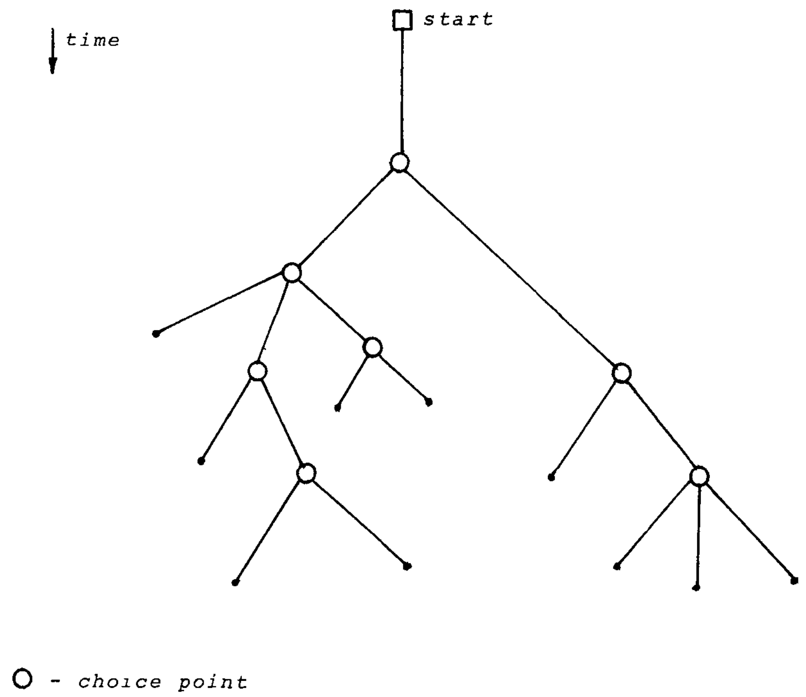


FIGURE 1.

choices may involve more than two outcomes.

This example will illustrate our first problem: "Determine the plot leading to a particular situation (say, one in which a given character is shot, survives, and ends up eloping with the heroine)."

There are two approaches which could be used to solve this problem, *parallel* and *sequential.*

In the first approach, the aid of a (potentially infinite) number of helpers for parallel viewing is requested. Their duty is to watch for the occurrence of the particular sequence of events which interests us. If choice points take place before this occurrence, the helpers simply split up and disperse into the different halls to watch the various sequels. Without the aid of the helpers, the parallel approach could only be used if we had the supernatural power to obtain copies of ourselves.[1] The spectator in charge of checking the desired sequence would obtain copies of himself at each choice point, the copies dispersing to the halls where the next episodes would be shown. A copy would cease to exist in the case of an *impasse.* Impasses occur when: 1) the movie ends and the sequence of desired events has not occurred, or 2) an event occurs which directly contradicts the desired sequence (say, in our example, if the character is *shot and dies*). The problem is solved when a copy witnesses the specified sequence of events. We could even ask him which projection halls he went through while checking for the sequence. It should be clear that for a specified sequence of events there may exist none, one, or several plots and therefore solutions to our problem. Furthermore, if the projection lasts indefinitely, the number of solutions may be infinite.

The solution of our problem using the

[1] Incidentally, Sabine, the beautiful heroine of the French novelist Marcel Aymé had such powers She could obtain new copies of herself whenever she desired. In the novel, Sabine resorted to her duplicating powers after falling in love with the handsome new lovers she had no trouble enticing By the end of the novel the number of Sabines rose to the thousands! In narrating what happens to the various Sabines, Aymé had to face the problem of describing non-deterministic situations to a reader who sequentially scans the text. This problem parallels that of translating a non-deterministic algorithm into one which is sequentially executed by a computer.

sequential approach involves: 1) an agreement with the movie operator to project the episodes at the times requested by the sole spectator solving our problem, and 2) a (potentially infinite) list where the spectator-solver will record information about choice points. This list is administered on a first-in last-out basis, i.e., as a stack. The strategy for solving the problem is as follows. At each choice point the spectator records the identity of the choice point and the number of its possible immediate sequels. He then asks the operator to project one of these sequels. As in the parallel mode, the spectator looks for the specified sequence of events. If an impasse is reached, the spectator asks the operator to project an as yet unviewed episode corresponding to the choice point which appears on the end of his list. If all episodes corresponding to a choice point have already been shown, the spectator erases the information about this choice point from the list and requests the projection of the next episode of the preceding choice point. (The reader familiar with algorithms for inspecting tree structures has undoubtedly noticed that the sequential mode corresponds to a depth-first inspection of the tree of choices.)

Let us assume that our spectator-solver has a very short memory for remembering plots. Most people are familiar with the backward projection of movies. Actions appear to be undone: divers spring from the water back to diving boards, people become younger, damage due to collisions is repaired, etc. If the solver of our problem has a short memory, he could ask the operator to run the movie backwards as soon as he sees an impasse. This backward projection is viewed until the scene corresponding to the most recent choice point appears on the screen. Projection is then either resumed in the forward direction with the next episode (if there is one left) or proceeds backwards to the preceding choice point. In this manner, the viewer confronted with an impasse has his memory "refreshed" to the state when the choice point was first encountered.

Essentially, non-deterministic algorithms are used to describe problems similar to the one just discussed. Some useful words for describing non-deterministic situations are: *choice, failure* and *success.* The first corresponds to a choice point, the second to an impasse, and the third to the finding of a desired sequence of events. Consider, for example, a non-deterministic movie having one choice point with three episodes. Suppose that we are interested in describing this movie, and determining the plot(s) consisting of the sequence of the two events: 1) "hero is shot and survives," and 2) "hero elopes with the heroine." This non-deterministic problem can be described by:

```
start
. . .
choice $e_1$, $e_2$, $e_3$
```

in which each of the episodes e_1, e_2, and e_3 contains the two conditionals:

```
. . .
if "hero is shot and dies" then failure
. . .
if "hero elopes with the heroine" then success
                                   else failure
```

The conditionals check for the desired sequence of events and signal a failure as soon as the facts are found inconsistent with the sequence.

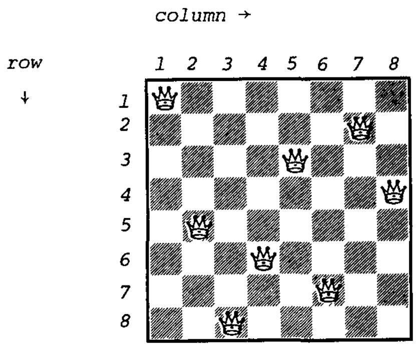
The above representation describes only the problem; it does not specify whether the parallel or the sequential mode is to be used for solving it.

In this paper we will concentrate on the sequential mode of solution to non-deterministic problems, since present-day computers are suited for that mode. (This situation will change as parallel processors become available.)

2. A CLASSICAL EXAMPLE

Floyd [FLOY67] illustrates the usefulness of non-deterministic algorithms with the classical 8-queens problem. The problem is to place 8 queens on a chessboard so that there is only one queen in each row, column or diagonal of the board (see Figure 2).

The tree of choices for this problem is too large to represent here, so we will first consider a scaled down version of the problem in which one must place four queens on a 4 × 4 chessboard. Four ordered integers from 1 to 4 suffice to indicate a configuration of the queens which satisfies the



(1, 5, 8, 6, 3, 7, 2, 4)
FIGURE 2.

problem. For example, the ordered quadruple (2, 4, 1, 3) indicates that the queens are in rows 2, 4, 1, and 3, of columns 1, 2, 3 and 4 respectively. This configuration is depicted in Figure 3a. Figure 3b shows part of a tree of choices for the 4-queens problem. Note that board configurations involving fewer than 4-queens can still be represented by quadruples. We use asterisks to indicate queens which are yet to be positioned. The root of the tree of choices can then be labelled by the configuration (*, *, *, *).

As the first choice, we position the first queen on the first column. There are four resulting configurations possible, and these are defined by the quadruples: (1, *, *, *), (2, *, *, *), (3, *, *, *), and (4, *, *, *). These are (partly) shown in the tree of Figure 3b. For each of the above configurations there are four possible ways of placing the second queen in the second column. Obviously the configurations (1, 1, *, *) and (1, 2, *, *) represent a failure—no matter what digits replace the asterisks. These nodes of the tree are labelled with an *F*.

The darker line in the figure shows the path in the tree of choices which leads to a solution of the problem. The reader is urged to verify that the sequential approach for solving this problem involves inspecting the tree nodes numbered 1, 2, 3, . . .9. He may also find it useful to complete the tree shown in Figure 3b.

Although we have consistently represented the branches coming from each node in ascending order, this is not strictly necessary. Actually, the order in which the branches are considered is irrelevant. This makes the word non-deterministic applicable, since there is an arbitrary choice among possible outcomes.

Now we are ready to describe a nondeterministic algorithm for solving the problem. Let the variables *col* and *row* represent a column and a row where a queen can be placed. We also assume that the test *T* below can be easily performed:

T: a queen at position (row, col) does not conflict with previously placed queens.

We will use the command:

choice among *row* ← *1* **to** *4,*

to indicate a node of our tree and its four branches. A non-deterministic algorithm which describes the tree in Figure 3b follows:

```
col ← 1;
choice among row ← 1 to 4;
if T then place queen at (row, col)
    else failure;
print (row);
col ← 2;
choice among row ← 1 to 4;
  if T then place queen at (row, col)
      else failure;
print (row);
col ← 3;
choice . . . (as above)
. . .
col ← 4;
choice . . .
. . .
success;
```

The following points are worth noting:

1) The board is assumed to be initially empty. The words **choice, failure,** and **success** were intuitively defined in the previous section.
2) The tree of choices is inspected from the root (*col* = 1) towards the leaves (*col* = 4).
3) The non-deterministic command *print* (*row*) consists of internally writing the value of *row.* (This value may be erased if later, during the execution, the branch containing the *print* command is backtracked.) Notice that the value of *col* need not be printed since it always increases from 1 to 4.
4) The command **success** is only reached after the cascading **choice**'s are executed. At that point the printed values of *row* reveal the solution of the problem.

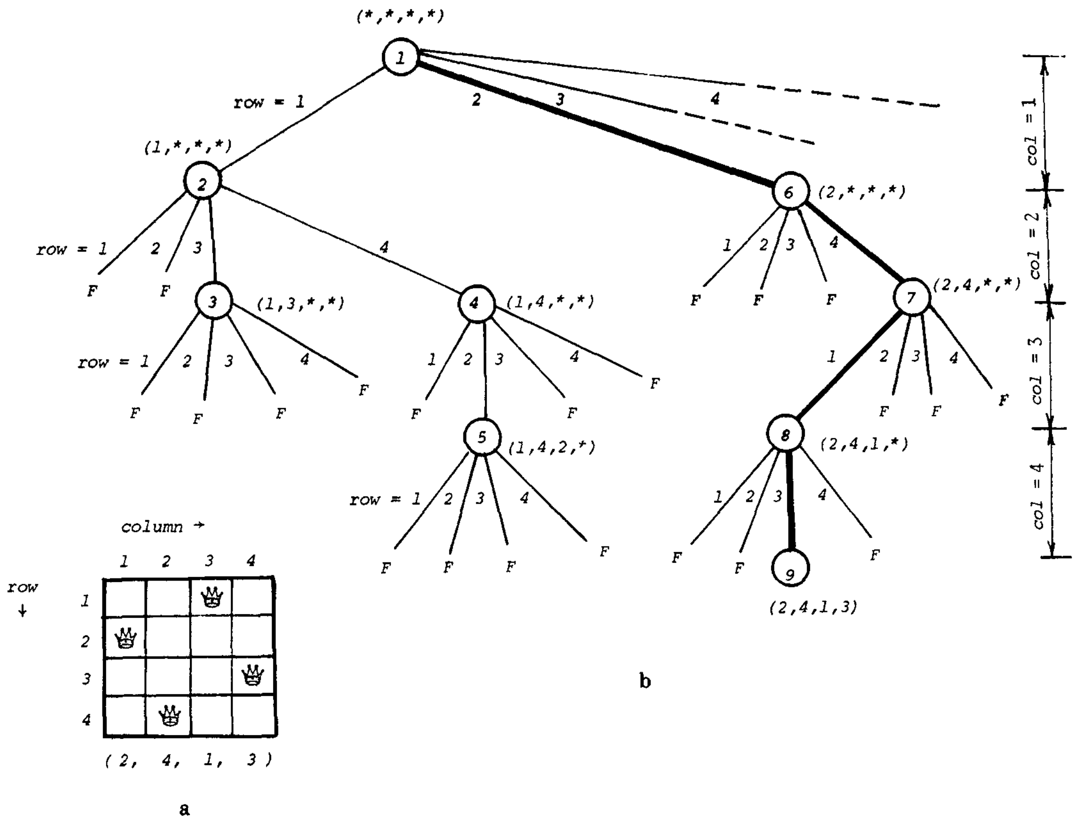


FIGURE 3.

A succinct version of the above program is obtained by using a **for** statement to successively assign the values 1, 2, . . . etc. to the variable *col*. Our original 8-queens problem can then be described as the following non-deterministic algorithm:

```
for col ← 1 to 8 do
   begin
     choice among row ← 1 to 8;
     if T then place queen at (row, col)
          else failure;
     print (row)
   end;
success;
```

This formulation is at least as succinct and perhaps more natural than the recursive algorithms which solve this problem (see DIJK72 and WIRT76).

3. TRANSLATION INTO STANDARD PROGRAMS

Non-deterministic algorithms can be translated automatically into computer programs by using transformation rules. The rules may be found in FLOY67 and COHE75. This presentation describes informally how the translation is accomplished, using the 8-queens problem as an example.

The translated program operates in the sequential mode described in the first section. Upon failure it backtracks to the previous choice point in a manner analogous to the backward projection of movies. When the statement $i \leftarrow i + 1$ is found in a program, it is executed in the backward direction as $i \leftarrow i - 1$, to undo what was originally performed. Another manner of handling assignments is to save the value of the variable just prior to the assignment, and to restore it to that variable during the backward execution.

Let us consider a sequence of non-deterministic commands $S_1, S_2, \ldots S_n$. We will indicate by S_i^+ the translation of S_i in the forward mode, and by S_i^- the translation of S_i in the backtrack mode.

The translation of the sequence into a

standard program uses a global Boolean variable B which indicates the mode: forward, or backtrack. We have:

```
                        ⟨forward⟩ ¦ ⟨backtrack⟩
     i ← 1; B ← true;           ¦
 loop: case i of                ¦
                        S1+     ¦ S1-,
                        S2+     ¦ S2-,
                         ·      ¦
                         ·      ¦
                        Sn+     ¦ Sn-
                                ¦
       end of case;
       i ← i + (if B then 2 else -2);
       goto loop;
```

The case statement transfers control to the ith statement contained between the delimiters **of** and **end**. The variable i is an instruction counter which is advanced to the next forward or backtrack instruction, depending on the value of B. This value can, of course, be altered by some of the S_i^+ or S_i^-. At least one of the S's is a transfer to a label outside of the loop.

As in the case of assignments the statement: **choice among** $x \leftarrow 1$ *to* n is translated into two parts: one for the forward execution and the other for backtracking. These will be represented respectively by the two procedures *fchoice* and *bchoice*, each having two parameters: the controlled variable x and its upper limit n. The procedures listed in Figure 4 assume that the variable x is successively assigned to the values 1, 2, $\cdots$ n. As previously stated, this is not strictly necessary. The variable x could have been successively assigned to the elements of a random permutation of the sequence 1, 2, $\cdots$ n. However, a glance at the tree of choices for the 4-queens problem (Figure 3b) shows the importance of selecting the "right" permutations, i.e., those that minimize searches through blind alleys.

The auxiliary procedures *backtrack* and *forward* readjust the values of B and i so that the execution proceeds in the corresponding mode. A stack, M, is used for saving the values of the controlled variable.

```
procedure fchoice (x,n) ; integer x,n ;
        if x>n then backtrack else
           begin stack M (x) , x←1 end ;
procedure bchoice (x,n) ; integer x,n ,
        begin x←x+1 ;
              if x>n    then unstack M (x)
                        else forward
        end ;
procedure backtrack , begin B←false ; i←i+1 end,
procedure forward ; begin B←true ; i←i-1 end ;
```

FIGURE 4.

Another stack, W, is used in the translation of a non-deterministic *print* (t) command. In the forward execution the translation of this command pushes into W the value of the variable t. In backtracking that value is simply popped, i.e., erased from the stack W.

Consider the non-deterministic **for** statement:

```
previous: ···;
   for j ← 1 to n do
      begin S1, S2, ··· Sm end;
next: ···;
```

A possible translation into a standard program is given by:

```
                      ⟨forward⟩ ¦ ⟨backtrack⟩
previous: ···;                  ¦
      i ← j ← 1;                ¦
      B ← true;                 ¦
loop: case i of                 ¦
      ef:  if j > n then        ¦
           goto next            ¦ eb:  if j = 1 then
                                ¦      goto previous
                                ¦      else i ←
                                ¦           2*m + 6,
                          S1+   ¦ S1-,
                          S2+   ¦ S2-,
                           ·    ¦
                           ·    ¦
                          Sm+   , Sm-,
                   j ← j + 1    ¦ j ← j - 1,
      repeat: i ← - 1           ¦
      end of case;              ¦
      i ← i + (if B then 2 else -2); goto loop;
next: ··· ;
```

The statements labelled *ef* and *eb* control the loop exit in the forward and backtrack directions. The one labelled *repeat* simulates the action of the **for** statement, i.e., it forces control to be transferred to the first statement of the **case**. Similarly, the false branch in the conditional labelled *eb* (i.e., $i \leftarrow 2^*m + 6$) forces the transfer of control to the statement which decreases the value of the controlled variable j, and therefore starts undoing the **for** statement.

Before presenting the final version of the translation of the algorithm for solving the 8-queens problem, we will make some remarks about the implementation of the test T in the non-deterministic algorithm. (Recall that this test is needed for checking if a queen can be placed at a given row and column of the board.)

We note that the sum of the indices of each element in one left-right diagonal of a matrix representing the chessboard is the same. This is also true for the differences of indices in a right-left diagonal. Therefore, two unidimensional Boolean arrays b and c can be used to test for the existence of queens in the diagonals. A third Boolean array a is used to indicate the presence of a queen in a row. The bounds of a, b, and c are respectively: 1:8, 2:16 (i.e., the range of the sum of the indices of matrix elements) and −7:7 (i.e., the range of the differences of indices). The arrays are initialized to **false.** The presence of a queen in a row or diagonal is indicated by setting the corresponding element of an array to **true.** Test T then becomes:

$$\neg\, (a[row] \vee b[row + col] \vee c[row - col])$$

According to this data structure, the placement of a queen in a row is represented by:

a[row] ← *b[row + col]* ← *c[row − col]* ← **true**

Its backtracking counterpart is therefore:

a[row] ← *b[row + col]* ← *c[row − col]* ← **false**

Keeping the information given in the previous paragraphs in mind, the reader should have no difficulty in understanding the program in Figure 5. It corresponds to the translation of our original non-deterministic algorithm.

The execution of the program in Figure 5 yields as a result the 8-tuple (1,5,8,6,3,7,2,4) which corresponds to the first solution of the problem (see Figure 2).

If we desired all successful configurations it would suffice to add a line consisting of the statements:

13) B ← **false**; *i ← 14;* **goto** *loop;*

These statements simulate an impasse by triggering the backtrack mode and transferring control to *col* ← *col* − 1 in line 9.

4. VARIANTS OF THE CHOICE COMMAND

In the previous section the **choice** command appeared in a form which only allowed a controlled variable x to be successively assigned values from 1 to n. In this section we will describe more versatile versions of **choice.** Consider the database consisting of the lists:

(CLIMATE ECUADOR TEMPERATE)
(SOIL URUGUAY FERTILE)
(CLIMATE URUGUAY TEMPERATE)
(SOIL ECUADOR BARREN)
(CLIMATE PARAGUAY TEMPERATE)
(SOIL PARAGUAY FERTILE)
(GROWS PARAGUAY WHEAT)
(GROWS ECUADOR CACTUS)

The above list contains geographical information about Latin American countries, and has been used by Carbonell to illustrate his question-answering program (see BOBR74).

An interesting variant of the **choice** command appears under the disguise of a *search.* Consider the two commands:

search (CLIMATE x TEMPERATE)
search (SOIL x FERTILE)

```
1)          initialize arrays a, b, c to false, stacks M and W to empty
            i ← 1 ; B ← true ; col ← 1 ;                 |
                                              (forward) | (backtrack)
                                              ----------|------------
2) loop:    case i of                                   |
3)                       if col > 8 then goto success   | if col = 1 then stop else i ← 14,
4)                                    fchoice (row, 8)  | bchoice (row, 8),
                                                        |
5)            if a [row] ∨ b [row+col] ∨ c [row−col]    |
6)                                     then backtrack   | no operation,
7)             a[row]←b[row+col]←c[row−col]←true        | a[row]←b[row+col]←c[row−col]←false,
8)                                    stack W (row)     | unstack W,
9)                                        col←col+1     | col←col−1,
10)                                           i← −1     |
          end of case ;
11)       i  ← 1 + (if B then 2 else − 2); goto loop ;
12) success · print W ;
```

FIGURE 5.

These commands request binding the variable x to a country in the database such that both searches are satisfied. In other words, these commands correspond to determining a country (or countries) with temperate climate and fertile soil. A nondeterministic situation arises because there is a choice for binding x when satisfying the first search. If the sequential approach is used, x is initially bound to *ECUADOR*. When the second search is attempted a failure occurs since that country has a barren soil.

Backtracking takes place, the first search is resumed and x is now bound to *URUGUAY*. This binding is satisfied by the second search. Had we wished to check if the database contained other countries which satisfied the two *search* commands, we could, as in the case of the 8-queens, simulate a failure immediately after having bound x to *URUGUAY*. This would enable us to find *PARAGUAY* as another suitable binding for x.

We will now informally describe a minilanguage containing several features of languages proposed for solving problems in artificial intelligence [BOBR74]. Programs written in this language may contain sequences of *search* commands as illustrated above. In addition, these sequences can be grouped into procedures which may or may not have formal parameters and local variables. Calls of procedures are accomplished by a variant of **choice,** referred to as **orchoice.** Its parameter is a list of *searches* or procedures being called. Finally, we will allow the **orchoice** command to appear within the body of procedures. Recursive procedures are, of course, permitted.

Both the commands *search* and **orchoice** embody properties of choice points. In the case of *search,* the set of possible values consists of the lists contained in the database which could be successfully matched. The command **orchoice** greatly increases the power of the language. **orchoice,** like *search,* successively tries a number of paths until it finds one leading to a successful situation. In this case, the paths represent the execution of the commands which are given as **orchoice** parameters. The **orchoice** command operates as a backtrackable coroutine. Every time **orchoice** is re-entered in the backtrack mode, it proceeds from where it had left off; however, care must be taken to first undo the tasks performed during the latest activation. The example below illustrates the potentially intricate control structure which results from using the **orchoice** command. Consider the following program in which the S_i's are basic commands:

```
    $S_1$;
    orchoice $S_2$ $S_3$ $S_4$;
    $S_5$;
```

Let us, as before, denote by S_i^+ the execution of S_i in the forward mode and S_i^- the corresponding execution in the backtrack mode. Assume that sometime during the execution S_i^+ leads to a backtrack mode; we will indicate this occurrence by the notation $S_{i\uparrow}^+$. The following sequences of S_i's show some of the possible flows of control while the above program is executed:

$S_{1\uparrow}^+$ S_1^- ··· *(continues backtracking)*
S_1^+ $S_{2\uparrow}^+$ S_2^- S_3^+ S_5^+ ··· *(continues forward)*
S_1^+ S_2^+ $S_{5\uparrow}^+$ S_5^- S_2^- S_3^+ S_5^+ ··· *(continues forward)*
S_1^+ $S_{2\uparrow}^+$ S_2^- $S_{3\uparrow}^+$ S_3^- S_4^+ $S_{5\uparrow}^+$ S_5^- S_4^- S_1^- ··· *(continues backtracking)*

The translation of the *search* and **orchoice** commands into their deterministic counterparts can be made in a manner similar to that described in Section 3. The command *search* is really a variant of **choice** and is translated into calls of procedures *fsearch* and *bsearch* which parallel *fchoice* and *bchoice* (see Figure 4). The translation of **orchoice** is slightly more complex: it involves simulating non-deterministic subroutines. This is done by first checking if a subroutine is to be executed in the forward or in the backtrack mode. If it is to be executed in the backtrack mode, control is transferred to "undo" the last statement of the subroutine; otherwise, control is transferred to the first statement for normal forward execution. Details are found in FLOY67.

We are now ready to present a program which answers questions about crops growing in the countries described by the database given at the beginning of this section.

Consider, for example, the question: *What grows in Uruguay?* Notice that the database does not explicitly contain the answer to this question. However, the answer can be inferred by using the following inference rule: Whenever the crop of a country is not present in the database, provide as an answer the crop of some other country which has the same soil and climate conditions. This rule can be implemented by the two procedures listed below (the variables in the lists appearing as parameters are represented by small case identifiers; capital letters are used in the atoms of the lists to indicate actual (quoted) strings):

```
proc grows (country);
     begin local crop;
       comment first check if the answer is
          available in the database. If not, call
          the procedure infer;
       orchoice search (GROWS country
          crop)
             infer (country crop);
       write (country CAN GROW crop)
     end of grows;
proc infer (country crop);
     begin local country2, soil, climate;
       comment this procedure incorporates
          the inference rule: if two countries
          have identical climate and soil con-
          ditions, they can grow the same crops;
       search (CLIMATE country climate);
       search (SOIL country soil);
       search (CLIMATE country2 climate);
       search (SOIL country2 soil);
       search (GROWS country2 crop)
     end of infer;
```

Upon a call of *grows (URUGUAY)*, the parameter *country* is bound to *URUGUAY*. The **orchoice** in the procedure *grows* is then executed: First the *search (GROWS URUGUAY crop)* is performed, but it fails since the database contains no reference to crops in Uruguay. The next statement of the **orchoice** command is executed, namely: *infer (URUGUAY crop)*

This execution triggers the command:

search (CLIMATE URUGUAY climate)

The variable *climate* is bound to *TEMPERATE*. The next search in *infer* is:

search (SOIL URUGUAY soil)

which binds *soil* to *FERTILE*. The command:

search (CLIMATE country2 TEMPERATE)

binds *country2* to *ECUADOR*. Since *(SOIL ECUADOR FERTILE)* is not contained in the database, the backtracking mode is triggered: the binding of *country2* to *ECUADOR* is undone, the forward mode is resumed and *country2* is bound to *URUGUAY*. The search for (*SOIL URUGUAY FERTILE*) would then be satisfied. However, a search for (*GROWS URUGUAY crop*) fails and backtracking again occurs.

The reader can easily check that when *country2* is bound to *PARAGUAY*, the searches in the procedure *infer* are all satisfied and *crop* becomes bound to *WHEAT*. Upon returning to the procedure *grows*, *crop* is still bound to *WHEAT* and the *write* statement then produces the desired output:

(URUGUAY CAN GROW WHEAT)

Note that an efficient implementation of the procedure *infer* would reorder the successive searches that appear in its body. Such reordering could help to trigger a failure sooner and therefore bypass unfruitful searches.

5. A FINAL EXAMPLE

This example illustrates the use of the minilanguage of the last section in writing parsers for context-sensitive languages [HOPC69]. Only rudimentary knowledge of language theory is required for understanding this example since this parsing problem can be formulated in terms of graph transformations.

Consider the rewriting rules:

Rule No.	
1	$S \rightarrow aSBc$
2	$S \rightarrow abc$
3	$cB \rightarrow Bc$
4	$bB \rightarrow bb$

Starting with the letter S and using the second rule, we obtain the string *abc* and no further rules are applicable to transform the latter string. If, however, we start with an S but use the first rule instead, we obtain the string *aSBc*. Further replacements of S by Rule 1 yield the string

$a\ a \cdots a\ S\ Bc\ Bc \cdots Bc,$

in which the number of a's preceding the S equals the number of Bc's which follow it. A further replacement of S by Rule 2 yields

$a\ a \cdots aabcBc\ Bc \cdots Bc.$

Successive applications of the third rule allows us to move all B's towards the center of the string and all c's towards the right. We obtain

$aa \cdots aab\,B\,B \cdots B\,cc \cdots cc.$

Finally, application of the last rule enables us to rewrite the above string into

$aa \cdots aa\,bb \cdots bb\,cc \cdots cc.$

No rules are now applicable and the above strings contain the same number of a's, b's and c's. Using language theory terminology, we say that the given rewriting rules generate the language $a^n b^n c^n$ with $n \geq 1$. It is not difficult to show that the given rewriting rules generate only the strings of this language. The problem of parsing consists of showing how a given string could have been rewritten from S using the rules.

A graphical solution of the problem for the string *aabbcc* is shown in Figure 6. The elements of this string are initially used as labels of arcs of a directed graph from the node *LEFT* to the node *RIGHT* (solid lines in the figure). A traversal of this graph from left to right, consulting the rewriting rules, reveals that the subgraph labelled *bb* corresponds to the right-hand-side (rhs) of the fourth rule. We indicate that the lhs *bB* could have been used to obtain the substring *bb* by deleting the arcs labelled *b* and adding the new arcs *b* (from node 2 to 3) and *B* (from 3 to 4). The added arcs are indicated by the dashed lines in Figure 6. The reader can easily verify that a new traversal of the transformed graph shows that the lhs of *Bc* corresponds to a subgraph linking nodes 3, 4, 5. A replacement of this subgraph by one labelled *cB* now takes place. The process continues until the original graph is replaced by an arc labelled *S* going from *LEFT* to *RIGHT*.

While performing the graph transformations, several non-deterministic situations may arise. They are due to the fact that several rhs of rules may be applicable and each one of them may have to be tried.

In order to write a parser for the given language we will introduce two new commands to our minilanguage: The command *add* (*L*), adds the list (*L*) to the database. Its counterpart, the command *delete (L)*, deletes the list *L*. It follows that the backtracking counterpart of *add* is *delete* and vice versa.

A specific parser for the language generated by the given rules is presented in Figure 7. Each of the rules is translated into a procedure. Only those corresponding to Rules 1 and 3 are presented in Figure 7, since the others can be easily constructed by the reader.

The concatenation of elements of the vocabulary is expressed using the auxiliary variables p_i. The presence in the database of lists of the type:

$(p_1 \quad A \quad p_2)$
$(p_2 \quad B \quad p_3)$
$(p_3 \quad C \quad p_4)$

indicates that A is concatenated with B which in turn is concatenated with C.

The identifiers *A, B, C, BIGB, BIGS* are used to denote *a, b, c, B,* and *S*. The procedure expressing Rule 1 states that if the

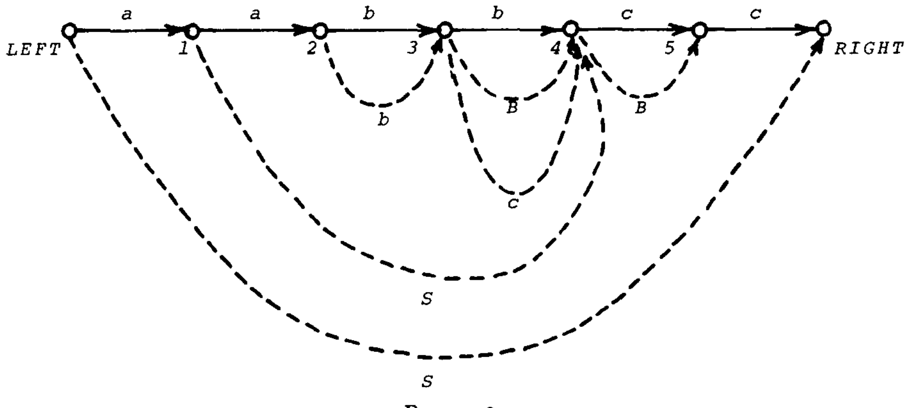


FIGURE 6.

```
begin proc rule1;
            begin local p1, p2, p3, p4, p5;
              delete (p1 A p2);
              delete (p2 BIGS p3);
              delete (p3 BIGB p4);
              delete (p4 C p5);
              add (p1 BIGS p5);
              write (FOUND BIGS FROM p1 TO p5)
            end of rule1;
       proc rule3;
            begin local p1, p2, p3,
              delete (p1 BIGB p2),
              delete (p2 C p3);
              add (p1 C p2);
              add (p2 BIGB p3),
              write (FOUND C AND BIGB FROM p1 TO p3)
            end of rule3;
            comment procedures for rules 2 and 4 are similar to the above and are not presented here;
              . ;
            proc parse;
              begin
                 orchoice rule1 rule2 rule3 rule4,
                 orchoice terminate parse;
              end of parse;
            proc terminate;
              begin
                 search (LEFT BIGS RIGHT);
                 success
            end of terminate;
            main: . . .
end
```

FIGURE 7.

sequence *a, S, B, c* is found in the database, then that sequence should be deleted and replaced by an *S*. Notice that the left and right neighbors of *aSBc* become the neighbors of *S*.

The procedure *parse* is the heart of the program. It first tries to apply any of the rules. If it succeeds, it checks for termination by calling the procedure *terminate*. This procedure assumes that the string to be parsed is "placed" between the delimiters *LEFT* and *RIGHT*. If *terminate* fails, a recursive call of *parse* is needed to keep applying rules which will eventually lead to a successful parse.[2] The parsing of the string *aabbcc* is accomplished by calling the procedure *parse*, after initializing the database to contain the lists: (*LEFT A* 1), (1 *A* 2), (2 *B* 3), (3 *B* 4), (4 *C* 5), (5 *C RIGHT*).

[2] Recursion is frequently used in parsing algorithms. The reader familiar with recursive descent parsers will notice similarities between that type of parser and the one presented in Figure 7. Note, however, that a recursive descent parser is "top-down," whereas the one in Figure 7 is "bottom-up"

6. CLOSING REMARKS

In the preceding sections we have covered some of the now classical concepts regarding non-deterministic algorithms. Our objective in this section is to make a brief survey of the more recent work done in this area. The references should help the interested reader to obtain additional information

Additional Desirable Primitives

The primitives described in the previous sections allow only a depth-first inspection of the tree of choices. A greater flexibility for this inspection can be achieved by adopting the primitives suggested in McDE74 and MONT77. For example, a proposed variant of **choice**, called *attempt,* enables the user to provide an ordering of its parameters so that the search for a successful configuration will, it is hoped, be faster. The primitive *sleep,* similar but of a less drastic effect than **failure**, also triggers backtracking. However, upon encountering its matching primitive *resume*, control is transferred back to the statement following *sleep*. These two primitives allow a greater flexibility in inspecting the tree of choices: Assume, for example, that a **choice** command involves many parameters (possibly generated while the program is being executed). Since many branches of the tree will now have to be examined, it may be worthwhile to postpone the examination by issuing the *sleep* command. This causes other parts of the tree to be inspected and more information collected about the branches which were "put to sleep." Some of these branches may then be pruned using the primitive *kill*. The original choice node may then look more appealing for examination, since it contains fewer branches. This examination can now be triggered by *resume*. The use of these primitives for solving the 8-queens problem is described in MONT77.

Implementation

There are several papers in the literature describing the implementation of non-deterministic algorithms written on a variety of languages. Floyd's paper [FLOY67] describes how to transform ("by hand") a flowchart containing non-deterministic primitives into a standard one. Johansen in JOHA67 shows how an ALGOL compiler has been modified to accept non-deterministic programs. The preprocessor described in COHE74 transforms extended FORTRAN programs which use non-deterministic primitives into standard FORTRAN. Embeddings of non-determinism in LISP, ECL, BCPL and a SNOBOL-like language have been proposed in SMIT73, PREN72, SELF75, and GRIS77.

Bitner and Reingold [BITN75] show examples of the use of macros in implementing non-deterministic algorithms. Reference BOBR73 is valuable when implementing the non-deterministic primitives described in the beginning of this section.

It should be noted that a standard program simulating a non-deterministic algorithm could operate in a manner different from the one presented in Section 3. For example, the entire state of the computer could be saved (perhaps using secondary storage) every time a choice point is encountered. Upon failure, the computer is reset to its previous state just before the choice, and its next parameter is considered. Whether this latter technique should be used instead of the one in Section 3 depends, of course, on the amount of storage needed to save the state of the computer and the speed with which this information can be transferred to secondary storage. Reference SMIT73 discusses the advantages and inconveniences of the two approaches.

Another aspect of implementation worth mentioning is optimizing the programs which simulate non-deterministic algorithms. As seen in the geography example of Section 4, this optimization involves altering the order in which searches are performed. In artificial intelligence applications, this reordering may be crucial for the practical solution of a problem [MOOR75]. Optimization also includes the use of hash tables to speed up the execution of set operations.

Program Correctness

It is known that certain recursive algorithms are more easily proved to be correct than their non-recursive counterparts [MANN76]. The same may hold true for certain non-deterministic algorithms [DERO77]. Techniques presently used in program correctness have also been used to eliminate searches through blind alleys from programs simulating non-deterministic algorithms [SINT76]. In certain cases it is even possible to avoid all blind alleys, or in other words, to eliminate backtracking. A classical example of this situation arises in connection with non-deterministic finite-state-automata, a class of abstract machines capable of recognizing certain types of strings [HOPC69, MANN76]. A theorem in language theory [HOPC69] states that it is always possible to find a deterministic-finite-state automaton equivalent to a non-deterministic one. Of course, this is not always feasible when dealing with more general types of automata.

Efficiency and Complexity

The difficulties and frustrations involved in estimating the efficiency of non-deterministic algorithms have been aptly expressed by Knuth [KNUT75]:

Sometimes a backtrack program will run to completion in less than a second. The author once waited all night for output from such a program, only to discover that the answers would not be forthcoming for about 10^6 centuries. A "slight increase" in one of the parameters of a backtrack routine might slow down the total running time by a factor of a thousand; conversely a "minor improvement" to the algorithm might cause a hundred fold improvement in speed; and a sophisticated "major improvement" might actually make the program ten times slower. These great discrepancies in execution time are characteristic of backtrack programs, yet it is usually not obvious what will happen until the algorithm has been coded and run on a machine.

In spite of these discouraging prognostics, however, Knuth proposes a semi-empirical method for estimating the efficiency of backtrack programs, which, by the evidence provided, works well in a variety of cases. It consists of evaluating a formula whose coefficients are determined by the use of Monte Carlo techniques, more precisely, by performing random walks in the tree of choices.

To conclude, let us briefly reconsider a hypothetical version of the non-deterministic movie problem of Section 1. Assume that every episode lasts for a time Δ, and that each choice point always results in a fixed number of episodes, say m. This implies that the tree of choices is m-ary and that the "length" of each branch is Δ. We also assume that the tree is full, i.e., each path from the root to a leaf has length $\ell\Delta$. It follows that the time for solving the corresponding movie problem using the parallel approach is $\ell\Delta$. However, a sequential approach could, in the worst case, take time $m^\ell\Delta$ since the desired plot might be the last one to be examined. Thus the parallel approach for solving the problem takes linear time, whereas the sequential approach takes exponential time. It is also true that if we are wise (or lucky) in choosing the paths in the tree of choices, the time for the sequential approach could be linear as well. Notice too that the parallel approach may require an exponential number of helpers.

Some related important theoretical results have been obtained in the area of complexity of algorithms [AHO76]. They state the existence of very hard non-deterministic problems which can only be solved in exponential time when using the sequential approach. This means that, for a certain class of problems, the "wisdom" necessary to avoid exponential time complexity is believed to be unattainable. These problems are said to belong to the class *NP*, which stands for "non-deterministic polynomial." The term "polynomial" indicates that the time needed to follow a path from the root to a leaf of the tree is expressible as a polynomial function of ℓ. (Our movie example belongs to this class since the corresponding time is linear.)

Several algorithms for solving problems in graph theory or in operations research, and for proving theorems in propositional calculus belong to the *NP* category. Recent results in complexity theory show that there is a remarkable subclass of *NP* called *NP-complete*. These are problems which are reducible to each other in the following sense: if at some time in the future one of them is found to have an efficient method of solution, that method can be modified to apply to all the others. The first proof that a problem belonged to the *NP*-complete class was given by Cook in COOK71. He established a correspondence between certain non-deterministic algorithms and theorems in propositional calculus for which proofs appear to require exponential time.[3] According to this correspondence, non-determinism can be viewed as *checking* a given proof of a theorem (i.e., following *one* path of the tree of choices and checking for success or failure). Determinism, on the other hand, corresponds to *finding* the proof of a theorem (i.e., exhaustively searching *all* branches of the tree). The

[3] This important classification of algorithms parallels A. Turing's classification of solvable and unsolvable problems. He proved the unsolvability of several problems by reducing them to a known unsolvable problem called the halting problem.

reader is referred to LEWI78 for a pedagogic presentation of several *NP*-complete problems and important open questions in complexity theory.

Although it may be impossible to avoid the exponential time complexity of simulating certain non-deterministic algorithms, a careful ordering of the branches of the tree of choices may greatly reduce the time needed for solving a problem. This may be accomplished by combined efforts in two directions. The first is people-oriented and consists of providing problem solvers with "adequate" sets of primitives, allowing them to program the best ordering for inspecting the tree. The second is machine-oriented and uses sophisticated optimization techniques designed to avoid, if possible, fruitless searches, or at least to make them as efficient as possible. Continued research in these directions may be the only hope to surmount the difficulties involved in solving currently intractable combinatorial problems.

ACKNOWLEDGMENTS

Robert Stachel implemented the minilanguage described in Section 2 and tested all the programs presented in this paper Joel Katcoff read the manuscript several times and made numerous suggestions which improved its readability. Gretchen Brown provided valuable help in preparing the final version of the manuscript. The comments and advice given by the referees are gratefully acknowledged.

In particular, the author was fortunate to have had Robert Sedgewick as one of the referees. His constructive and detailed comments were a source of encouragement for improving the paper.

REFERENCES

AHO76 AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The design and analysis of computer algorithms,* Addison-Wesley Publ Co., Reading, Mass., 1976.

BITN75 BITNER, J. R., AND REINGOLD, E. M. "Backtrack programming techniques," *Commun. ACM* **18,** 11 (Nov 1975), 651-656.

BOBR73 BOBROW, D. G , AND WEGBREIT, B. "A model and stack implementation of multiple environments," *Commun ACM* **16,** 10 (Oct. 1973), 591-603.

BOBR74 BOBROW, D G., AND RAPHAEL, B "New programming languages for artificial intelligence research," *Comput. Surv.* **6,** 3 (Sept 1974), 155-174

COHE74 COHEN, J , AND CARTON, E "Non-deterministic Fortran," *Comput. J.* **17,** 1 (Feb. 1974), 44-51.

COHE75 COHEN, J. "Interpretation of non-deterministic algorithms in higher-level languages," *Inf. Process. Lett.* **3,** 4 (March 1975), 104-109.

COOK71 COOK, S. A. "The complexity of theorem-proving procedures," in *Proc. Third Annual ACM Symp. Theory of Computing,* 1971, ACM, New York, 151-178.

DERO77 DEROEVEN, W. P. "On backtracking and greatest fixed points," in *Automata, languages and programming,* Lecture Notes in Computer Science, Vol. 52, Springer-Verlag, New York, 1977, 412-429.

DIJK72 DIJKSTRA, E W. "Notes on structured programming," in *Structured programming,* O. J. Dahl (Ed.), Academic Press, New York, 1972, 1-72.

FLOY67 FLOYD, R. W. "Nondeterministic algorithms," *J. ACM* **14,** 4 (Oct. 1967), 636-644

GOLO65 GOLOMB, S. W., AND BAUMERT, L. D. "Backtrack programming," *J. ACM* **12,** 4 (Oct. 1965), 516-524.

GRIS77 GRISWOLD, R. E., AND HANSON, D. R. "Language facilities for programmable backtracking," in *Proc. Symp. Artificial Intelligence and Programming Languages* Published in *SIGPLAN Not* **12,** 8 (Aug 1977), 94-99.

HOPC69 HOPCROFT, J E., AND ULLMAN, J. D. *Formal languages and their relation to automata,* Addison-Wesley Publ. Co., Reading, Mass., 1969.

JOHA67 JOHANSEN, P. "Non-deterministic programming," *BIT* **7** (1967), 289-304

KNUT75 KNUTH, D. E. "Estimating the efficiency of backtrack programs," *Math. Comput.* **29,** 129 (Jan 1975), 121-136.

LEWI78 LEWIS, H R., AND PAPADIMITRIOU, C. H. "The efficiency of algorithms," *Sci. Am* (Jan. 1978), 96-109.

MANN76 MANNA, Z. *Mathematical theory of computation,* McGraw-Hill, New York, 1976.

MCDE74 MCDERMOTT, D. V., AND SUSSMAN, G. J. *The Conniver reference manual,* AI Memo 259, MIT Project MAC, Cambridge, Mass., Jan 1974.

MONT77 MONTANGERO, C , PACINI, G , AND TURINI, F "Two-level control structure for nondeterministic programming," *Commun ACM* **20,** 10 (Oct. 1977), 725-730.

MOOR75 MOORE, R. C. *Reasoning from incomplete knowledge in a procedural deduction system,* AI-TR-347, MIT Artificial Intelligence Lab., Cambridge, Mass., Dec. 1975

PREN72 PRENNER, C , SPITZEN, J., AND WEGBREIT, B. "An implementation of backtracking for programming languages," in *Proc 27th National ACM Conf.,* 1972, ACM, New York, 763-771

SELF75 SELF, J A. "Embedding nondeterminism," *Softw. Pract Exper.* **5** (1975), 221-227.

SINT76 SINTZOFF, M. "Eliminating blind alleys from backtrack programs," *Third Int. Colloquium on Automata, Languages and Programming,* Lecture Notes in Computer Science, Springer-Verlag, 1976, New York

SMIT73 SMITH, D. C., AND ENEA, H. J. "Backtracking in MLISP2," in *Proc Third Int. Conf. Artificial Intelligence,* 1973, 671-685.

WALK60 WALKER, R. J. "An enumerative technique for a class of combinatorial problems," in *Combinatorial analysis; Proc. Symp. Applied Mathematics,* Vol X, American Mathematical Society, Providence, R.I, 1960

WIRT76 WIRTH, N. *Algorithms + data structures = programs,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976. See also, *Commun. ACM* **14,** 4 (1971), 221-227.

RECEIVED NOVEMBER 11, 1977, FINAL REVISION ACCEPTED OCTOBER 15, 1978