61 Backtracking Algorithms

"What comes out of a man is what makes him 'unclean'."

Have you ever seen poor blind people walking in roads? If they find any obstacles in their way, they would just move backward. Then they will proceed in other direction. How a blind person could move backward when he finds obstacles? Simple answer…by intelligence! Similarly, if a algorithm backtracks with intelligence, wonderful isn't it?

61.1 Recursive Maze Algorithm

Recursive maze algorithm is one of the good example for backtracking algorithms. In fact Recursive maze algorithm is one of the most available solutions for solving maze.

61.2 Maze

Maze is an area surrounded by walls; in between you have a path from starting position to ending position. We have to start from the starting point and travel towards the ending point

61.3 Principle of Maze

As explained above, in maze we have to travel from the starting point to ending point. The problem is to choose the path. If we find any dead-end before ending point, we have to backtrack and change the direction. The direction for traversing be North, East, West and South. We have to continue "move and backtrack" until we reach the ending point.

Example Maze
1 1 1 1 1 1 1
1 0 0 0 1 1 1
1 1 1 0 1 1 1
1 1 1 0 0 0 1
1 1 1 1 1 0 1
1 1 1 1 1 1 1

Assume that we are having a two-dimensional maze `cell[WIDTH][HEIGHT]`. Here `cell[x][y] = 1` denotes wall and `cell[x][y] = 0` denotes free cell in the particular location `x, y` in the maze. The directions we can move in the array are North, East, West and South. The first step is to make the boundary of the two-dimensional array as 1 so that we won't go out of the maze, and always reside inside the maze at any time.

Now start moving from the starting position (since the boundary is filled by 1) and find the next free cell, then move to the next free cell and so on. If we reach a dead-end, we have to backtrack and make the cells in the path as 1(wall). Continue the same process till the ending point is reached.

61.4 Program

The following program finds whether there is a path available between the two positions in maze or not. Here I have used (1, 1) and (8, 10) as the two positions.

```
/*-------------------------------------------------------------------
                                  Maze
                                   by
                             K. Joseph Wesley
                        http://JosephWesley.itgo.com
---*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef int BOOLEAN;

#define WIDTH     (12)
#define HEIGHT    (10)

#define TRUE      (1)
#define FALSE     (0)

int cell[10][12] = {
                  {1,1,1,1,1,1,1,1,1,1,1,1},
                  {1,0,0,0,0,1,1,1,1,1,1,1},
                  {1,1,0,0,0,1,1,1,0,1,1,1},
                  {1,1,0,1,0,0,0,0,0,1,1,1},
                  {1,1,0,1,0,0,0,0,0,1,1,1},
                  {1,1,0,1,0,0,0,0,0,0,1,1},
                  {1,1,0,1,0,0,0,0,0,0,0,1},
                  {1,1,0,0,0,0,0,0,0,0,0,1},
                  {1,1,0,0,0,0,0,0,0,0,0,1},
                  {1,1,1,1,1,1,1,1,1,1,1,1}
              };

void PrintMatrix( void )
{
   int i, j;
   for( i=0;i <HEIGHT; ++i )
     {
      for( j=0; j<WIDTH ; ++j )
           printf( "%2d", cell[i][j] );
           printf( "\n" );
     }
} /*--PrintMatrix( )-----*/
```

```
void Traverse( BOOLEAN *pathavailable, int x1, int y1, int x2, int y2 )
{
    if ( x1 == x2 && y1 == y2 )
      *pathavailable = TRUE;
    if( cell[x1][y1] == 0 )
       {
         cell[x1][y1] = 1;
         Traverse( pathavailable, x1, y1+1, x2, y2 );
         Traverse( pathavailable, x1+1, y1, x2, y2 );
         Traverse( pathavailable, x1, y1-1, x2, y2 );
         Traverse( pathavailable, x1-1, y1, x2, y2 );
       }
} /*--Traverse( )------*/

int main( void )
{
    BOOLEAN pathavailable = FALSE;
    clrscr();
    printf( "Original Maze: \n" );
    PrintMatrix( );
    Traverse( &pathavailable, 1, 1, 8, 10 );
    printf( "Maze after operations: \n" );
    PrintMatrix( );
    printf( "Path is %s available \n", (pathavailable)? "" : "NOT");
    getch( );
    return ( 0 );
} /*--main( )--------*/
```

Exercises

1. Find out other backtracking problems.
2. Solve 8 Queen problem.