PROBLEM SOLVING STRATEGY

BACKTRACKING

PRESENTED BY
MADHURIMA RAWAT
ROLL NO-300012821042
DATASCIENCE(CSE)

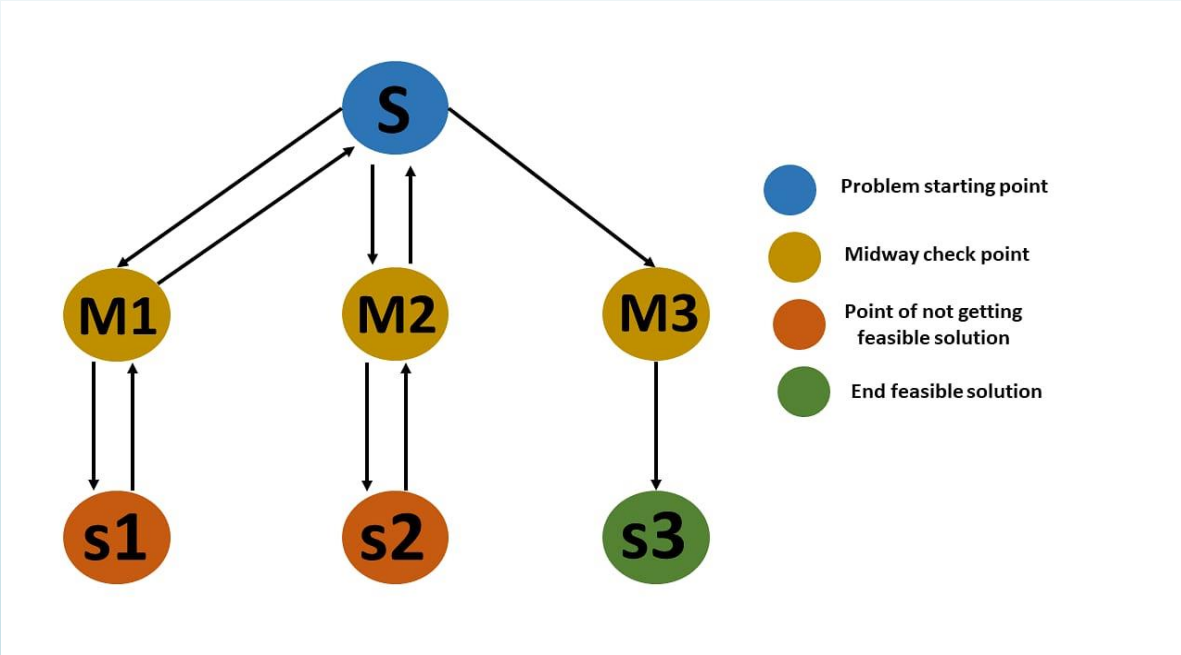
CHHATTISGARH SWAMI VIVEKANANDA TECHNICAL UNIVERSITY

WHAT IS BACKTRACKING?

- **Backtracking** is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.
- Backtracking algorithm is applied to some specific types of problems,
- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

EXAMPLE OF BACKTRACKING

- In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.
- Here, when the algorithm propagates to an end to check if it is a solution or not,
- if it is then returns the solution otherwise
- backtracks to the point one step behind it to find track to the next point to find solution.



ALGORITHM

Step 1 – If current_position is goal, return success

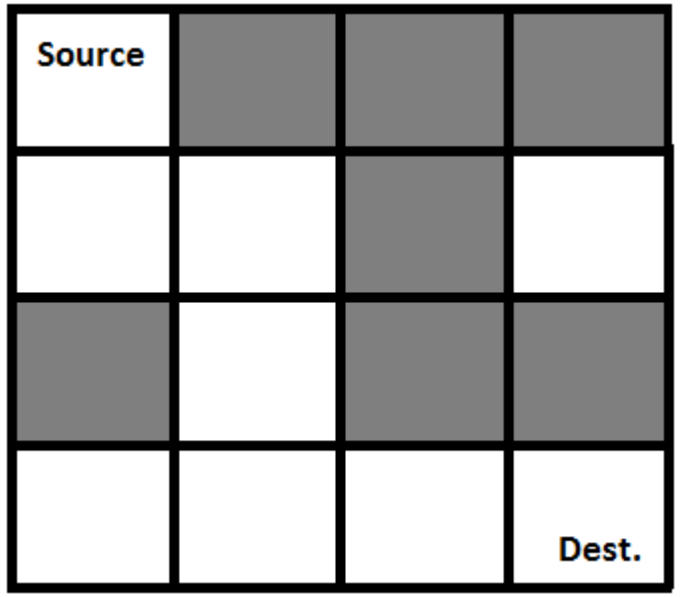
Step 2 – Else

Step 3 – If current_position is an end point, return failed.

Step 4 – Else, if current_position is not end point, explore and repeat above steps.

RAT IN A MAZE BACKTRACKING APPLICATION PROBLEM

- Constraints:
- A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1].



A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.

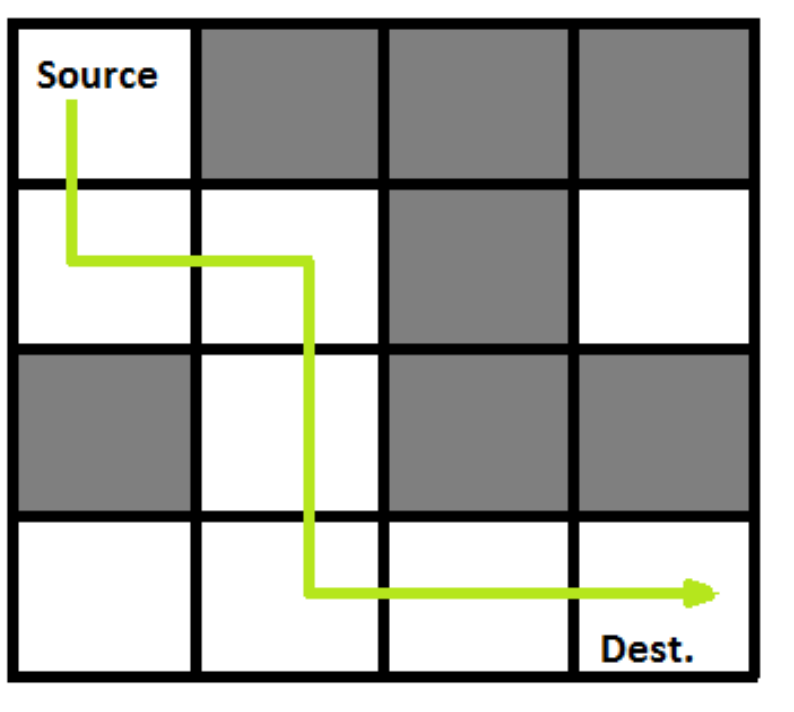
- In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination.

Gray blocks are dead ends (value = 0).

REPRESENTATION IN BINARY FORM

Following is a binary matrix representation of the above maze.

```
{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}
```



Following is the solution matrix (output of program) for the above input matrix.

```
{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}
```

All entries in solution path are marked as 1.

SOLUTION ANALYSIS

- **Approach:** Form a recursive function, which will follow a path and check if the path reaches the destination or not. If the path does not reach the destination then backtrack and try other paths.
- **Algorithm:**

1. Create a solution matrix, initially filled with 0's.
2. Create a recursive function, which takes initial matrix, output matrix and position of rat (i, j).
3. if the position is out of the matrix or the position is not valid then return.
4. Mark the position ou ut[i][j] as 1 and check if the current position is destination or not. If destination is reached print the output matrix and return.
5. Recursively call for position (i+1, j) and (i, j+1).
6. Unmark position (i, j), i.e output[i][j] = 0.

CODE IN C FOR IMPLEMENTATION

```
#include <stdio.h>
#include<stdbool.h>
// Maze size
#define N 4
bool solveMazeUtil(int maze[N][N], int x, int
y,int sol[N][N]);

  void printSolution(int sol[N][N]){
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", sol[i][j]);
        printf("\n"); }}
// A utility function to check if x, y is
valid index for N*N maze
bool isSafe(int maze[N][N], int x, int y){
    // if (x, y outside maze) return false
    if (x >= 0 && x < N && y >= 0 && y < N &&
maze[x][y] == 1)
        return true;
    return false;}
```

```
bool solveMaze(int maze[N][N]){
    int sol[N][N] = { { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 } };
    if (solveMazeUtil(maze, 0, 0, sol) ==
false) {
        printf("Solution doesn't exist");
        return false;}
    printSolution(sol);
    return true;}

 bool solveMazeUtil(int maze[N][N], int x, int y,
 int sol[N][N]){
// if (x, y is goal) return true
if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {
    sol[x][y] = 1;
    return true; }
// Check if maze[x][y] is valid
if (isSafe(maze, x, y) == true) {
    // Check if the current block is already part of
    // solution path.
    if (sol[x][y] == 1)   return false;
```

CONTINUATION

```
 // mark x, y as part of solution
path
        sol[x][y] = 1;
      /* Move forward in x
direction */
        if (solveMazeUtil(maze, x +
1, y, sol) == true)
            return true;
        // If moving in x direction
doesn't give solution
        // then Move down in y
direction
        if (solveMazeUtil(maze, x,
    y + 1, sol) == true)
            return true;
```

```
 // BACKTRACK: unmark x, y as part of solution
path
        sol[x][y] = 0;
        return false;}
    return false;}

  // driver program to test above function
     int main(){
   int maze[N][N] = { { 1, 0, 0, 0 },
                      { 1, 1, 0, 1 },
                      { 0, 1, 0, 0 },
                      { 1, 1, 1, 1 } };
solveMaze(maze);
return 0;}
```

OUTPUT:

The 1 values show the path for rat

```
1 0 0 0
1 1 0 0
1 0 0 0
0 1 1 1
```

Complexity Analysis:

- **Time Complexity:** O(2^(n^2)).
  The recursion can run upper-bound 2^(n^2) times.
- **Space Complexity:** O(n^2).
  Output matrix is required so an extra space of size n*n is needed.

THANK YOU FOR LISTENING TO ME

ANY QUESTIONS