A procedure mechanism for backtrack programming

Article · January 1976

DOI: 10.1145/800191.805625

1 author:

David Roy Hanson
Princeton University
90 PUBLICATIONS **1,575** CITATIONS

SEE PROFILE

A PROCEDURE MECHANISM FOR BACKTRACK PROGRAMMING*

David R. HANSON+
Department of Computer Science, The University of Arizona
Tucson, Arizona 85721

One of the difficulties in using nondeterministic algorithms for the solution of combinatorial problems is that most programming languages do not include features capable of easily representing backtracking processes. This paper describes a procedure mechanism that uses coroutines as a means for the description and realization of nondeterministic algorithms. A solution to the eight queens problem is given to illustrate the application of the procedure mechanism to backtracking problems.

1. INTRODUCTION

Although backtrack programming has been known for several years [1-4], the method has yet to become a common programming technique for the realization of nondeterministic algorithms. Floyd [1] alluded to the reason for this situation: most programming languages do not include features that facilitate backtrack programming. He suggested that programming languages ought to possess mechanisms capable of representing nondeterministic algorithms.

Since the appearance of Floyd's paper, considerable research has been undertaken to add facilities of this kind to new or existing languages. This work has covered a large part of the spectrum of programming languages, from general descriptions with a slant toward Algol-like languages [5,6], to languages for artificial intelligence research [7], and even to Fortran [8]. In all the work cited, features that were added or proposed for backtracking were cast in a framework of recursive functions with additional built-in mechanisms or primitives with which to implement backtracking. That is, the basic procedure mechanism of the proposed languages or language extensions was the traditional recursive function.

This paper presents a general procedure mechanism that includes coroutines as a means for the description and realization of nondeterministic algorithms. The SL5 programming language [9-12] in which this procedure mechanism is implemented is the vehicle used to describe this method and its application to backtrack programming.

*This work was supported by the National Science Foundation under Grant DCR75-01307.

+Author's present address: Department of Computer Science, Yale University, New Haven, Connecticut 06520.

To facilitate comparison with previous work, the eight queens problem [13-15] is used as the example of backtracking throughout this paper. This is a nontrivial problem whose solution is ideally suited to the backtracking strategy, and has frequently been used as an example that can be solved by nondeterministic programming.

2. THE SL5 PROGRAMMING LANGUAGE

SL5 is an expression-oriented language that is structurally similar to BLISS or Algol 68. SL5 is a "typeless" language in the same sense that SNOBOL4 is -- a variable can have a value of any datatype at any time during program execution.

2.1 Control Structures and Signaling

An expression returns a signal, "success" or "failure", as well as a value. The combination of a value and a signal is called the result of the expression. SL5 possesses most of the "modern" control structures, each of which is an expression and returns a result. Control structures are driven by signals rather than by boolean values. For an example, in the expression

$$\underline{\text{if}}\ e_1\ \underline{\text{then}}\ e_2\ \underline{\text{else}}\ e_3$$

e_1 is evaluated first. If the resulting signal is success, e_2 is evaluated. Otherwise, e_3 is evaluated. The result of the if-then-else expression is the result (value and signal) of e_2 or e_3, whichever is evaluated.

Other typical control structures are:

$$\underline{\text{while}}\ e_1\ \underline{\text{do}}\ e_2$$
$$\underline{\text{until}}\ e_1\ \underline{\text{do}}\ e_2$$
$$\underline{\text{repeat}}\ e$$
$$\underline{\text{for}}\ v\ \underline{\text{from}}\ e_1\ \underline{\text{to}}\ e_2\ \underline{\text{do}}\ e_3$$

The while and for expressions behave in the conventional manner. The until expression repeatedly evaluates e_2 until e_1 succeeds. The repeat expression evaluates e repeatedly until a failure signal is returned. Expressions may be grouped together as a single expression using begin ... end or { ... }.

2.2 Procedures

In SL5, procedures and their environments (activation records) are separate source-language data objects. A procedure is "created" by an expression such as

```
gcd := procedure (x, y)
  while x ~= y do
    if x > y then x := x-y else y := y-x;
  succeed x
end;
```

which assigns to *gcd* a procedure that computes the greatest common divisor of its arguments.

The invocation of a procedure in the standard recursive fashion is accomplished using the usual functional notation $f(e_1,e_2, \ldots ,e_n)$, which invokes the procedure that is the current value of the variable f.

Procedure activation may be decomposed into several distinct source-language operations that permit SL5 procedures to be used as coroutines. These operations are the creation of an environment for the execution of the given procedure, the binding of the actual arguments to that environment, and the resumption of the execution of the procedure.

The create expression takes a single argument of datatype procedure, creates an environment for its execution, and returns this environment as its value. For example, the expression

```
e := create f
```

assigns to e an environment for the execution of f.

The with expression is used to bind the actual arguments to an environment. The expression

```
e with (e1,e2, ... ,en)
```

binds the actual arguments, e_1 through e_n, to the environment e.

The execution of a procedure is accomplished by "resuming" it via the resume expression. The expression

```
resume e
```

suspends execution of the current procedure and activates the procedure for which e is an environment.

A procedure usually "returns" a result to its "resumer". This is accomplished by the expressions

```
succeed v
fail v
```

which return v as the value of the procedure and signal either success or failure as indicated. If the procedure is activated by a resume, the result given in succeed or fail is transmitted and becomes the result of the resume expression. The execution of succeed or fail causes the suspension of that environment. If the environment is again resumed, execution proceeds from where it left off. The argument v may be omitted, in which case the null string is assumed.

A label generator illustrates a simple example of coroutine usage:

```
genlab := procedure (n)
  repeat {
    succeed "L" || lpad(n, 3, "0");
    n := n+1
    }
end;
```

An environment for *genlab* generates the next label of the form Lnnn each time it is resumed. The sequence begins at the integer given by the argument. (*lpad* is a built-in procedure that pads n on the left with zeros to form a 3-character string, and || denotes string concatenation.) For example, an expression such as

```
gen := create genlab with 10
```

assigns to *gen* an environment for *genlab* that generates a sequence of labels beginning at L010. To obtain the next label, the execution of the environment is resumed:

```
x := resume gen
```

Notice that the sequence may be restarted by retransmitting the argument, e.g.,

```
gen := gen with 10
```

2.3 Declarations

SL5 has declarations for identifiers that are used to determine only the interpretation and scope of identifiers that appear in a procedure, not their type. The declaration

```
private x
```

declares x to be a private identifier whose value is available only to the procedure in which it is declared; it cannot be examined or modified by any other procedure. Private identifiers are used, for example, when a coroutine must "remember" information from one resumption to the next. Other declarations and the scope of identifiers are described in refs. 9 and 12.

3. BACKTRACKING AND THE EIGHT QUEENS PROBLEM

There are many problems for which an analytic solution is not known, but for which a solution can be constructed by trial and error. A classic example is the eight queens problem, sometimes referred to as the *n-by-n* nonattacking queens problem. The object is to place eight queens on a chess board so that no queen can capture any of the others. One such solution is shown in fig. 1.

There are 92 solutions to this problem, although only 12 are unique.

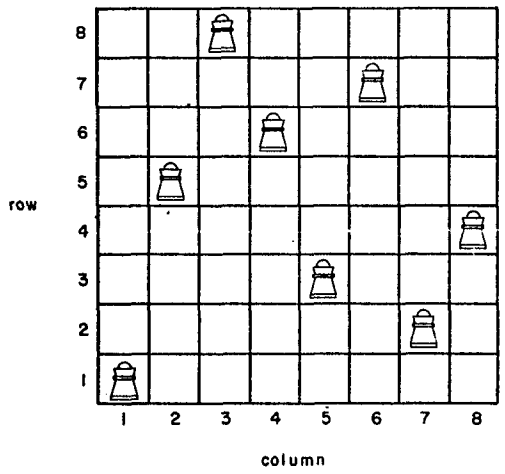

Fig. 1 - A Solution to the Eight Queens Problem

A brute force approach to this problem is to test all the possible configurations of the queens to find the 92 "safe" ones. Although the number of possible configurations can be substantially reduced by observing that only one queen may occupy a given column, the brute force approach requires an impractical amount of computation.

3.1 Backtracking

A better approach for solving this type of problem is to construct a solution one queen at a time rather than testing the validity of every possible configuration. This is called the "backtracking" approach. For example, if the first queen (the leftmost one in fig. 1) is placed on row 1, the second queen can only be placed on rows 3 through 8. Configurations with the first queen on row 1 and the second queen on row 1 or 2 cannot lead to a solution regardless of the positions of queens 3 through 8. Thus only the partial solutions (1,3), (1,4), ..., (1,8) need to be considered when searching for a solution.

The idea in backtracking is to form the k^{th} partial solution $(x_1,x_2,\ldots,x_k)$ and extend it to a k+1st partial solution $(x_1,x_2,\ldots,x_k,x_{k+1})$ by selecting a suitable x_{k+1}. When k+1 is equal to 8, a complete solution has been found. The term backtracking is derived from the action taken when the k^{th} partial solution cannot be extended to a k+1st partial solution. In this case, it is necessary to "backtrack" to the k-1 partial solution and try to compute a different x_k for a k^{th} partial solution. This backtracking step requires that whatever computation was required to form the k^{th} partial solution be undone in order to get back to the k-1 partial solution. This is often called "reversing effects" or "backwards execution". For the eight queens problem, this amounts to freeing the squares on the board covered by the k^{th} queen.

For example, it is easy to place the first five queens to form the partial solution (1,3,5,2,4). But the sixth queen cannot be placed. It is necessary to backtrack to the partial solution (1,3,5,2) and try again. This partial solution can be extended to (1,3,5,2,8) but no further. It is necessary to backtrack all the way to the partial solution (1,3,5), which can then be extended to (1,3,5,7,2,4,6). This backtracking process continues until the solution (1,5,8,6,3,7,2,4) is found, which is shown in fig. 1.

A more formal description of the backtracking strategy is given in ref. 2. A particularly lucid explanation can be found in ref. 16, which describes a method for estimating the efficiency of backtracking programs.

3.2 Realization of the Nondeterministic Algorithm

The usual method for programming the solution to the eight queens problem is to use a procedure that generates all solutions with the first queen on rows 1 to 8 by calling itself recursively to generate all solutions for the second queen in rows 1 to 8, etc. The following procedure, similar to the Pascal solution given in ref. 15, operates in this fashion.

```
generate := procedure (col) private row;
  for row from 1 to 8 do
    if test(row, col) then {
      occupy(row, col);
      x[col] := row;
      if col = 8 then print(x)
                 else generate(col+1);
      release(row, col)
      };
  succeed
end;
```

The details of the board representation are contained in procedures *test*, *occupy*, and *release*. *test*(*row*, *col*) succeeds if the queen in column *col* can be placed on the indicated row. The procedure *occupy*(*row*, *col*) marks as occupied all positions covered by the queen at the position *row*, *col*. *release*(*row*, *col*) reverses the effect of *occupy*; it marks those positions covered by the given queen as free. Possible representations for the actual board are given in refs. 1 and 13-15. *print*(*x*) prints the contents of the solution vector *x*.

The program is started by *generate*(1). A portion of the backtracking in this solution is somewhat obscured by the recursion; it is accomplished implicitly by repeated recursive invocations of *generate*. It is not necessary to use recursion to accomplish the backtracking but it is sometimes used because the only form of procedure available is the recursive function.

The coroutine method, on the other hand, does not require the use of recursion to accomplish the backtracking. The basic approach is to create eight environments for a single procedure; one for each column. Each environment represents one queen. The procedure, called *queen*, attempts to place a queen on the given column beginning with

row 1. If a queen is successfully placed, the procedure suspends its execution and signals success to its resumer. If it is subsequently resumed, it reverses its previous effects, i.e. removes the queen from the row, and tries the next row. If the queen cannot be placed, the procedure fails indicating that backtracking must occur. Subsequent resumption after failure indicates that the process should begin again at row 1.

The eight environments for procedure queen are stored in a vector q. The first step is to create the eight environments for procedure *queen*, each with the proper column number:

```
q := vector(1, 8);
for i from 1 to 8 do
  q[i] := create queen with i;
```

To begin the search for a solution, the execution of the first queen, $q[1]$, is resumed. The second queen is then resumed, and so on. If the resumption of a queen fails, backtracking is indicated. If the i^{th} queen fails, queen $i-1$ must be resumed in order to be repositioned. This is equivalent to queen $i-1$ attempting to find a new $i-1$ partial solution. If the i^{th} queen succeeds, queen $i+1$ is resumed in hopes of extending the i^{th} partial solution. A complete solution has been found when the eighth queen is successfully placed. This entire process can be written as

```
i := 1;
until i > 8 do
  if resume q[i]
    then i := i+1
    else i := i-1;
print(x);
```

The index i is incremented as long as the i^{th} queen is successfully placed, i.e., as long as the extension to the i^{th} partial solution is possible. It is decremented when the i^{th} queen signals failure indicating that the i^{th} partial solution could not be formed.

The procedure queen is as follows.

```
queen := procedure (col) private row;
  repeat {
    for row from 1 to 8 do
      if test(row, col) then {
        occupy(row, col);
        x[col] := row;
        succeed;
        release(row, col)
        };
    fail
    }
end;
```

The expression repeat { ... } is a nonterminating loop.

All 92 solutions can be found by modifying the until loop given above so that after a solution has been found the execution of the eighth queen is again resumed. If the subsequent placement is successful, a second solution is generated. If it fails, the seventh queen must be repositioned. This is equivalent to making a solution fail, after recording it, in order to search for all possible solutions using the backtracking strategy. The process is stopped when the first queen signals failure. This loop can be written as follows.

```
i := 1;
until i = 0 do
  if resume q[i]
    then (if i = 8 then print(x) else i := i+1)
    else i := i-1;
```

Notice that i is not incremented after successful placement of the eighth queen, thus forcing its repositioning at the next resumption. This program can be generalized for n queens by substituting n wherever 8 appears.

The general form is the same for many similar backtracking problems. For example, if the procedures *test*, *occupy*, and *release* are modified to assume rooks instead of queens, the program computes all possible permutations of the integers 1 to n.

4. COMPARISON OF THE METHODS

The major difference between the recursive approach and the coroutine approach is in the control regime used to achieve backtracking. This is illustrated in fig. 2. The left part of fig. 2 shows the control relationship among the eight instantiations of *generate* when a recursive solution has been computed. The relationship is strictly hierarchical: *generate* is written to use recursion in order to "resume" the next queen. The procedure *generate* must include not only the semantics of placing a queen, but is must also contain the backtracking mechanism.

The right part of fig. 2 shows the control relationship among the eight environments for the coroutine solution. In this case, the procedure only needs to know how to place a queen, not about the order in which each environment is resumed. The main program controls the resumption of the coroutines.

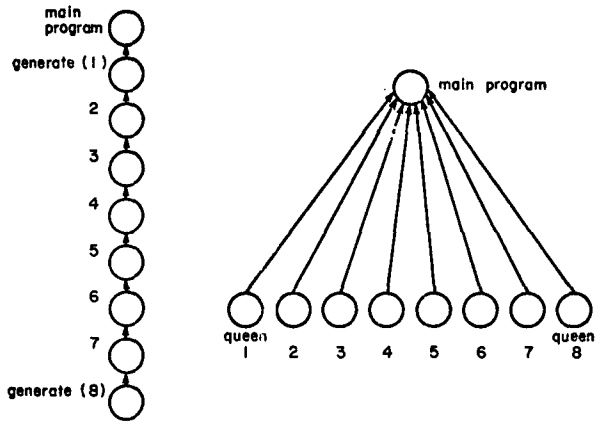

Fig. 2 - Control Regimes among the Eight Queens

5. CONCLUSIONS

The procedure facility of a high-level language is one of the most powerful tools for abstraction available to the programmer. The SL5 mechanism is designed to provide, at the linguistic level, facilities that permit the programmer to implement solutions to backtracking problems in a way that closely parallels the abstract formulation of the problem.

The coroutine approach to backtracking is not limited to SL5. The same idea can be used in other languages that support coroutines, such as Simula 67 [17]. Alternatively, SL5 can be used as a specification language in which to formulate the solutions to backtracking problems. The resulting program can then be used as a guide to an actual implementation in a lower-level language. This is done in the Appendix for the eight queens problem; the SL5 program given in sec. 3.2 is used as a guide for constructing a solution in Fortran.

There are other problems, such as parsing and string pattern matching, that can be solved using backtracking techniques. Unlike the eight queens problem, however, the domain of the search is not known beforehand, but is determined as the search proceeds. Nonetheless, the coroutine approach appears to be applicable to these types of problems. For example, SL5 contains a pattern-matching facility that is based on a coroutine model of pattern matching in SNOBOL4 [18]. The SL5 facility is significantly more general and flexible than the facility in SNOBOL4, and has proven to be easier to implement and to understand than the resursive approach used in SNOBOL4 [19,20].

ACKNOWLEDGEMENT

Significant contributions to SL5 have been made by Dianne E. Britton, Frederick C. Druseikis, and Ralph E. Griswold.

APPENDIX

The following Fortran program computes all 92 solutions to the eight queens problem, and is derived from the SL5 program given in sec. 3.2. The board representation, embodied in test, occupy, and release, can be derived from that given in refs. 13-15.

```
c     main program
      logical queen
      integer row, i
      common /env/ row(8)
c
      i = 1
   30 if (i .le. 0) stop
      if (queen(i)) go to 40
      i = i - 1
      go to 30
c
   40 if (i .eq. 8) go to 50
      i = i + 1
      go to 30
   50 write(6, 100) row
  100 format(8(1x, i1))
      go to 30
      end
```

```
      logical function queen(col)
      integer row, col, j, p(8)
      logical test
      common /env/ row(8)
      data p/8*1/
c
      j = p(col)
      go to (10, 20, 50),j
c
   10 if (row(col) .gt. 8) go to 40
      if (.not. test(row(col), col))
     1 go to 30
      call occupy(row(col), col)
      p(col) = 2
      queen = .true
      return
c
   20 call releas(row(col), col)
   30 row(col) = row(col) + 1
      go to 10
c
   40 p(col) = 3
      queen = .false.
      return
c
   50 row(col) = 1
      go to 10
      end
```

REFERENCES

[1] Robert W. Floyd, Nondeterministic algorithms, J. ACM, vol. 14, October 1967, 636-644.

[2] Solomon W. Golomb and Leonard D. Baumert, Backtrack programming, J. ACM, vol. 12, October 1965, 516-524.

[3] Derrick H. Lehmer, Combinatorial problems with digital computers, Proc. of the Fourth Canadian Math. Congress, 1957, 160-173.

[4] Robert J. Walker, An enumerative technique for a class of combinatorial problems, Proc. of the Symposium on Applied Mathematics, vol. 10, October 1960, 91-94.

[5] Charles J. Prenner, Jay M. Spitzen and Ben Wegbreit, An implementation of backtracking for programming languages, Proc. of the ACM Annual Conference, August 1972, 763-771.

[6] John A. Self, Embedding non-determinism, Software -- Practice and Experience, vol. 5, September 1975, 221-227.

[7] Daniel G. Bobrow and Bertram Raphael, New programming languages for artificial intelligence, Computing Surveys, vol. 6, September 1974, 155-174.

[8] Jacques Cohen and Eileen Carton, Nondeterministic fortran, Computer J., vol. 17, February 1974, 44-51.

[9] Dianne E. Britton, et al., Procedure referencing environments in SL5, Third ACM Symposium on Principles of Programming Languages, January 1976, 185-191.

[10] Ralph E. Griswold and David R. Hanson, An overview of the SL5 programming language, SL5 project document S5LD1a, Dept. of Computer Science, The University of Arizona, Tucson, February 1976.

[11] David R. Hanson, The syntax and semantics of SL5, SL5 project document S5LD2a, Dept. of Computer Science, The University of Arizona, Tucson, April 1976.

[12] David R. Hanson and Ralph E. Griswold, The SL5 procedure mechanism, SL5 project document S5LD4, Dept. of Computer Science, The University of Arizona, Tucson, February 1976.

[13] Ole-Jahn Dahl, Edsger W. Dijkstra and C. A. R. Hoare, Structured Programming, Academic Press, London, 1972, sec. I.17.

[14] Niklaus Wirth, Program development by stepwise refinement, Comm. ACM, vol. 14, April 1971, 221-227.

[15] Niklaus Wirth, Algorithms + Data = Programs, Prentice-Hall, Englewood Cliffs, New Jersey, 1976, sec. 3.5.

[16] Donald E. Knuth, Estimating the efficiency of backtrack programs, Mathematics of Computation, vol. 29, January 1975, 121-139.

[17] Ole-Jahn Dahl, Bjorn Myhrhaug and Kristen Nygaard, The Simula 67 common base language, Norwegian Computing Centre, Oslo, Norway, 1968.

[18] Frederick C. Druseikis and John N. Doyle, A procedural approach to pattern matching in SNOBOL4, Proc. of the ACM Annual Conference, November 1974, 311-317.

[19] Ralph E. Griswold, String scanning in SL5, SL5 project document S5LD5a, Dept. of Computer Science, The University of Arizona, Tucson, June 1976.

[20] Ralph E. Griswold, String analysis and synthesis in SL5, Proc. of the ACM Annual Conference, October 1976.