Backtrack Programming with SIMULA

Keld Helsgaun
Department of Computer Science, Roskilde University Center, P.O. Box 260, 4000 Roskilde, Denmark

Backtrack programming is a technique which has been built into a number of languages, either by defining new primitives or by extension, for the solution of problems of a combinatorial nature. An extension of SIMULA is described, from the user's point of view, which allows all the existing features but also includes facilities for backtrack programming.

INTRODUCTION

Backtrack programming[1–4] is a well-known technique for solving problems of a combinatorial nature. Among its many applications are combinatorial enumeration, syntax analysis and optimization problems. The technique is so widespread that special backtracking primitives have been incorporated into a number of high-level languages. This is especially true in the area of artificial intelligence languages.[5] The generality of the technique is illustrated by various attempts to extend Fortran, Algol and Pascal with such primitives.[6–8]

This paper describes an extension of the language SIMULA with facilities for backtrack programming. The extension is provided in the form of a SIMULA class called BACKTRACKING. All of SIMULA's features are available to the user, including its simulation facilities. The use of class BACKTRACKING in connection with SIMULA's facilities for text handling and simulation is demonstrated through examples of a tutorial nature.

Class BACKTRACKING is described mainly from the user's point of view; its implementation is only sketched. A knowledge of SIMULA is an advantage but not a necessity.

BACKTRACK PROGRAMMING

Backtrack programming is a simple technique for solving combinatorial search problems. The combinatorial problem is perceived as a *multi-stage decision problem* where, at each stage, a choice among a number of alternatives is to be made. The solution of this decision problem can be expressed as a *backtracking algorithm* which in this paper is presented using two special primitives: CHOICE and BACKTRACK.

CHOICE(N) represents a decision point where one of N alternatives is to be chosen.

BACKTRACK is used to signal that the previous choices cannot possibly lead to a solution of the problem.

Execution of a backtracking algorithm results in a series of *choice points* corresponding to the algorithm's invocation of the CHOICE function. At each choice point the first alternative is chosen (CHOICE = 1) and the algorithm continues either until a solution is found, or until BACKTRACK signals that a 'bad' choice has been made. In the latter case, the algorithm *backtracks*, that is to say, re-establishes its state exactly as it was at the most recent choice point and chooses the next untried alternative at this point. If all the choice point's alternatives have been tried, the algorithm backtracks to the previous choice point.

The backtracking technique is perhaps best understood by means of an example. A classical example is the solution of the 8-queens problem. Here the task is to place eight queens on a chess-board so that no queen is attacked by another; that is, so that there is at most one queen in each row, column and diagonal. There are 92 solutions to this problem. One of them is shown in Fig. 1.

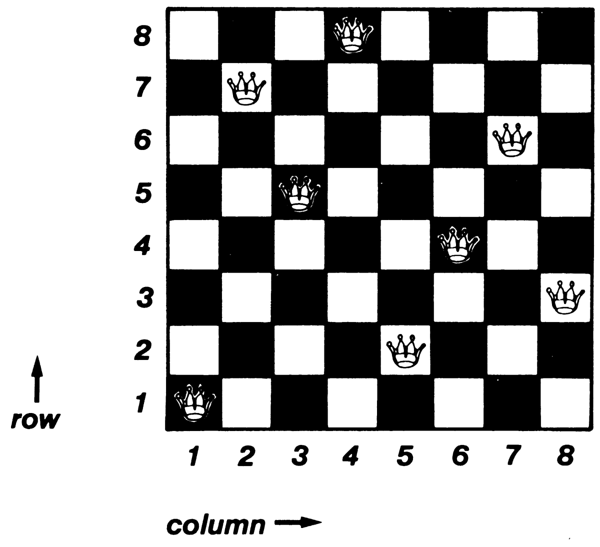

Figure 1. One solution of the 8-queens problem.

It is easy to see that in each solution there has to be one and only one queen per row on the board. Therefore the problem can be formulated as a multi-stage decision problem where, for each of the eight rows, one of the eight possible queen placements may be chosen. Figure 2 shows the corresponding backtracking algorithm written in an Algol notation.

```
for ROW: = 1 step 1 until 8 do
begin
   COL: = CHOICE(8);
   if UNDERATTACK(ROW,COL) then BACKTRACK;
   PLACEQUEEN(ROW,COL);
end;
PRINTSOLUTION;
```

Figure 2. Backtracking algorithm for solving the 8-queens problem.

CCC–0010–4620/84/0027–0151 $04.00

The procedures UNDERATTACK, PLACEQUEEN and PRINTSOLUTION contain the details of the board representation. UNDERATTACK(ROW, COL) is true if the square (ROW, COL) is under attack. PLACEQUEEN(ROW, COL) places a queen on the square (ROW, COL) and marks all squares in the same column and in the two diagonals as being under attack. PRINTSOLUTION prints the solution.

BACKTRACK PROGRAMMING WITH SIMULA

It can be very difficult to use the backtracking technique with programming languages without backtracking primitives. Writing algorithms which explicitly handle their own backtracking, i.e. save information necessary to re-establish previous states, is difficult, tedious and error-prone. By using a language which includes backtracking primitives, the programmer no longer needs to concern himself with this bookkeeping task; he can concentrate on solving the actual problem at hand.

The following describes backtracking primitives implemented as an extension of the general-purpose programming language SIMULA.[9]

SIMULA, itself an extension of Algol 60, additionally offers class and coroutine concepts, reference variables, list handling facilities, discrete-event simulation, and extensive text and input/output capabilities.

The backtracking extension is an enhancement of SIMULA, so that all of SIMULA's facilities are available to the user. Thus fairly advanced applications, for example optimization in connection with simulation, are within the programmer's reach.

The backtracking primitives CHOICE and BACKTRACK are available as two procedures in a SIMULA class called BACKTRACKING. An outline of the class is shown in Fig. 3.

```
class BACKTRACKING;
virtual: procedure FIASCO;
begin
   integer procedure CHOICE(N); integer N; ... ;
   procedure BACKTRACK; ... ;
   procedure FIASCO;;
end;
```

Figure 3

The procedure CHOICE generates succesive integer values from 1 to N. Calling CHOICE returns the value 1. The values 2 to N are returned through subsequent calls of procedure BACKTRACK.

Class BACKTRACKING is normally used as a *prefix* for a block:

```
BACKTRACKING
begin
   <declarations>;
   <backtracking algorithm>;
end;
```

Calling CHOICE causes the block's actual state to be recorded and the value 1 returned. A subsequent call of BACKTRACK causes restoration of the block's state and the next integer in succession to be returned as the value of CHOICE. If the last value, N, has been returned, BACKTRACK refers to the previous call of CHOICE.

In this way BACKTRACK 'rolls' the program back to the last unfinished call of CHOICE. However, input/output operations which have taken place will not be undone.

Calling CHOICE with a non-positive argument is equivalent to calling BACKTRACK.

The BACKTRACKING block defines a kind of context for the procedures CHOICE and BACKTRACK. Only variables in the BACKTRACKING block itself will be remembered by CHOICE, and therefore re-established by BACKTRACK. Variables outside the block are not touched. Such *global* variables make possible communication between calls of CHOICE and BACKTRACK and are absolutely necessary for solving optimization problems, as shown later in this paper.

If in calling BACKTRACK, no unfinished call of CHOICE exists, the procedure FIASCO is automatically called after which the BACKTRACKING block is terminated. FIASCO is predefined with an empty procedure body but, since the procedure is specified *virtual*, the user may program its desired effect.

A complete BACKTRACKING program which solves the 8-queens problem is shown in Fig. 4. The integer array Q, initially zero-filled, is used to record the current placement of queens: Q(COL) contains the row number of the queen in column COL. If the value is zero, no queen has yet been placed in that column. The two Boolean arrays UP and DOWN are used to determine if a given square is diagonally under attack from queens already placed. The algorithm exploits the fact that the difference between row number and column number uniquely determines an upward diagonal and their sum likewise determines a downward diagonal.

```
 1. BACKTRACKING
 2. begin
 3.  integer array Q(1:8);
 4.  Boolean array UP(-7:7), DOWN(2:16);
 5.  integer ROW, COL;
 6.  for ROW: = 1 step 1 until 8 do
 7.  begin
 8.    COL: = CHOICE(8);
 9.    if Q(COL)< >0 or UP(ROW-COL) or DOWN(ROW + COL) then BACKTRACK;
10.    Q(COL): = ROW; UP(ROW-COL): = DOWN(ROW + COL): = true;
11.  end;
12.  for COL: = 1 step 1 until 8 do
13.  begin OUTINT(Q(COL),4); OUTINT(COL,2); end;
14. end;
```

Figure 4. BACKTRACKING program for solving the 8-queens problem.

The program produced the following output:

```
1 1   7 2   5 3   8 4   2 5   4 6   6 7   3 8
```

corresponding to the solution pictured in Fig. 1.

The program finds only one of the 92 solutions of the problem. If all the solutions are desired, one can just add a BACKTRACK call at the end of the program (between line 13 and 14). With this addition the program will continue until all CHOICE-possibilities are exhausted. Note that calling BACKTRACK does not 'recall' previously outputted solutions.

BACKTRACKING AND TEXT ANALYSIS

In a BACKTRACKING program all of SIMULA's facilities for text handling are available. The following example shows how these facilities can be used with regard to syntax analysis. In addition, the example demonstrates the use of recursion in connection with backtracking.

Suppose we have a language S defined by the following syntax rules (BNF notation):

```
<S> ::= 2<S> | <A>3
<A> ::= 1<A>1 | 1<A>2 | 1
```

Here 1, 2 and 3 are the terminal symbols and S and A are the non-terminal symbols. S is the start symbol.

Using class BACKTRACKING it is easy to write a parser program that can decide whether a given text string is or is not a sentence in the language. Figure 5 shows such a program.

The program performs top-down parsing. For each of the non-terminal symbols, S and A, in the language there exists a corresponding procedure which reads a character sequence from INPUT and attempts to find a matching syntax rule. Procedure CHOICE is used to choose such a rule from among the possible alternatives.

The procedure CHECK (lines 19–20) decides whether the next character of INPUT is equal to the terminal symbol in the rule under consideration. If this is not the case, BACKTRACK is called choosing another rule.

```
 1. BACKTRACKING
 2. begin
 3.  procedure FIASCO; OUTTEXT("SYNTAX ERROR");

 4.  text INPUT;
 5.  integer RULE;

 6.  procedure S;
 7.  begin
 8.    RULE: = CHOICE(2);
 9.    if RULE = 1 then begin CHECK('2'); S; end else
10.    if RULE = 2 then begin A; CHECK('3'); end;
11.  end;

12.  procedure A;
13.  begin
14.    RULE: = CHOICE(3);
15.    if RULE = 1 then begin CHECK('1'); A; CHECK('1'); end else
16.    if RULE = 2 then begin CHECK('1'); A; CHECK('2'); end else
17.    if RULE = 3 then CHECK('1');
18.  end;

19.  procedure CHECK(C); character C;
20.  if INPUT.GETCHAR < >C then BACKTRACK;

21.  INPUT:-INTEXT(80);
22.  S;
23.  CHECK(' ');
24.  OUTTEXT("NO ERRORS");
25. end;
```

Figure 5. BACKTRACKING program for the syntax analysis example.

The analysis begins with the calling of the procedure S (line 22). It is assumed that the text string, INPUT, is at most 80 characters in length and ends with a space.

If the text string obeys the syntax rules, the message 'NO ERRORS' is written (line 24). If the syntax rules are not obeyed, the virtual procedure FIASCO (line 3) is called and the program stops with the message 'SYNTAX ERROR'.

A run of the program with the following input data

22221111122223

produced the correct response: 'NO ERRORS'.

BACKTRACKING AND SIMULATION

SIMULA's class concept allows for the use of coroutines. A coroutine is represented as an object of some class which, with the help of RESUME-operations, co-operates with objects of the same or of another class. Coroutines are well suited for creating the illusion of parallelism, a facility which is useful in simulation.

SIMULA contains a built-in class called SIMULATION which simplifies discrete-event simulation. In principle this is a package constructed with the coroutine facilities native to SIMULA. The active objects, or *processes*, in a simulation are represented as coroutines which operate in quasi-parallel under control of a scheduling algorithm. Incorporating backtracking primitives in SIMULA makes possible the interesting combination of backtracking and simulation.

The following example demonstrates how backtracking can be used in connection with discrete-event simulation for solving a flow-shop scheduling problem.

A number of different jobs are to be processed by a series of machines where each machine can process only one job at a time. Each job must be processed by all machines in the same fixed sequence. Jobs may pass each other, that is the processing sequence may vary from machine to machine. The problem is to find a processing schedule which minimizes the *total elapsed time*, which is the time from the start of the first job until all jobs have been processed by all machines.

Assume, for example, that there are 5 jobs and 4 machines, and that each job's processing time at each machine is as shown in Table 1.

Table 1. Table of processing times (in hours)

	Machine			
Job	1	2	3	4
1	7	4	6	3
2	9	5	1	8
3	5	1	9	7
4	6	2	3	5
5	10	3	2	4

Furthermore, assume that each job is to be processed by machines 1 to 4 in that order. The time for transporting jobs between machines is assumed to be negligible. What then is the best start time for each job at each machine so that the total elapsed time is as small as possible?

The logical parallelism inherent in this problem suggests using simulation, whereas its combinatorial nature points to using backtracking. The backtracking extension of SIMULA allows a synthesis of both approaches. As shown in Fig. 6 the insertion of a SIMULATION block in a BACKTRACKING block makes this possible.

```
BACKTRACKING
begin
  ┌──────────────┐
  │SIMULATION    │
  │begin         │
  │   .          │
  │   .          │
  │   .          │
  │end;          │
  └──────────────┘
end;
```

Figure 6

There are no constraints regarding the use of procedures CHOICE and BACKTRACK; they can be called at any point during the simulation. Calling BACKTRACK 'rolls back' the simulation to the last unfinished call of CHOICE.

A detailed discussion of a program solving the flow-shop problem—using a top-down approach—follows. The program's outermost block levels are sketched in Fig. 7.

```
begin
  integer JOBS, MACHINES;
  <input JOBS and MACHINES>;
  begin
    real array PROCESSTIME, BESTSTARTTIME(1:JOBS,1:MACHINES);
    real MINFINISHTIME;
    <input PROCESSTIME>;
    ┌──────────────┐
    │BACKTRACKING  │
    │begin         │
    │   .          │
    │   .          │
    │   .          │
    │end;          │
    └──────────────┘
    <print solution: MINFINISHTIME and BESTSTARTTIME>;
  end;
end;
```

Figure 7. Outermost block levels in the flow-shop program.

The input to the program consists of the number of jobs to be processed, JOBS, the number of machines involved, MACHINES, and a table, PROCESSING-TIME, containing each job's processing time at each machine (see Table 1).

The output consists of the minimal total processing time, MINFINISHTIME, and a table, BESTSTART-TIME, containing the optimal start time of each job at each machine. Notice that these variables must be global with respect to the BACKTRACKING block since their values must remain unchanged by eventual calls of BACKTRACK.

The BACKTRACKING block solves the actual optimization problem and is sketched in Fig. 8.

```
BACKTRACKING
begin
  real array STARTTIME(1:JOBS,1:MACHINES);
  real FINISHTIME;
  MINFINISHTIME: = <large value>;
  ┌──────────────┐
  │SIMULATION    │
  │begin         │
  │   .          │
  │   .          │
  │   .          │
  │end;          │
  └──────────────┘
  MINFINISHTIME: = FINISHTIME;
  <set BESTSTARTTIME to STARTTIME>;
  BACKTRACK;
end;
```

Figure 8. BACKTRACKING block of the flow-shop program.

The BACKTRACKING block contains an inner SIMULATION block which, using both simulation and backtracking, attempts to determine a schedule where the total processing time, FINISHTIME, is less than the shortest processing time previously found, MINFINISH-TIME. Before entering the SIMULATION block, MINFINISHTIME is assigned so large a value that at least one schedule is guaranteed.

Each time the SIMULATION block finds a better schedule, MINFINISHTIME is set to FINISHTIME and BESTSTARTTIME is set to STARTTIME. STARTTIME is the scheduling table under consideration and contains the start time of each job at each machine. BACKTRACK is then called in an attempt to find yet a better schedule. The BACKTRACKING block is exited when all possibilities have been tried. Figure 9 shows a sketch of the SIMULATION block.

```
SIMULATION
begin
  ref(HEAD) array QUEUE(1:MACHINES);
  Boolean array BUSY(1:MACHINES);
  integer JOBSDONE;
  ┌──────────────────────────────────────────┐
  │PROCESS class JOB(NUMBER); integer NUMBER;│
  │begin                                     │
  │   .                                      │
  │   .                                      │
  │   .                                      │
  │end;                                      │
  └──────────────────────────────────────────┘
  for M: = 1 step 1 until MACHINES do QUEUE(M):-new HEAD;
  for J: = 1 step 1 until JOBS do activate new JOB(J) at 0;
  HOLD(MINFINISHTIME);
  if JOBSDONE<JOBS then BACKTRACK;
end;
```

Figure 9. SIMULATION block of the flow-shop program.

Jobs are the active components of the system and are defined in a subclass of PROCESS with the name JOB. Each job tries to 'schedule' itself at each machine in such a way that the total elapsed time of all jobs, FINISH-TIME, is less than the provisional minimum, MINFIN-ISHTIME.

Each machine M has a queue, QUEUE(M), where jobs can wait. When a job is processed at M, BUSY(M) is set to true.

The main program starts by creating empty queues and activating numbered JOB-objects. It then suspends itself and waits MINFINISHTIME time units. If not all jobs are finished (JOBSDONE < JOBS) when the main program again becomes active, procedure BACKTRACK is called in a new attempt to find a schedule where the total elapsed time is less than MINFINISHTIME. Figure 10 shows the first version of class JOB.

```
 1. PROCESS class JOB(NUMBER); integer NUMBER;
 2. begin
 3.   integer MACHINE;
 4.   for MACHINE: = 1 step 1 until MACHINES do
 5.   begin
 6.     INTO(QUEUE(MACHINE));
 7.     if BUSY(MACHINE) then PASSIVATE;
 8.     while CHOICE(2) = 2 do
 9.       begin activate SUC; PASSIVATE; end;
10.     FINISHTIME: = TIME + PROCESSTIME(NUMBER,MACHINE);
11.     if FINISHTIME> = MINFINISHTIME then BACKTRACK;
12.     OUT;
13.     STARTTIME(NUMBER,MACHINE): = TIME;
14.     BUSY(MACHINE): = true;
15.     HOLD(PROCESSTIME(NUMBER,MACHINE));
16.     BUSY(MACHINE): = false;
17.     activate QUEUE(MACHINE).FIRST;
18.   end;
19.   JOBSDONE: = JOBSDONE + 1;
20. end;
```

Figure 10. Class JOB of the flow-shop program.

Class JOB describes the processing of each job. The integer attribute MACHINE (line 3) denotes the number of the current machine where processing is to take place. The algorithm should be fairly self-explanatory. In line 8 a choice is made between two alternatives, namely (1) the job is processed immediately at the idle machine, or (2) the job waits and allows other jobs to go first. Line 11 states that a simulation which would take longer than the current MINFINISHTIME will be stopped by calling BACKTRACK.

The program is not very efficient. Even with only 5 jobs and 4 machines it can happen that the program must examine more than 600 million different combinations before finding the optimal schedule.

Fortunately, the program can easily be made more efficient. In the first place, it is unnecessary for the very last job to wait for other jobs to pass. Secondly, it is easy to show that one need only consider schedules where the processing sequences at the first and second machines are the same, and where the processing sequences at the last and next-to-last machines are the same.[10] To implement these short-cuts lines 8–9 in class JOB are replaced by the following:

```
if MACHINE< >2 and MACHINE< >MACHINES then
  while this JOB = / = LASTJOB and CHOICE(2) = 2 do
    begin activate SUC; PASSIVATE; end;
```

where LASTJOB refers to the last job of the sequence.

These few changes greatly improve the program's efficiency. In the case of the 5 jobs and 4 machines at most $(5!)^2 = 14\,400$ different schedules are examined.

Branch-and-bound techniques yield further improvements. If a lower bound for the total elapsed time can be calculated from knowledge of the first choice and if this boundary is greater than the provisional minimum, then there is no reason to continue with this choice sequence.

This method is implemented in the complete flow-shop program in Fig. 11. The program maintains two tables, JSUM and MSUM, where JSUM(J) and MSUM(M) are, respectively, the sum of job J's and machine M's remaining processing times. The program prunes as described above when there is at least one job J such that TIME + JSUM(J) is greater than or equal to MINFINISHTIME (line 38 and lines 45–46), or when there is at least one machine M such that TIME + MSUM(M) is greater than or equal to MINFINISHTIME (lines 35–36).

```
 1. begin
 2.   integer JOBS, MACHINES;
 3.   JOBS: = ININT; MACHINES: = ININT;
 4.   begin
 5.     real array PROCESSTIME, BESTSTARTTIME(1:JOBS,1:MACHINES);
 6.     real MINFINISHTIME;
 7.     integer J,M;
 8.     for J: = 1 step 1 until JOBS do
 9.       for M: = 1 step 1 until MACHINES do
10.         PROCESSTIME(J,M): = INREAL;
11.     BACKTRACKING
12.     begin
13.       real array STARTTIME(1:JOBS,1:MACHINES);
14.       real FINISHTIME;
15.       MINFINISHTIME: = 1000000;
16.       SIMULATION
17.       begin
18.         ref(HEAD) array QUEUE(1:MACHINES);
19.         Boolean array BUSY(1:MACHINES);
20.         real array MSUM(1:MACHINES), JSUM(1:JOBS);
21.         ref(JOB) LASTJOB;
22.         PROCESS class JOB(NUMBER); integer NUMBER;
23.         begin
24.           integer MACHINE;
25.           if NUMBER = JOBS then LASTJOB:-this JOB;
26.           for MACHINE: = 1 step 1 until MACHINES do
27.           begin
28.             INTO(QUEUE(MACHINE));
29.             if BUSY(MACHINE) then PASSIVATE;
30.             if MACHINE< >2 and MACHINE< >MACHINES then
31.               while this JOB = / = LASTJOB and CHOICE(2) = 2 do
32.                 begin activate SUC; PASSIVATE; end;
33.             FINISHTIME: = TIME + PROCESSTIME(NUMBER,MACHINE);
34.             MSUM(MACHINE): = MSUM(MACHINE)-PROCESSTIME(NUMBER,MACHINE);
35.             for M: = MACHINE,M + 1 while M< = MACHINES and not BUSY(M) do
36.               if FINISHTIME + MSUM(M) > = MINFINISHTIME then BACKTRACK;
37.             JSUM(NUMBER): = JSUM(NUMBER)-PROCESSTIME(NUMBER,MACHINE);
38.             if FINISHTIME + JSUM(NUMBER) > = MINFINISHTIME then BACKTRACK;
39.             if this JOB = = LASTJOB and PRED = / = none then LASTJOB:-PRED;
40.             OUT;
41.             STARTTIME(NUMBER,MACHINE): = TIME;
42.             BUSY(MACHINE): = true;
43.             HOLD(PROCESSTIME(NUMBER,MACHINE));
44.             BUSY(MACHINE): = false;
45.             for J: = 1 step 1 until JOBS do
46.               if TIME + JSUM(J) > = MINFINISHTIME then BACKTRACK;
47.             activate QUEUE(MACHINE).FIRST;
48.           end;
49.         end * * * JOB * * *;
50.         for M: = 1 step 1 until MACHINES do QUEUE(M):-new HEAD;
51.         for M: = 1 step 1 until MACHINES do
52.           for J: = 1 step 1 until JOBS do
53.             begin
54.               MSUM(M): = MSUM(M) + PROCESSTIME(J,M);
55.               JSUM(J): = JSUM(J) + PROCESSTIME(J,M);
56.             end;
57.         for J: = 1 step 1 until JOBS do activate new JOB(J) at 0;
58.         HOLD(MINFINISHTIME);
59.       end;
60.       MINFINISHTIME: = FINISHTIME;
61.       for J: = 1 step 1 until JOBS do
62.         for M: = 1 step 1 until MACHINES do
63.           BESTSTARTTIME(J,M): = STARTTIME(J,M);
64.       BACKTRACK;
65.     end;
66.     OUTTEXT("MINIMUM ELAPSED TIME = ");
67.     OUTFIX(MINFINISHTIME,2,6); OUTIMAGE; OUTIMAGE;
68.     OUTTEXT("START TIME:"); OUTIMAGE;
69.     OUTTEXT("JOB/MACHINE"); OUTIMAGE;
70.     for M: = 1 step 1 until MACHINES do OUTINT(M,7); OUTIMAGE;
71.     for J: = 1 step 1 until JOBS do
72.     begin
73.       OUTINT(J,2);
74.       for M: = 1 step 1 until MACHINES do OUTFIX(BESTSTARTTIME(J,M),2,7);
75.       OUTIMAGE;
76.     end;
77.   end;
78. end;
```

Figure 11. The complete optimized flow-shop program.

Given the input data in Table 1 the program produced the correct output in Table 2.

Table 2

```
MINIMUM ELAPSED TIME = 46.00
START TIME:
JOB/MACHINE
      1        2        3        4
1   20.00    27.00    31.00    37.00
2    0.00     9.00    14.00    15.00
3   15.00    20.00    21.00    30.00
4    9.00    15.00    17.00    23.00
5   27.00    37.00    40.00    42.00
```

Even though the pruning strategy is relatively simple the program's efficiency is quite satisfactory. The example data gave rise to only 340 calls of procedure BACKTRACK.

This example has illustrated the use of class BACKTRACKING in connection with discrete-event simulation. It is also possible to apply the class to combined continuous and discrete simulation, for example with DISCO, a SIMULA-class designed for combined simulation.[11] Used in this way class BACKTRACKING provides a tool for solving many industrial planning problems. Typically such problems require combinatorial decisions within the context of both continuous and discrete processes. Advanced applications of this type are interesting but beyond the scope of this introductory article.

IMPLEMENTATION

If one rules out compiler modifications, there are two methods of implementing backtracking mechanisms:

(1) By means of a preprocessor which can convert a backtracking program to a program written in one of the conventional languages, such as Algol or Fortran.[6,7]

(2) By means of special procedures for saving and restoring the program state at CHOICE and BACKTRACK points.[12]

The present implementation of the SIMULA-class BACKTRACK uses the second method. The procedures CHOICE and BACKTRACK are written in assembler language. They are sketched in Fig. 12 in a kind of pseudo-SIMULA.

```
integer procedure CHOICE(N); integer N;
begin
   integer CHOSEN;

   if N<1 then BACKTRACK;
   if N>1 then PUSH;
NEXT:
   CHOICE: = CHOSEN: = CHOSEN + 1;
   if N>1 and CHOSEN = N then POP;
end;
```

```
procedure BACKTRACK;
begin
   RESTORE;
   goto NEXT;
end;
```

Figure 12

CHOICE and BACKTRACK are based upon three auxiliary procedures: PUSH, RESTORE and POP. Procedure PUSH saves the program's state at the top of a stack. Only information having to do with the user's BACKTRACKING block will be saved. Procedure RESTORE restores the program's state using the element at the top of the stack. Procedure POP removes the top stack element.

The number of elements in the stack at any given time is equal to the number of unfinished CHOICE-calls. In order to provide quick access to the stack, space is allocated in internal storage. This allocation is in most cases sufficient to contain all stack elements. If, however, the allocation is insufficient, backing storage is used, in such a way that as much as possible of the uppermost portion of the stack is kept internally, while the remaining portion resides on the backing storage.

The author has implemented the system on a UNIVAC 1100 machine with the assembler portion taking only about 400 machine instructions. The implementation assumes a detailed knowledge of SIMULA's run-time system. The UNIVAC 1100 SIMULA runtime-system closely follows the guidelines found in the *SIMULA Implementation Guide*.[13] As long as these guidelines have been followed, the same principles apply when implementing class BACKTRACKING on other machines with SIMULA.

The procedures CHOICE and BACKTRACK are available as two external library procedures in class BACKTRACKING shown in its entirety in Fig. 13.

```
class BACKTRACKING;
virtual: procedure FIASCO;
begin
   external library integer procedure CHOICE(N); integer N;;
   external library procedure BACKTRACK;;
   external library procedure CUT;;
   external library integer procedure NEXTCHOICE;;
   external library procedure INITIATE;;
   external library procedure TERMINATE;;
   procedure FIASCO;;

   INITIATE;
   if CHOICE(2) = 2 then begin FIASCO; goto EXIT; end;
   inner;
EXIT:
   TERMINATE;
end;
```

Figure 13

The first action of the class body is the calling of the procedure INITIATE which does preparatory tasks such as file assignment and garbage collection. The CHOICE-call in the next line returns the value 1 causing the user-defined portion (inner) of the BACKTRACKING block to be executed. If all the user's CHOICE-possibilities are exhausted and BACKTRACK is called, the procedure FIASCO is invoked. FIASCO has an empty body. However, since the procedure is virtual, the user may define its desired effect. Before exiting the BACKTRACKING block, the procedure TERMINATE is called which empties the stack and releases the file.

The class contains two additional user-procedures, CUT and NEXTCHOICE, which in certain cases may be used to reduce the program's execution time.

Procedure CUT deletes the last CHOICE-call and causes a BACKTRACK to the prior CHOICE-call.

Procedure NEXTCHOICE immediately returns the next value of the most recent CHOICE-call; but (unlike BACKTRACK) does not restore the program's state. If however, the most recent CHOICE-call is finished, calling NEXTCHOICE is equivalent to calling BACKTRACK. For example, consider the program fragment

```
I: = CHOICE(N);
if B(I) then BACKTRACK;
```

As long as an evaluation of the condition B(I) has no side-effects, it is unnecessary to restore the program state by calling BACKTRACK. Instead the procedure NEXTCHOICE can be used as follows:

```
I: = CHOICE(N);
while B(I) do I: = NEXTCHOICE;
```

An example using these two procedures is given in the 8-queens program shown in Fig. 14.

```
for ROW: = 1 step 1 until 8 do
begin
   COL: = CHOICE(8);
   while UNDERATTACK(ROW,COL) do COL: = NEXTCHOICE;
   PLACEQUEEN(ROW,COL);
end;
PRINTSOLUTION;
CUT;
```

Figure 14

However, the best way to increase the efficiency of a backtracking program is to reduce the number of nodes in the search tree. Thus, it is important to prune the tree using BACKTRACK as much as possible.

Of secondary importance is the time spent in processing each node of the tree. This processing time consists of two elements, namely the user-program's own computation time and the administration time (overhead) involved in calling procedures CHOICE and BACKTRACK (saving and restoring the program state). The efficiency of the actual implementation is such that in most cases the user-program's own computation time dominates.

Table 3 reflects the performance of class BACKTRACKING in the examples of this paper. Total overhead and total execution time are measured in cpu-seconds on a UNIVAC 1100/82. Each call of CHOICE or BACKTRACK contributes approximately the same amount of overhead, on the average 50 microseconds.

Practical experience with class BACKTRACKING has shown that it is a fairly effective tool, not only with respect to execution time, but also regarding the programming effort.

The actual implementation is quite simple and, as shown in the examples, very general. The procedures CHOICE and BACKTRACK can be called almost anywhere in a SIMULA program.

There are, however, a few restrictions regarding the use of class BACKTRACKING:

Table 3. Performance of class BACKTRACKING in the examples

Example	Number of CHOICE-calls	Number of BACKTRACK-calls	Overhead time	Execution time
8-queens problem (all 92 solutions)	1966	13 756	0.614	0.863
8-queens problem (all 92 solutions, using NEXTCHOICE and CUT)	1966	1742	0.155	0.404
Syntax analysis (Figure 5)	29	45	0.005	0.013
Flow-shop (Figure 11)	340	340	0.054	0.190

(1) The number of unfinished CHOICE-calls must not exceed 1000.
(2) Only one non-terminated BACKTRACKING block is allowed at any time.
(3) Global references outside the BACKTRACKING block must not reference objects created by the BACKTRACKING block.

Violation of the first two restrictions causes the program to terminate with an explanatory error message. On the other hand, a violation of the third restriction will not automatically be detected. In this case there is a risk of a mystifying run-time error since the object pointed to by the global reference-variable might have been removed due to backtracking. The restriction is in itself logical, but unfortunately its violation will not be discovered immediately. This is the price for a simple implementation.

The second restriction, concerning the system's generality, is also a consequence of the simplicity of the implementation. Lindstrom[8] has proposed an extension of Pascal which allows for *concurrent* backtracking systems. In this way a number of backtracking systems working in quasi-parallel can co-operate in solving a given problem. An efficient implementation of such a facility seems fairly difficult; the author has no knowledge of any existing implementations.

CONCLUSIONS

The SIMULA-class BACKTRACKING is not only a didactic aid for demonstrating non-deterministic programming using backtracking techniques, but also an effective tool for solving practical problems of a combinatorial nature. In spite of a simple implementation, the class allows rather advanced applications, especially in connection with simulation.

The class is available for the UNIVAC 1100 machine series. Implementations on other machines are considered.

The backtracking software as implemented for UNIVAC SIMULA may be obtained at a nominal cost by writing to the author.

REFERENCES

1. J. R. Bitner and E. M. Reingold, Backtrack programming techniques. *Commun. ACM* **18**(11), 651–656 (1975).
2. J. Cohen, Non-deterministic algorithms. *Computing Surveys* **11**(2), 79–94 (1979).
3. R. W. Floyd, Nondeterministic algorithms. *Journal ACM* **14**(4), 636–644 (1967).
4. S. W. Golomb and L. D. Baumert, Backtrack programming. *Journal ACM* **12**(4), 516–524 (1965).
5. O. G. Borrow and B. Raphael, New programming languages for artificial intelligence research. *Computing Surveys* **6**(3), 155–174 (1974).
6. J. Cohen and E. Carton, Non-deterministic Fortran. *The Computer Journal* **17**(1), 44–51 (1974).
7. P. Johansen, Non-deterministic programming. *BIT* **7**, 289–304 (1967).
8. G. Lindstrom, Backtracking in a generalized control setting. *ACM Trans. Prog. Lang. Sys.* **1**(1), 8–26 (1979).
9. G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug and K. Nygaard, *SIMULA **BEGIN***. Studentlitteratur, Lund (1973).
10. R. W. Conway, W. L. Maxwell and L. W. Miller, *Theory of Scheduling*, pp. 80–81, Addison Wesley, Massachusetts (1967).
11. K. Helsgaun, DISCO—a SIMULA-based language for combined continuous and discrete simulation. *SIMULATION* **35**(1), 1–12 (1980).
12. J. A. Self, Embedding non-determinism. *Software—Practice and Experience* **5**, 221–227 (1975).
13. O.-J. Dahl and B. Myhrhaug, *SIMULA implementation guide*. Norwegian Computing Center, Publication S-47, Oslo (1973).