PRL

TECHNICAL NOTE

TN-85-46

SOLUTIONS OF SEVERAL SEARCH PROBLEMS BASED ON THE BACK-TRACKING ALGORITHM

BY
D.P. AHALPARA

July 1985

COMPUTER CENTRE
PHYSICAL RESEARCH LABORATORY
NAVRANGPURA
AHMEDABAD- 380 009

DOCUMENT CONTROL AND DATA SHEET

1. REPORT NO. : PRL : TN-85-46	
2. TITLE AND SUBTITLE : "Solutions of Several Search Problems based on the Backtracking Algorithm"	3. REPORT DATE : July 29, 1985 4. TYPE OF REPORT : Technical Note 5. PAGES : 43 6. PRICE : Unpriced 7. NO. OF REFERENCES : 19
8. AUTHORS(S) : Dr. D.P.Ahalpara	
9. a) PURPOSE : The note illustrates the usage of Backtracking Algorithm in carrying out a systematic search while solving a variety of search problems of diverse nature. b) USEFUL FOR : Provides a guideline for solving any given search problem belonging to the GPS (General Problem Solving) category.	
10. ORIGINATING UNITS/DIVISIONS AND ADDRESS : Computer Center, Physical Research Laboratory, Navrangpura, Ahmedabad - 380 009	
11. SPONSORING AGENCY : Nil	
12. ABSTRACT : A suite of programs in PASCAL language have been developed to solve problems that require a tree of nodes to be searched in a systematic way. The common theme used in solving these otherwise diverse looking problems is the Backtracking Algorithm which simulates the "thinking" process.	
13. KEYWORDS : GPS(General Problem Solving), Popular Puzzles, Backtracking, Recursion, Systematic Search process, Tree structure.	
14. DISTRIBUTION STATEMENT : Distribution Unlimited.	
15. SECURITY CLASSIFICATION : Unclassified	

SOLUTIONS OF SEVERAL SEARCH PROBLEMS BASED ON THE BACKTRACKING ALGORITHM

D.P.AHALPARA
Cmputer Centre
Physical Research Laboratory
Ahmedabad

INDEX

ABSTRACT

In the present note we consider a problem solving task known as GPS (General Problem Solving). These problems ingeneral do not have analytical solutions. starting from the given initial state of a problem the task ammounts to finding out the path(s) that lead to either a given goal state or to those goal states distinguished by a predicate. The successive "Moves" which lead to new states from one state to the next form typically into a tree structure which ingeneral would grow quite rapidly with the increase in the deapth of the tree. Rather than following a fixed and static pattern of computation, the algorithm which would solve a typical problem belonging to the GPS category is expected to employ a RECURSIVE search process that gradually builds up and prunes through the entire tree of subtasks at each level of the task.

We have used the Backtracking algorithm to solve such problems. The algorithm employs a look-ahead strategy at each level in a recursive way. A variety of otherwise diverse looking problems are solved by this algorithm using the same general problem solving scheme. For each problem an interactive package in PASCAL language is developed employing the backtracking algorithm. It is intersting to note that each package is distinctly seperable into following 2 parts :

1. The tree search algorithm which prunes through the entire tree in a systematic way.

2. The task oriented part which is specific to each problem and ammounts to initializing the problem and specifying the task (i.e. the information about the initial and the goal state).

With small modification in the algorithm (corresponding to part (1) above) it is possible to selectively generate all or one of the follwoing 3 kinds of solutions :

1. Any ONE Solution

2. ALL the possible Solutions

3. The OPTIMAL Solution having the least path length

1.0 INTRODUCTION

A variety of problems and popular puzzles are based on a common theme known as "General Problem Solving" (GPS) (Ref. 1-3,20). These problems ingeneral defy any analytical method for finding out their solutions. It would therefore be necessary to develop an algorithm which carries out a systematic search process in the 'Move Space' of the problem. (We may note that for many of the problems the search process becomes quite involved one and it is then necessary to look for appropriate heuristic scheme to reasonably cut down the involved search. The word heuristic is derived from the well known Greek word EUREKA. Heuristic ingeneral means a rule of thumb that serves to help discover.)

A well known problem belonging to the GPS category is the "8-Queens" problem in which 8 Queens are to be placed on a Chess Board such that no two queens can capture each other. The problem was investigated way back in 1950 by the well known mathmatician C.F.Gauss who gave its partial solution. Another popular example is that of a "Treasure Hunt" in a mesh of raods and walls where the task is to find out the shortest path from a given initial square to the final one. The "Tower of Hanoi" problem invites to rearrange a stack of disks of varying size, arranged in a discending order, from one peg to another using one more auxilliary peg. The movement of the disks is however restricted in that at no time could you place a bigger disk on top of a smaller disk. (The legend has it that long time ago in an Indian temple this task was assigned to several monks for a stack of 63 disks and it was believed that our cosmos would come to an end as soon as the monks would accomplish their task !). The classical "Route Finding problem", i.e. that of finding out the shortest route between two given nodes in a network, or of finding out the canonical route which optimally traverses through all the nodes of the network (without revisiting any node), also falls under the category of GPS. In the "Knight's tour" problem one considers an n*n Chess Board. Starting from the given initial square the path made up of a knight's successive moves is to be found out such that the entire board would be swept through by visiting each and every square - only once. The well known "M-Partition and Knapsack" problems of integer programming look for all possible subsets of integer numbers (from an apriorily given set of integer numbers) which satisfy certain given constraints. The popular puzzle of "3 Water Jugs" requires to divide the given 8 litre water (filled initially in an 8 litre jug) into 2 equal parts with the use of 2 other empty jugs with capacities of 3 and 5 litres each.

These otherwise diverse looking problems have a common built-in feature that by and large it is rather difficult to look for a fixed strategy of computation which can DIRECTLY lead to the solution one is looking for. In absence of an analytical solution a Computer-Algorithmic-Approach is therefore quite relevent.

An interesting algorithm, which employs a systematic (and possibly a heuristic) search method through a tree of successive moves, is the Backtracking algorithm (Ref. 1-3). The algorithm is known to be quite useful for solving one-person game type of problems falling under the GPS category. Given a GPS problem in which initial and goal states are defined, the trial and error search process employed in the algoritnm goes as follows. At each stage of the search the task is decomposed into partial subtasks. Each subtask is then recursively scanned through till either the solution is found out or a terminating condition is reached. Thus the search gradually builds up and prunes the entire tree, made up of tasks and subtasks, in a recursive manner. In most of the problems the search grows quite rapidly and usually exponentially. Unless the tree is not quite deep and the branching ratio to subtrees at each level is not large, it would be necessary to cut down the search by adopting some Heuristic guideline.

The set of problems for which a heuristic search based on backtracking Algorithm may be effectively used are defined as follows (Ref. 3) :

Given	: * An initial state * A set of 'Operators' which allow new states to be generated from a given state * A definition of the desired goal state
To Find	: * The sequence(s) of successive 'moves' (forming the solution path) which would lead to the goal state from the given initial state

It may be noted that the above strategy defines essentially the one-person game type of structure. In a graph theoretical representation we may denote each state by a node and the possible moves from a given node would give rise to various branches forming, in a recursive way, into a multibranch tree structure. The strategy mentioned above may equivalenty be posed in the following way with reference to the tree structure :

Given	: * The root node of a connected tree (representing the initial state) * A set of 'Operators' which generate all possible branches from a given node * The description of node(s) which satisfy a predicate (representing the goal state(s))
To Find	: * The sequence(s) of operators which describe a path starting from the root node to the goal node(s)

This MAPPING of one-person game type of problems to a search process in a tree of nodes from the root node to the goal node has been established long back. The mapping has been used extensively in the research studies towards developing algorithms for an intelligent game play on Computers. Some of the refrrences are : Von Neumann and Morgenstern (Ref. 4), Shannon (Ref. 5), Strachey (Ref 6), Samuel (Ref 7,8), Newell, Shaw and Simon (Ref 9), Newell and Simon (Ref 10), Doran and Michie (Ref 11), Hart,Nilsson and Raphael (Ref 12), Moore (Ref 13), Dantzig (Ref 14), Dijkstra (Ref 15), Pohl (Ref 16,17), Quinlan (Ref 18), Michie and Ross (Ref 19) and Michie (Ref 2). Subsequently this area of study has been known as GPS : General Problem Solving.

An interesting outcome of these studies is that given a GPS problem one can clearly establish a distinct separation between two essentially different parts of the program : (1) The general tree search algorithmic part and (2) The task orinted part which is typical of a given problem and specifies the initial and goal states and a repertoir of operators which transforms from one state to another.

Figure 1 shows a typical tree structure which arises in the solution of a GPS problem. The root of the tree represents the initial node. The problem is to find out the path(s) that lead from the initial node to the goal node. The desired solution path is shown in the figure by a dashed line. The tree shown in the figure is a Binary tree where for each node there are at the most 2 branches emerging out.

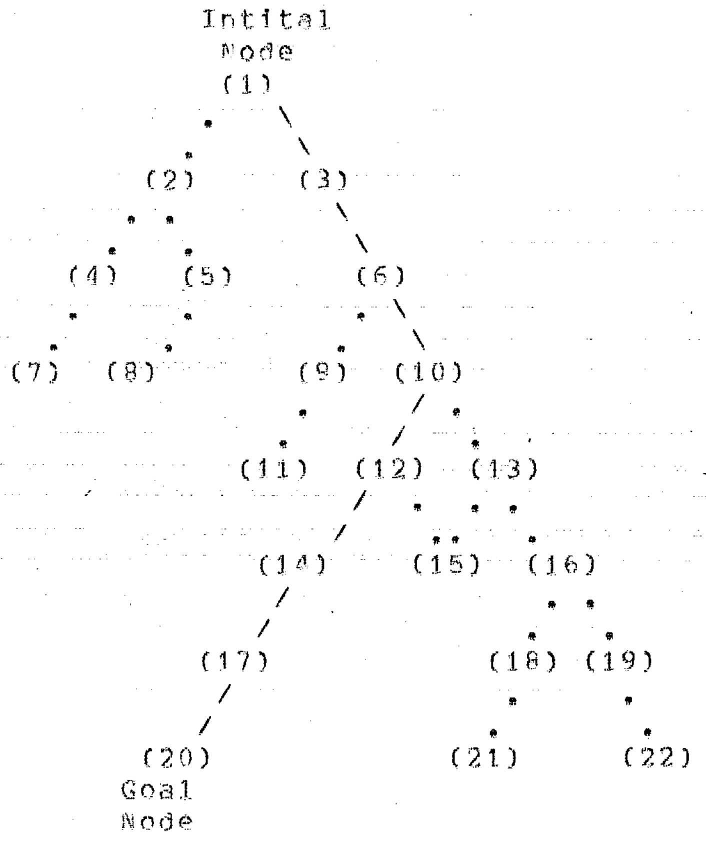
Since in general the size of the tree to be searched may be quite large, it would be an unnecessary overhead to store the entire tree and then make a tree traversal to look for the solution path. The backtracking algorithm avoids such an overhead by generating and storing only the portions of the tree for which search continues and deleting those portions for which the search fails (i.e. typically when a terminal node is reached for which all the neighbouring nodes have been visited before)

For some of the GPS problems instead of the typical tree structure shown in figure 1, a Graph is formed. The essential difference then is that there may exist more than one path connecting the initial node to the goal node. In such cases it would be interesting to look for ALL possible solution paths. For problems related to operation research applications one would also be interested in finding out the OPTIMAL solution path for which the path length is minimum. (An example here is to look for the shortest route between two given nodes in a network). It is interesting to note that with just a minor modification in the backtracking algorithm one would be able to get selectively either all the solution paths or the optimal solution path.

In the present note we outline a number of interesting GPS problems which are solved by developing a suite of programs in Pascal language. (The Pascal language has been used because of its rich data structuring facility). It is interesting to note that although the problems are of diverse nature, the same general scheme of the backtracking algorithm has been used for carrying out a systematic search process in the look out for the solution paths. Each program is made up of about 100 to 200 Pascal statements and requires about 2 to 15K words of core memory at the compile time on DEC-1091 Computer System.

Section 2 describes 3 versions of the algorithm which respectively find ONE solution, ALL the solutions and the OPTIMAL solution. Section 3 discusses 9 different GPS problems and their solutions obtained by the Pascal programs. A brief discussion follwos through in section 4.

Figure 1



A typical BINARY TREE STRUCTURE showing the initial node (node number (1)) which forms the ROOT node of the tree and the GOAL node (node number (20)). The solution path which connects the Initial node to the Goal nide is as follows :

(1) -> (3) -> (6) -> (10) -> (12) -> (14) ->
(17) -> (20)

2.0 THE BACKTRACKING ALGORITHM

2.1 Finding ONE Solution Path

The algorithm which searches for ONE possible solution path in a graph of nodes is shown on the next page as the procedure SEARCH1. The procedure has 3 parameters. The first 2 parameters are value parameters and represent the move number along the path (i.e. the level of depth in a directed graph) and the discription of the state for the present node. 'successful' is a variable parameter and would get the boolean value TRUE or FALSE depending on whether the search is successful to locate a solution path or not. It may be noted that SEARCH1 would be called once in the main body of the program. Then onwards it would call itself recursively through the statement SEARCHE1(I+1, ...). The recursive calls are guided through the boolean variable 'found'. At each stage during the recursion the next instantiation of the procedure SEARCH1 is made only if the variable 'found' has the current value as FALSE. The final value of the procedure parameter i (i.e. when the solution is found), would correspond to the length of the solution path.

The REPEAT .. UNTIL loop is executed kmax times corresponding to the kmax number of branches emerging from the current (i)th node. The array VISITED is ingeneral a multdimensional array of type boolean. It keeps track of the states which have been visited so far at each stage during the search process. A new state (out of the kmax number of states) is entertained only if it is not visited before. This is an important feature which avoids unnecessary cycles within the solution path, thus giving only the basic solution path for which no state is repeated again. As soon as a solution is found out the variable parameter successful gets the value TRUE and correspondingly the recorded solution may be displayed in the main program. If no solution exists then the variable successful holds the value FALSE.

Algorithm 1

The search algorithm to find out ONE possible solution.

```
procedure SEARCH1(i:integer ; state:state_type ;
                  var successful:boolean) ;
  var k,kmax:integer ; found:boolean ;
begin
  (* let MAX be the number of all possible new_states which
     can be generated from the given state *)
  k:=0 ;
  repeat
    k:=k+1 ;
    (* generate the (k)th new_state *)
    if (not visited[new_state]) then
    begin
      (* Record the (i)th move *)
      visited[new_state]:=true ;
      if (* Reached at the Solution *) then found:=true else
      begin
           SEARCH1(i+i,new_state,found) ;
           if (not found) then
           begin
             (* Erase the (i)th move *)
             visited[new_state]:=false ;
           end
      end
    end
  until (found or (k=kmax)) ;
end ;
```

2.2 Finding ALL the Possible Solution Paths.

The search algorithm which looks for all the possible solution paths is shown as the procedure SEARCH2 below. The basic search procedure is the same as in the algorithm SEARCH1. However the modifications which allow for the generation all the paths are as follows. Firstly it may be noted that the variable parameter successful which was used in SEARCH1 is not necessary because as soon as a solution is found out it is displayed. The main difference is that the REPEAT .. UNTIL loop is replaced by a FOR .. DO loop. This allows for scanning all the branches of the graph from a given node even if a solution has been found. (Recall that algorithm SEARCH1 would stop as soon as a solution is obtained because of the boolean condition of the REPEAT .. UNTIL loop, namely, (found or (k=kmax)). Thus if there are multiple paths from the starting state to the goal state then each basic solution path would be found out by the algorithm SEARCH2.

Algorithm 2

The search algorithm to find out ALL possible solutions.

```
procedure SEARCH2(i:integer ; state:state_type) ;
  var k,kmax:integer ;
begin
  (* let max be the number of all possible new states
     which can be generated from the given state *)
  for k:=1 to max do
  begin
    (* record the (k)th new_state *)
    if (not visited[new_state]) then
    begin
      (* record the (i)th move *)
      visited[new_state]:=true ;
      if (* reached at the solution *)
      then (* display the solution *)
      else SEARCH2(i+1,NEW_STATE) ;
      (* erase the (i)th move *)
      visited[new_state]:=false ;
    end ;
  end ;
end ;
```

2.3 Finding out the OPTIMAL Path

Suppose that for a graph corresponding to a given problem there are multiple paths having the path lengths l1,l2,l3 ... ln, where n is the number of different solution paths. It would then be of interest to locate that path for which the path length is optimal. Such a path would be called an optimal path. The algorithm which looks for the optimal path is shown below as the procedure SEARCH3.

The modification to the procedure SEARCH2 which allows to look for the optimal path is as follws. As soon as a new solution path is encountered, its path length is compared with that of the current optimal path. If the length of the former is less than that of the latter then the optimal path is redefined as the presently found path; otherwise the current optimal path is left unchanged. Thus at the end of searching all the paths the path stored as optimal path would contain that path whose path length is optimal.

Algorithm 3

The algorithm to find out the OPTIMAL solution

```
procedure SEARCH3(i:integer ; state:state_type) ;
  var k,kmax:integer ;
begin
  (* let MAX be the number of all possible new states which
     can be generated from the given state *)
  for k:=1 to max do
  begin
    (* Generate the (k)th new_state *)
    if (not visited[new_state]) then
    begin
      (* Record the (i)th move *)
      visited[new_state]:=true ;
      if (* Reached at the solution *) then
      begin
       if (* path length of the current solution is
       the path length of the last solution *) then
       optimal_solution:=current_solution
      end else SEARCH3(i+1,new_state) ;
      (* Erase the (i)th move *)
      visited[new_state]:=false ;
    end
  end
end ;
```

3.0 Applications to some GPS Problems

In this section we consider various well-known problems that belong to the GPS category and solve them using the backtracking Algorithm. For each application the problem is desribed by specifying the following :

1. The INITIAL state

2. The set of operators which transform a given state into another state

3. The GOAL state(s) defined through a given predicate

The problems include applications to integer programming, an OR (Operation Research) application and several popular puzzles belonging to the one-person game type of structure.

3.1 Permutation of n Integers

The problem of finding out all n! distinct permutations of n integers 1,2,3, ... ,n occurs quite frequently in many diverse applications. Two permutations of n items are said to be distinct if they differ in the order of the n items. We now discuss the program which uses the backtracking algorithm and solves this problem in a general way.

1. Problem Definition.

   1. Initial State : Initial sequence of n natural integers

   2. Operator : Change the given sequence of n integers by swapping any 2 integers

   3. Goal state : All n! sequences of the n integers which are distinct to one another according to the order of the integers within sequence

2. Pascal Program

```
Size        :  100 Pascal lines
Core        :  2K words
```

The Pascal program which generates the n! permutations has the following input/output :

   1. Input : The value of the integer n (The initial sequence would be :1,2,3, ..,n)

   2. Output : List of all the n! distinct permutations of the given initial sequence

It may be noted that without using the recursive search construct in the program it would be quite cumboresome to develop an iterative version of the program which would solve the general problem of generating all n! distinct permutations of the given initial sequence.

The recursive method used in the program is as follows. The basic operation to change the sequence of a given permutation is the swapping of two integers. We also develop a procedure PERM3 which generates all 6 permutations of first 3 integers of a given sequence. Consider now the permutation of 4 integers. The 4th integer is initially kept as 4. The above procedure PERM3 is then called which generates 6 permutations having the 4th integer as 4. The 4th integer is then swapped with the 3rd integer of the ORIGINAL sequence. The procedure PERM3 is then called to generate 6 new permutations. This method of calling PERM3 is repeated till the 4th element is swapped with the 1st element.

For given number of elements n (4) the above process looks for all the permutations in which the last element is repeatedly swapped with all the rest elements. This method is elegant in the sense that it always generates only the DISTINCT permutations, thus avoiding the overhead on our part to temporarily store the list of permutations generated at a given stage follwoed by the need to check whether the newly generated permutation is distinct or not.

3. Test Runs

Table 1 gives the average CPU time taken by the present program in generating all the distinct permutations of n elements for several different values of n.

It may be noted that the algorithm is quite efficient as far as the CPU time taken to generate the permutations is considered.

However as the number of integers to be permuted increases rapidly : with each increase of an integer the CPU time is increased by a factor of 10.

Table 1
CPU time taken for generating all the n! distinct permutations of n elements.

n	Number of permutations	CPU time (milli sec)
3	6	< 1
4	24	< 1
5	120	2
6	720	15
7	5040	115
8	40320	970
9	362880	8790
10	3628800	87930

3.2 The 'M-Partition' and Knapsack Problem

These two problems are closely related to each other and are considered as classical problems in integer programming having a wide range of applications. Having given a set of n integers, both the problems look for all the subsets of integers which follow certain constraints. We now consider the two problems one by one.

3.2.1 The M-Partition Problem

1. Problem Definition

Given a set S of n integers {s(1), s(2), ... ,s(n)} it is required to list out all distinct subsets of S whose elements add up to a given value M.

1. Initial state : Null set {} having no elements
2. Operator : Adds up an element from S to the present set.
3. Goal state : All the subsets whose members add up to a given value M.

2. The Pascal program

Size	:	150 Pascal lines
Core	:	2K words

The recursive Pascal program has been developed which makes an exhaustive search and lists out ALL the desired subsets of integers whose members add up to M. Having given the set S of n integers and the sum value M, the program first locates two optimal numbers, namely, the minimum and the maximum possible sum values. The minimum value is essentially the smallest number of the given integer whereas the maximum number is the sum of all the integers. For the given n integers the search process would then be triggered only if the value of M lies between the above two bounds.

The input/output for the program are as follows :

```
Input   : * n = number of members of set S
          * The n members of set S
          * M = the desired sum value

Output  : * List of ALL subsets of S for which their
            members add up to sum value M.
```

3. Tets Runs.

Example 1

```
Input    : n=5
           S={8,5,3,6,1}
           M=9

Output   :
           no     Subset              Nodes searched
           ==     ======              ==============
           1      {8,1}                     16
           2      {5,3,1}                    5
           3      {3,6}                      5
           =========================================

           Total nodes searched                    = 31
           Average nodes searched for 1 soln.      = 10
           Rune (milli sec)                        = 1
```

Example 2

```
Input    :  n=7
           S={7,6,5,4,3,2,1}
           M=12

Output   :
           no     Subset              Nodes searched
           ==     ======              ==============
           1      {7,5}                     34
           2      {7,4,1}                    8
           3      {7,3,2}                    2
           4      {6,5,1}                   21
           5      {6,4,2}                    6
           6      {6,3,2,1}                  4
           7      {5,4,3}                    7
           8      {5,4,2,1}                  2
           =========================================

           Total nodes searched                    = 76
           Average nodes searched for 1 soln.      = 9
           Rune (milli sec)                        = 1
```

Example 3

```
Input    : n=5
           S={2,1,0,-1,-2}
           M=0

Output   :
           no     Subset              Nodes searched
           ==     ======              ==============
           1      {2,1,0,-1,-2}              5
           2      {2,1,-1,-2}                3
```

```
3       {2,0,2}                         5
4       {2,-2}                          3
5       {1,0,-1}                        3
6       {1,-1}                          2
7       {0}                             2
----------------------------------------------

Total nodes searched                   = 26
Average nodes searched for 1 soln.     = 4
Rune (milli sec)                       = 1
```

3.2.2 The Knapsack Problem

1. Problem Definition
Consider a set S of n integers, namely, {s(1), s(2), ... s(n)}. Associated with this set is another set P of n integers, namely, {p(1), p(2), ... p(n)}. The p's are known as profit values. The Knapsack problem lies in finding out that subset of P which meets with the following two constraints :

1. Its members add up to a maximum possible profit value

2. The corresponding subset of S must be such that its members add up to less than or equal to a given sum value M. (For a subset of P ,namely, {p(1),p(2),p(5)}, the corresponding subset of S would be {s(1),s(2),s(5)})

The problem is defined by the initial state, the goal state and the operator as defined below :

1. INITIAL STATE : The null subset {} of P

2. OPERATOR : Adds up a member to the present subset from the given set P.

3. GOAL STATE : The subset of P whose members add up to a maximum possible value with the constraint that the members of the corresponding associated subset of S must add up to less than or equal to the given sum value M.

2. The Pascal Program
   Size : 180 Pascal lines
   Core : 3K words

The recursive algorithm in the present program finds out a valid subset of set P (that satisfies 2nd constraint above) and stores it in a buffer. Whenever a new valid subset is found, it is compared for optimality with the subset in the buffer. If the newly found subset is better than that in the buffer then the content of the buffer is replaced by the newly found subset. At the end of the recursive search the buffer would contain the desired optimal subset of P.

The input/output for the program are as follows :

```
        Input     :  * set S
                  :  * set P
                  :  * sum value M

        Output    :  The optimal subset of P satisfying
                   the constraints mentioned above
```

Example 1

```
         Input    :  S = {3,1,9}
                   P = {20,10,22}
                   M = 10

         Output   :  Desired subset of P = {10,22}
                   Number of nodes searched = 7
                   Run time (milli sec)     = 1
```

Example 2

```
         Input    :  S = {1,2,3,4,5,6,7}
                   P = {12,10,20,100,80,90,50}
                   M = 7

         Output   :  Desired subset of P = {12,10,100}
                   Number of nodes searched = 127
                   Run time (milli sec)     = 71
```

3.3 The Shortest Path in a Network

This is a classical problem in the operation research area and occurs quite frequently in many practical situations. Consider a network having certain number of nodes which are linked to one another. associated with each linkage between two nodes i and j there is a number d(i,j) representing the 'distance' (in some units) between the two nodes. Given two nodes, nemely, the STARTING node and the GOAL node, the problem is to find out the optimal path having the shortest distance enroute from the initial node to the goal node.

1. Problem Definition.

   1. Initial State : The starting node

   2. Operator : Allows to go to one of the nodes which are linked to the present node

   3. Goal state : The goal node

2. Pascal Program
   Size : 200 Pascal lines
   Core : 4K words

In order to look for the desired optimal path the program prunes through the network. The recursive search process finds out all possible paths that connect the starting node to the goal node. At the same time whenever a new path is obtained, the program keeps track of the shortest path out of all the paths generated so far. The search becomes quite involved with the increase in the total number of nodes within the network.

Input : * n = The total number of nodes in the network
* The distance matrix d(i,j) for all the pairs of nodes i and j which are linked together
* The starting and the goal node

Output : * Number of paths found
* The optimal path having the minimum distance between initial and goal node

3. Test Runs :

Example 1

Input : * n=6
* The d(i,j) matrix :

```
                      node j
                  1   2   3   4   5   6
                 ------------------------
             1|   -   4   -  10   -   -|
             2|   4   -   8   -  11   -|
      node   3|   -   8   -   7   -   4|
       i     4|  10   -   7   -   3   -|
             5|   -  11   -   3   -   2|
             6|   -   -   4   -   2   -|

         * Starting node     : 1
           Goal node         : 6

 Output  : * Number of paths found : 8
           * The shortest Path        : 1 -> 4 -> 5 -> 6
           * Optimal Distance         : 15
           * Nodes searched           : 22
           * Run time (milli sec)    : 84
```

Example 2

```
 Input   : * n=8
           * The d(i,j) matrix :

                        node j
                 1   2   3   4   5   6   7   8
                ---------------------------------
            1|   -  10  12   -   -   -   -   -|
            2|  10   -   -   9   -   -   -   -|
     node   3|  12   -   -   8   -   -   -   -|
      i     4|   -   9   8   -   -  18  20   -|
            5|   -   -   -   -   -   -   -   8|
            6|   -   -   -  18   -   -   -   9|
            7|   -   -   -  20   -   -   -   5|
            8|   -   -   -   -   8   9   5   -|

         * Starting node : 1
           Goal node     : 5

 Output  : * Number of paths found : 4
           * The shortest Path        : 1 -> 2 -> 4 -> 7 ->
                                        8 -> 5
           * Optimal Distance       : 52
           * Nodes searched         : 22
           * Run time (milli sec)   : 46
```

3.4 The 8-Queens Problem

Consider a CHESS board having 8*8 cell structure. The problem essentially is to arrange 8 queens on the board such that no two queens can capture each other. It is clear that for the desired arrangement of queens, we must place only 1 queen on each row, each column and each diagonal. Thus during the search process we may restrict the (i)th queen's movement to the (i)th row only. Hence no two queens would be on the same row and thus we would only need to verify that no two queens are on the same column or the same diagonal. This would reduce the search process considerably.

Let us now consider a more general problem of placing n queens (instead of 8 queens) on an n*n board. The above comment of placing (i)th queen on the (i)th row is still valid.

1. Problem Definition.

   1. Initial State : The n queens are initially placed on the 1st column of successive n rows

   2. Operator : One of the queen may be shifted to one square backwards or forwards on its row

   3. Goal state : The list of the arrangement of the n queens such that no two queens can capture each other

2. Pascal Program

Size	:	170 Pascal lines
Core	:	3K words

The successive queens are placed on the board with the constraint that the queen (under consideration for being placed at an (i,j) cell) must not be in a position to capture any of the previously placed queen. We keep track of the squares which can not be occupied by another queen by forming a two dimensional boolean array named DO_NOT_PLACE. Initially all the elements of the array are assigned to value FALSE. Once a queen is placed on a square, the array elements corresponding to the squares falling on its row, column and the 2 diagonals are assigned to the value TRUE. Thus while recursively placing the queens on the board, it can be checked whether a given square is valid for placing a queen or not. The recursive process generates all the possible solutions for a given number of n queens.

It may be pointed that the book keeping aspect is a subtle one in that it requires an additional overhead. Note that the array DO_NOT_PLACE[I,J] is passed as a parameter to the recursive search procedure. Suppose now that the (i)th queen is successfully placed on the (j)th column. Once all the search for the rest of the queens is carried out, it would be necessary at some stage to backtrack and place the (i)th queen on (j+1)th column in order to again carry out an exhaustive search for the rest of queens. It would then be necessary to update the array DO_NOT_PLACE[I,J] array. The updation process involves 2 steps :

1. Make those elements of DO_NOT_PLACE array to FALSE which were made to TRUE because of the placement of (i)th queen on the (j)th column.

2. Make those elements of DO_NOT_PLACE array to TRUE which are affected by placing (i)th queen on (j+1)th column.

The stpe no (2) is a straightforward one but the step (1) is a subtle one. It also requires the information of the list of all squares which were made to TRUE while placing the (i)th queen on (j)th column. Basically this additional overhead comes about because it is quite possible that a given square is attacked by MORE THAN ONE queen placed so far.

```
Input    : * n = Rank of the board
Output   : * List of all solutions. Each solution being
             represented by n integers corresponding to
             the column number for the queen placed on
             successive rows.
```

3. Test Runs :

Example 1

```
Input    : * n=3

Output   : * No solution exists
           * Nodes searched              : 5
           * Run time (milli sec)        : 1
```

Example 2

```
Input    : * n=5

Output   : * Number of solutions obtained = 10
             (Some solutions are shown below)

             No    Arrangement                  Nodes searched
             --    -----------                  --------------
              1    1 3 5 2 4                          5
              6    3 5 2 4 1                          4
             10    5 3 1 4 2                          4
             ---------------------------------------------------
```

```
* Nodes searched          : 53
* Average nodes searched  :  5
* Run time (milli sec)    : 33
```

Example 3

```
Input   : * n=8

Output  : * Number of solutions obtained = 92
            (Some solutions are shown below)

            No    Arrangement                Nodes searched
            --    ---------------            --------------
             1    1 5 8 6 3 7 2 4                 113
             2    1 6 8 3 7 4 2 5                  34
            20    3 6 2 7 5 1 8 4                   4
            81    7 1 3 8 6 4 2 5                  60
            ----------------------------------------------
          * Nodes searched          : 2056
          * Average nodes searched  :   22
          * Run time (milli sec)    : 3591
```

Example 4

```
Input   : * n=10

Output  : * Number of solutions obtained = 724
            (Some solutions are shown below)

            No    Arrangement                   Nodes searched
            --    ------------------            --------------
              1    1  3  6  8 10  5  9  2  4  7     102
             20    1  5  8 10  7  4  2  9  6  3      26
             65    2  4  6  8 10  1  3  5  7  9     201
            111    2 10  7  4  1  3  9  6  8  5     205
            250    4  8  5  9  1 10  7  3  6  2       6
            540    7  9  4  6  1 10  2  5  3  8      46
            ----------------------------------------------
          * Nodes searched          : 35538
          * Average nodes searched  :   724
          * Run time (milli sec)    : 89802
```

3.5 The "Knight's Tour" Problem

In this interesting problem a knight is to moved on a CHESS board from a given initial square such that it visits each and every square of the board without repeating its visit to any square. It is clear that the solution path has a length which is same for all possible different solutions and that it is equal to the total number of squares on the board. It may be noted that we do not give any constraint on the location of the last square visited. (In other words it is not necessary that the knight should visit all squares once and finally come back to the starting square).

For the sake of generality we consider an n*n board rather than the usual 8*8 Chess board.

1. Problem Definition.

Initial State	:	A knight placed on a given cell. The rest of the squares are unvisited so far
Operator	:	The knight jumps to one of the square (out of the maximum 8 possible ones) which has not been visited as yet
Goal state	:	All the squares of the board are visited by the knight (only once)

2. Pascal Program

Size	:	150 Pascal lines
Core	:	2K words

The information regarding whether a square has been visited as yet or not is stored in a boolean array VISITED[I,J]. The backtracking procedure employed in the program continues further till one of the following two conditions is met :

(1) All squares are visited
(2) The knight can not make a legal move

If condition (1) is met then the problem is solved and the solution is reported. If condition (2) is met then the recursive procedure backtracks in order to look for another solution.

It is obvious that the backtracking algorithm used by the program is very crucial and that a repeatitive procedure would not help at all.

The program does not adopt any Heuristic approach. Thus the search is an exhaustive one.

Input : * n = Rank of the board

```
             * (x,y) = The coordinates of the Starting square

   Output  : * All the successful paths of the knight's tour
               on the board.
```

3. Test Runs :

Example 1

```
   Input   : * n=5
             * (x,y) = (1,1)

   Output  : * Number of solutions found :      304
             * Number of nodes searched  : 1735078
             * Run time (milli sec)      :  316484
               (Some of the solutions are shown below)


           No        Nodes searched          Solution
           --        --------------          --------------
           1              8839                1  6 15 10 21
                                             14  9 20  5 16
                                             19  2  7 22 11
                                              8 13 24 17  4
                                             25 18  3 12 23

           5            156223                1 12 17  6 21
                                             18  5 20 11 16
                                             13  2  9 22  7
                                              4 19 24 15 10
                                             25 14  3  8 23

         120               440                1 10 21 16 25
                                             20 15 24  9 22
                                             11  2 13  6 17
                                             14 19  4 23  8
                                              3 12  7 18  5

         300              5618                1 16 11  6 25
                                             10  7  2 17 12
                                             15 20  9 24  5
                                              8  3 22 13 18
                                             21 14 19  4 23
               ----------------------------------------------------
```

Example 2

```
   Input   : * n=5
             * (x,y) = (3,3)

   Output  : * Number of solutions found :       64
```

```
* Number of nodes searched  : 641576
* Run time (milli sec)      : 116506


   (Some of the solutions are shown below)
No        Nodes searched             Solution
==        ==============             ===============
1            9908                    23 10 15  4 25
                                     16  5 24  9 14
                                     11 22  1 18  3
                                      6 17 20 13  8
                                     21 12  7  2 19

21          62667                    23  6 17 12 25
                                     16 11 24  7  2
                                      5 22  1 18 13
                                     10 15 20  3  8
                                     21  4  9 14 19

50              7                    21  6 11 16 23
                                     12 17 22  5 10
                                      7 20  1 24 15
                                      2 13 18  9  4
                                     19  8  3 14 25

61          65072                    25  4 15 10 19
                                     14  9 18  5 16
                                      3 24  1 20 11
                                      8 13 22 17  6
                                     23  2  7 12 21
     ================================================
```

3.6 The problem of 3 Water Jugs

This popular puzzle provides a good illustration of solving one-person game type of problems in the GPS approach.

Three jugs are provided with the capacities of 8,5 and 3 litres each. Initially the 8 litre jug is filled with water upto its brim. The task of the problem is to divide the water equally into 2 parts, i.e. having 4 litres each. in order to do so it allowed to transfer water from one jug to the other. However no measuring device is provided for finding the ammount of water contained in a given jug. The task is thus to find out that path of moves (each move ammounts to pouring water from jug i to jug j) which divides the 8 litre water into 2 equal parts.

In this problem a state would be defined by 3 integers, namely, n1,n2 and n3, indicating the ammount of water currently contained in the 3 jugs having capacities of 8,5 and 3 litres respectively. Thus for example the state designated by (3,3,2) denotes that currently the ammount of water in the jugs having capacities of 8,5 and 3 litres are 3,3 and 2 litres respectively.

It should also be noted that a partial transfer of water from one jug to another would lead to an unknown state and thus such moves must be avoided.

1. Problem Definition.

   Initial State : (8,0,0)
   Operator : water can be poured from one jug to another
   Goal state : (4,4,0)

2. Pascal Program
   Size : 200 Pascal lines
   Core : 4K words

In the Pascal program the sequence of moves is stored in an array of the following Record structure :

```
   move = record
                    n1,n2,n3:0..8
          end ;
```

It would be necessary to avoid looping of moves in the sense that the same state should not be repeated. This is taken care of by maintaining a boolean array VISITED[N1,N2,N3] which stores the information whether a given state has been visited earlier or not. Thus the program is ensured to LOOK AHEAD on the search process and never get lost in a possible looping. (A possible looping would have been : (8,0,0) -> (3,5,0) -> (8,0,0) -> (3,5,0) -> (8,0,0) .....).

   Input : (No input is required)
   Output : * All the possible sequences of moves which would

lead to the Goal state.

3. Test Runs :

```
Input    : * (No input)

Output   : * Number of solutions found :       16
           * number of nodes searched  :      152
           * Run time (milli sec)      :       79
             (Some of the solutions are shown below)


     Nodes        Path
No  Searched      Length   Solution
--  --------      ------   --------------------------------
 1     10           11     (800) => (350) => (053) =>
                           (503) => (233) => (251) =>
                           (701) => (710) => (413) =>
                           (440)

 5     10           16     (800) => (350) => (323) =>
                           (620) => (602) => (152) =>
                           (143) => (053) => (503) =>
                           (530) => (233) => (251) =>
                           (701) => (710) => (413) =>
                           (440)

 7      1            8     (800) => (350) => (323) =>
                           (620) => (602) => (152) =>
                           (143) => (440)

15      9            9     (800) => (503) => (530) =>
                           (233) => (251) => (701) =>
                           (710) => (413) => (44)
----------------------------------------------------------
```

3.7 Finding out a path in a Maze

Consider a 2-Dimensional Maze having n*n squares. From any given square one can move ingeneral to one of the 4 adjacent squares, namely, lying on left, right, up and down directions relative to the present square. The maze is further initialized by specifying walls between some of the pairs of the adjacent squares. A wall between two adjacent squares would imply that it would not be possible to move from one square to the other. An Initial and a Goal square is also specified. The problem is then to find out all possible paths connecting the initial and the goal squares. It would also be necessary to find out the optimal path having the shortest distance enroute from the initial to the goal square.

1. Problem Definition.

Initial state	: Current position is on the initial square
Operator	: Move to one of the adjacent squares
Goal state	: To reach at the Goal square

2. The Pascal program

Size	: 190 Pascal lines
Core	: 13K words

A recursive Pascal program has been developed which makes an exhaustive search and finds out all possible paths connecting the initial and the goal squares. The program keeps track of the squares which have been visited earlier through a 2-dimensional boolean array VISITED[I,J]. Whenever a square is visited, the corresponding array element of VISITED is switched to TRUE. A visit to a new square would be permitted only if the square has not been visited before. Thus starting from the initial square the search proceeds further till the goal square is reached. All the successful paths would be reported. By keeping track of the shortest path found so far, it would be possible to update the shortest path appropriately whenever a new path is found out.

The input/output for the program are as follows :

Input : * n = Rank of the Maze
* Information about possible existance of a wall between all pairs of adjacent squares.
* The coordinates of initial and goal squares.

Output : * All the successful paths which connect the initial and goal squares. Also the shortest path of all the successful paths.

3. Tets Runs.

Example 1

```
Input    : * n=4
           * The maze :
                          -- -- -- --
                      4 |    i        |
                                --
                      3 |   |  |      |

                      2 |   |   g |   |
                             --
                      1 |             |
                          -- -- -- --
                          1  2  3  4
           * Initial squares : (2,4)
             Goal squares    : (3,2)

Output   : * Number of paths found        : 5
           * Number of nodes searched     : 31
           * Run time (milli sec)         :  8
             (Some of the solutions are shown below)

               Nodes      Path
         No   searched   length   Solution
         --   --------   ------   ---------------------------------
         1       7          8       (2,4) => (3,4) => (4,4) =>
                                    (4,3) => (4,2) => (3,1) =>
                                    (3,2)

         3       3          4       (2,4) => (2,3) => (2,2) =>
                                    (3,2)

         4       7          8       (2,4) => (1,4) => (1,3) =>
                                    (1,2) => (1,1) => (2,1) =>
                                    (3,1) => (3,2)
         ----------------------------------------------------------
                     The shortest path is the path No 3
```

Example 2

```
Input    : * n=6
           * The maze :
                          -- -- -- -- -- --
                      6 |                    |
                          --    --    --
                      5 |                    |
                             --    --    --
                      4 |              g     |
                          --    --    --
                      3 |                    |
                             --    --    --
                      2 |    i               |
                          --    --    --
                      1 |                    |
                          -- -- -- -- -- --
                          1  2  3  4  5  6

           * Initial squares : (2,2)
             Goal squares    : (5,4)

Output   : * Number of paths found         :  220
           * Number of nodes searched      : 2367
           * Run time (Milli sec)          :  464
             (Some of the solutions are shown below)

              Nodes     Path
        No  searched   length  Solution
        --  --------   ------  ---------------------------------------
        1      5         6       (2,2) => (3,2) => (3,3) =>
                                 (4,3) => (4,4) => (5,4)

        10     4         8       (2,2) => (3,2) => (3,3) =>
                                 (4,3) => (5,3) => (6,3) =>
                                 (6,4) => (5,4)

        35     10        20      (2,2) => (3,2) => (4,2) =>
                                 (5,2) => (5,3) => (4,3) =>
                                 (4,4) => (3,4) => (2,4) =>
                                 (1,4) => (1,5) => (2,5) =>
                                 (3,5) => (4,5) => (4,6) =>
                                 (5,6) => (6,6) => (6,5) =>
                                 (5,5) => (5,4)
        ----------------------------------------------------------------
                     The shortest path is the path No 1
```

3.8 The Magic Square

Consider a 2-Dimensional matrix having n*n number of elements. For a given value of n we consider the natural numbers 1,2,3 .. n*n. We then look for the arrangement of these natural numbers in the matrix such that the follwoing 2 conditions are met :

1. Each element gets a unique number

2. The individual rows, columns and the 2 major diagonals add up to the same sum value.

As an example, an instance of a magic square for a 3*3 matrix is as follows :

```
4 9 2

3 5 7

8 1 6
```

The sum value for this magic square is 15.

1. Problem Definition.

```
Initial state : All the matrix elements are unfilled
Operator      : Add a number in one of the unfilled
                  matrix elements
Goal state    : All the rows, columns and the
                  2 major diagonals add up to the same
                  sum value
```

2. The Pascal program

```
Size    : 200 Pascal lines
Core    : 2K words
```

The program looks for all possible magic squares with any sum value. However it can be checked that the sum value for a given n*n matrix is : 3*(1+n*n)/2

The input/output for the program are as follows :

```
Input    : * n = Rank of the Matrix

Output   : * All the possible magic squares
```

3. Tets Runs.

Example 1

```
Input    : * n=2
```

```
 Output   : * No solution exists

Example 2
=========

 Input    : * n=3

 Output   : * Number of solutions found    :    8
            * Number of nodes searched     : 6185
            * Run time (Milli sec)         : 2567
              (Some of the Magic squares found are as follows)

                  Nodes
          No    searched       Magic Square
          ==    ========       ============
           1      2287            4  3  8
                                  9  5  1
                                  2  7  6

           3       960            2  7  6
                                  9  5  1
                                  4  3  8

           6       284            8  3  4
                                  1  5  9
                                  6  7  2

           8       170            6  7  2
                                  1  5  9
                                  8  3  4
          ====================================
```

3.9 Problem of Missionaries and Cannibals

Consider that initially 3 Missionaries and 3 Cannibals and a boat are on one side of the river. Their task is essentially to cross the river using the boat and reach on the other side of the river. Their are however several constraints to their movement as mentioned below :

1. The boat would hold at most 2 passangers at a time
2. The Cannibals are dangerous guys and would eat up the Missionaries provided they outnumber them on any one side of the river

It is obvious that the Cannibals can not outnumber the Missionaries enroute on the boat as the boat can hold only 2 passangers. In all other respect the Cannibals would obidiently follow any instructions that they are assigned to : for example, when asked to travel in the boat in company or without the company of a Missionary etc. The problem is thus to find out the appropriate sequence of travel of one or two passangers at a time in the boat back and forth the river sides such that all the 6 people (and ofcourse the boat !) would reach at the other side of the river without any Missionary being eaten up by the Cannibals. There may be more than one solution paths and our task is to find all of them.
For the sake of generality we would now onwards consider the problem of n Missionaries and n Cannibals to be transported on the other side of the river.

1. Problem Definition.

Initial state	:	n Missionaries, n Cannibals and the boat are on THIS side of the river
Operator	:	At_the most 2 people may be shipped to other side of the river (from the side where the boat presently is)
Goal state	:	n Missionaries, n Cannibals and the boat are on THAT side of the river

2. The Pascal program

Size	:	240 Pascal lines
Core	:	8K words

A state in this problem is defined by specifying the distribution of Missionaries and Cannibals on the two sides of the river and the information as to where the boat presently is (i.e. THIS or THAT side of the river). However it sufficient to designate the state by specifying the number of Missionaries and Cannibals on any one side of the river (say THIS side of the river). We thus form a 3-Dimensional array VISITED[N1,N2,BOATSWHEREABOUT] where variables n1 and n2 are of type integer and stand for the number of Missionaries and Cannibals respectively on THIS side of the river and the variable boatswhereabout is of scalar type having the values (here,there).

At any stage during the search for solution, at the most 5 different moves would be possible corresponding to shipping 2M, 2C, 1M+1C, 1M and 1C people on the boat (where M and C stand for Missionary and Cannibal respectively). The program makes recursive search for all these possibilities (whichever are allowed by the constraints) at each stage of the search process. The boolean array VISITED helps to drive the search process only in the FORWARD direction by allowing to get away with any possible looping within the solution path.

The input/output for the program are as follows :

```
    Input    : * n = No of Missionaries (= No of Cannibals)

    Output   : * The solution path(s) indicating the people
                 travelling in the boat during each shift
                 from one side to the other
```

3. Tets Runs.

Example 1

```
    Input    : * n=2

    Output   : * Number of solutions found :  4
               * Number of nodes searched  : 18
               * Run time (Milli sec)      : <1
                 (Some of the solutions are shown below)

    Solution No          : 1
    Nodes searched       : 8

               This side                 That side
      No      M     C    Boat           M     C    Boat
      --    ---------------------     ------------------
      1       2     2    yes            0     0    no
      2       1     1    no             1     1    yes
      3       2     1    yes            0     1    no
      4       0     1    no             2     1    yes
      5       1     1    yes            1     1    no
      6       0     0    no             2     2    yes
    ------------------------------------------------------


    Solution No          : 2
    Nodes searched       : 2

               This side                 That side
      No      M     C    Boat           M     C    Boat
      --    ---------------------     ------------------
      1       2     2    yes            0     0    no
```

```
   2        2      0      no                 0      2      yes
   3        2      1      yes                0      1      no
   4        0      1      no                 2      1      yes
   5        0      2      yes                2      0      no
   6        0      0      no                 2      2      yes
 ----------------------------------------------------------------
```

Example 2

```
 Input    : * n=3

 Output   : * Number of solutions found :   4
            * Number of nodes searched  : 29
            * Run time (Milli sec)      : 1
              (Some of the solutions are shown below)

 Solution No              : 1
 Nodes searched           : 13

              This side                    That side
   No       M      C     Boat            M      C     Boat
   --      ----------------             ----------------
   1        3      3      yes             0      0      no
   2        2      2      no              1      1      yes
   3        3      2      yes             0      1      no
   4        3      0      no              0      3      yes
   5        3      1      yes             0      2      no
   6        1      1      no              2      2      yes
   7        2      2      yes             1      1      no
   8        0      2      no              3      1      yes
   9        0      3      yes             3      0      no
   10       0      1      no              3      2      yes
   11       1      1      yes             2      2      no
   12       0      0      no              3      3      yes
 ----------------------------------------------------------------


 Solution No              :  2
 Nodes searched           : 12

              This side                    That side
   No       M      C     Boat            M      C     Boat
   --      ----------------             ----------------
   1        3      3      yes             0      0      no
   2        3      1      no              0      2      yes
   3        3      2      yes             0      1      no
   4        3      0      no              0      3      yes
   5        3      1      yes             0      2      no
   6        1      1      no              2      2      yes
   7        2      2      yes             1      1      no
   8        0      2      no              3      1      yes
   9        0      3      yes             3      0      no
   10       0      1      no              3      2      yes
```

```
  11      1      1     yes              2      2     no
  12      0      0     no               3      3     yes
-----------------------------------------------------------------
```

Example 3

```
Input    : * n=4

Output   : * Number of solutions found :  0
           * Number of nodes searched  : 19
           * Run time (Milli sec)      : 1

             No solutions exist.
```

4.0 DISCUSSION

The present note discusses the BACKTRACKING ALGORITHM and illustrates its usage in solving typical "ONE-PERSON GAME" type of problems. The well known mapping of such problems onto search in a graph of node (from the given Initial node to the Final one) is used. The problems solved are essentially the ones belonging to the GPS (General Problem Solving) category. It is emphasized that the algorithm provides a neat and elegant way to solutions of the following kinds :

1. Any ONE solution.

2. ALL the solutions.

3. The OPTIMAL solution.

The real potential of this approach lies in that the SAME general form of the search algorithm may be used to solve a variety of diverse problems.

It may be pointed out that the "Breadth First" search employed in the present algorithm makes an exhaustive search thereby consuming a substantially heavy computation. In particular the search becomes quite involved and may even reach to unmanageable bounds for graphs which are deep and at the same time have large branching ratios. In such cases appropriate Heuristic approach must be used which would guide the search favourably in the more promising zones of the underlying graph thereby reducing the search.

ACKNOWLEDGEMENTS

The author is grateful to Professors A.Sen and K.H.Bhatt for the encouragement in the present work.

He is also grateful to Mr. H.S.Mazumdar for reading the manuscript and making suggestions for its improvement.

The interest and encouragement provided by Professor S.P.Pandya is acknowledged with deep sense of gratitude.

REFERRENCES

1. Wirth N, "Algorithm + Data Structure = Program", Prentice Hall Publication (1976)

2. Michie D., "Machine Intelleigence and Related Topics - An Information Scientist's weekend Book", Gordon and Breach Science Publishers (1982)

3. Ernst G.W. and Newell A., "GPS : A case study in Generality and Problem Solving", Academic Press (1969)

4. Von Neumann J. and Morgenstern)., "Theory of Games and Economic Behaviour", Princeton University Press (1944)

5. Shannon C.E., Programming a Computer for Playing Chess, Philosophical Magazine vol.41 p.256 (1950)

6. Strachey C.E., Logical or Non-Mathematical Programmes, Proc. of ACM meeting p.46, Toronto (1952)

7. Samuel A.L., Some studies in Machine Learning using the game of Checkers -2, IBM J. of Res. and Devl. vol.11, p.601 (1967)

8. Newell A., Shaw J.C. and Simon H.A., Preliminary Description of General Problem Solving Program - 1, CIP working paper no.7, Pittsburg : Carnegie Institute of Technology (1957)

9. Newell A. and Simon H.A., GPS : A Program that Simulates thought, Lenede Aoutomaten, (Ed. Dilling H.) p.109 (1961)

10. Doran J.E. and Michie D., Experiments whith the Graph Traverser Program, Proc. of Royal Soc. (A) vol.294, p.235 (1966)

11. Hart P., Nilsson N. and Raphael B., A formal basis for the Heuristic Determination of Minimum cost Paths, IEEE Trans. on Systems Science and Cybernatics (1980)

12. Moore E., The Shortest Path through a Maze, Proc. of Int. Symp. on the theory of Swithing, part.2, p.285, Cambridge : Harvard University Press (1959)

13. Dontzig G., Linear Programming and Extensions, Princeton University Press (1963)

14. Dijkstra E., A note on two problems in connection with Graphs, Numerische Mathematik vol.1, p.269 (1959)

15. Pohl I., First Result on the effect of error in Heuristic Search, Machine Intelligence vol.5, p.219 (1970)

16. Pohl I., Heuristic Search viwed as Path Finding in a Graph, Artificial Intelligence vol.1, p.193 (1970)

17. Quinlan J.R., A task independent experience gathering scheme for a problem solver, Int. Joint Conf. on Artificial Intelligence (Ed. Walker D.E. and Norton L.M.) p.193 (1969)

18. Michie D. and Ross R., Experiments with the Adaptive Graph Traverser, Machine Intelligence vol.5, p.301 (1970)

19. Slagle James R., "Artificial Intelligence : The Heuristic Programming Approach", McGraw Hill Publication (1971)

****

INDEX