CSE246: Algorithms (Section-07)

Summer 2025

Project Report

Maze Path Finding using Backtracking

Submitted by:

Student ID	Student Name	Contribution Percentage
2023-1-50-004	Ekramul hassan ratul	25%
2023-1-50-016	Farabi sarker	25%
2022-3-60-089	Md sayed hossain	25%
2023-1-50-010	Easin Arafat	25%

Part 1 : Problem Definition & Theoretical Background.

1.1 Problem Statement

The Maze Path finding problem deals with navigating through an N x N Matrix where :

1 = open path

0 = wall

The task is to start at the top left cell (0,0) and reach the bottom right cell (N-1, N-1) using valid moves (up, down, left, right). The main challege is finding a feasible path without crossing walls or going outside the maze.

1.2 Input and Output

Input :

- Integer N (maze size)
- 2D matrix maze [N][N] with 0 and 1

Output :

- Boolean results : whether a path exists.
- If yes then, sequence of coordinates from start to end.
- If no then, "No solution exists".

Objectives

- To understand and solve the maze pathfinding problem using algorithms.
- To implement backtracking for efficient path exploration.
- To compare backtracking with brute force and show why brute force is impractical.
- To highlight real world uses of pathfinding.

Real World Applications

Maze solving has direct applications in :

- Robotics and navigation - obstacle avoidance and path planning.
- Computer networks - routing data packets effeiency.
- Game development - AI movement and Puzzle-solving.
- Logistics - route optimization in warehouses and delivery.

Why Backtracking?

Brute fore explores all possible paths, which becomes exponentially slow for larger mazes, backtracking is more practical because it :

- Explores step by step
- Abandons dead ends quickly.

- Finds a valid path efficiently (if one exists).

Theoretical Background

Backtracking is a recursive search strategy:

i) Start at (0, 0)

ii) Move to a valid neighboring cell.

iii) If stuck, backtrack to try another direction.

iv) Repeat until (N-1, N-1) is reached or all options are exhausted.

This makes backtracking well-suited for maze problems where multiple possible paths exists.

Algorithm Design

Problem setup

find ~~i) a~~

i) any valid path (Backtracking) and

ii) the shortest path (BFS) from (0,0) to (n-1, n-1) in an nxn binary maze (1=path, 0= wall).

Assumption

Start / end open: maze [0] [0] == 1,

maze [n-1] [n-1] == 1

moves: 4-neighborhood - Right (0, 1), Down (1,0)

Left (0, -1), up (-1, 0)

output: solution grids (sol bt, sol bfs)

and ondend coordinate paths.

Sate / Pate

maze [n] [n]

visited: visited bt, visited bfs

solutions: sol bt, sol bfs

~~DFS~~ Design choices:

Backtracking (DFS): depth-first, backtracks on dead ends. Simple finds a path not ~~simple~~ guaranteed shortest.

~~com~~ Termination

BT: finite branching + backtracking visited avoids cycles. Success ⇒ constructed path valid.

~~BFS~~ BFS: visited nodes in non-decreasing distance. first goal dequeue ⇒ shortest. parents reconstruct path.

complexity:

BT time exponetial in path length ($\leq n^2$) ; space $O(n^2)$

BFS time $O(n^2)$; space $O(n^2)$.

start goal blocked ⇒ no solution

```
Procedure Solve BT (maze, n):
MAKE sol bt [n] [n] ← 0
MAKE path visited [n] [n] ←0
MAKE path-bt ← empty list
IF maze [0] [0] == 0 OR maze [n-1]
[n-1] == 0:
  Return fail
IF DFS (0,0) == True:
 Return (sol bt, path bt)
 Else:
 Return fail

Procedure in bounds (x, y, n):
 Return (0≤x<n) and (0≤y<n)
Procedure open (x, y):
 Return in bounds (x, y, n) And (maze [x]
 [y] == 1) And (visited [x] [y] == 0)
```

do
Pegudode

Procedure ~~DFS~~ BFS (x, y):

If not open (x, y):

Return false

visited [x] [y] ← 1

sol bt [x] [y] ← 1

Append (x, y) To path bt

IF x == n-1 And y == n-1:

Return True

for each (dx, dy) IN [(0,1), (1,0), (0,-1)
(-1,0)]:

nx ← x + dx

ny ← y + dy

IF ~~DFS~~ BFS (nx, ny) == True:

Return True

Remove Last Element
from path bt
sol bt [x] [y] visited [x] [y] ← 0
Return false

Role Assignment (section 5)

dbt ( ) ⟶ recursive DFS for backtracking solution.

run_bt ( ) → sets up and runs backtracking

run_bfs ( ) ⟶ sets up and runs BFS shortest path

printg ( ) ⟶ prints a 2D grid of the solution path.

count_1 ( ) → counts path cells.

read_row_digits ( ) ⟶ reads input rows (supports 1010 or 1 0 1 0 formats)

Main function → Takes input n and maze grid.

Calls both algorithm.

Prints solution paths and step counts.

Experiment Results :

Backtracking (BT) :

Find any valid path faom $(0, 0) \rightarrow (n-1, n-1)$

prints path matrix, coordinates. and total steps.

If no path → None.

BFS : Finds the shortest path.

prints path matrix, coordinates in order and step count.

If no path → NONE.

Sample Output Behavior

BT

< path matrix >

(0, 0) (0, 1) ... (n-1, n-1) steps

BFS

< shortest path matrix >

(0, 0), (1, 0) ... (n-1, n-1)

steps.

Tools and Technologies Used

Language : C

Liabnanies :

<stdio.h> → input /output

<string.h> → string operations

<ctype.h> → chanacten checking

Algonithm :

- Backtracking (DFS) → recursive path search
- BFS (Queue - based) → shontest path search

Data Structure :

- 2D annay for grid, visited states and paths
- Queues (annay) for BFS
- Annays for storing path coordinates.

References:

- Cormen, Leiserson, Rivest, Stein: Introduction to Algorithoms (CLRS)
- Course lecture notes on recursion and backtracking
- Online tutorials on maze solving in C

APPendix: Final c source code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 50
int n,g[NMAX][NMAX];
int inb(int x,int y){return x>=0&&x<n&&y>=0&&y<n;}
int open(int x,int y){return inb(x,y)&&g[x][y]==1;}
```

```
int sbt[Nmax][Nmax], vbt[Nmax][Nmax],
    pxb[Nmax*Nmax], pyb[Nmax*Nmax], Ib;

int dbt(int x, int y){
 if(!(open(x,y)&& vbt[x][y])) return 0;
 vbt[x][y]=1; sbt[x][y]=1; pxb[Ib]=x;
 pyb[Ib]=y; Ib++;
 if(x==n-1 && y==n-1) return 1;
static int dx[4]={0,1,0,-1}, dy[4]={1,0,-1,0};
for(int k=0; k<4; k++){
    int nx=x+dx[k], ny=y+dy[k];
    If(dbt(nx,ny)) return 1;
}
  vbt[x][y]=0; sbt[x][y]=0; Ib--;
  return 0;
}
   int run_bt(void){
for(int i=0; i<n; i++) for(int j=0; j<n; j++){
 sbt[i][j]=0; vbt[i][j]=0;}
```

```
lb=0;
if(!open(0,0)||!open(n-1,n-1)) return 0;
return dbt(0,0);
}
int sbf[NMax][NMAX], vbf[NMax][NMAX],
  px[NMax][NMax];
int run_bfs(void){
for(int i=0;i<n;i++) for(int j=0;j<n;j++)
  {sbf[i][j]=0; vbf[i][j]=0; px[i][j]=
if(!open(0,0))||!open(n-1,n-1)) return 0;
int qx[NMAX*NMax], qy[NMax*NMAX], f=0,b=0;
vbf[0][0]=1; qx[b]=0; b++
int dx[4]={0,1,0,-1}, dy[4]={1,0,-1,0};
while(f<b){
    int x=qx[f], y=qy[f]; f++;
    if(x==n-1&y==n-1) breake;
    for(int k=0;k<4;k++){
      int nx=x+dx[k], ny=y+dy[k];
```

```
while (!(cx == 0 && cy == 0)){
  bfp [cx] [cy] = 1;
  int tx = px [cx] [cy], ty = py [cx] [cy];
  cx = tx; cy = ty;
}
bfp [0] [0] = 1;
return 1;
}
void printing (int a [Nmax] [Nmax]){
  for (int i=0; i<n; i++){
```

```
if (open (nx, ny) && !vbf [nx] [ny]){
  vbf [nx] [ny] = 1; px [nx] [ny] = x; py [nx] [ny]
    = y;
  q x [b] = nx; q y [b] = ny; b++;
}
}
}
if (!vbf [n-1] [n-1]) return 0;
int cx = n-1, cy = n-1;
```

```
for (int j=0; j<n; j++) printf("%d", a[i][j]);
printf("\n");
}
}
int count1(int a[Nmax][Nmax]){
int c=0; for (int i=0; i<n; i++) for (int j=0; j<n;
j++)
if (a[i][j]) c++; return c;
}
void flush_line(void){
int ch;
while ((ch=getchar())!='\n' && ch!=EOF);
}
int read_row_digits(int row){
char buf[512];
If (!fgets(buf, sizeof(buf), stdin)) return -
int col=0;
for (int i=0; buf[i]; i++){
```

```
f(buf[i]=='\n') break;
if (isspace((unsigned char)buf[i])) continue;
if(buf[i]=='0' || buf[i]=='1'){ if(col>=n)
 return 0;
 }
 }
} return (col==n);
} int main(void){
printf("Maze size n (1...%d): ",NMax);
if (scanf("%d",&n)!=1 || n<=0 || n>NMax{
 printf("Invalid n\n");
 return 0;
} flush_line();
printf("%dx%d maze\n");
for(int i=0;i<n;i++){
while(1){
 printf("Row %d: ",i);
```

```
while (1){
Printf("  wrong input! give input again");
}
}
  int ok1 = run_bt();
  int ok2 = run_bfs();
  if(ok1){
   Printf("BT\n");
   Path_bt();
  for (int i=0;i<lb;i++)
Printf("(%d,%d)%s",pxb[i],pyb[i],(i+1<lb)?
        "  ":"\n");} else{
Printf("BT\nNone\n");
} if (ok2){
  int prs[Nmax*Nmax], pys[Nmax*Nmax],
     l=0, cx=n-1, cy=n-1;
  while (!(cx==0&&cy==0))
 }else {printf("BFS\nNONE\n");
```

Conclusion, Documentation & Appedix

This section provides the final summary and supporting documents. The Maze Path Finding Project demonstrate the power of recursion and backtracking. It shows how complex problems can be solved by systematically trying possibilities and undoing choice. The project has value in learning Algorithom design, handling edge cases, and writing clean, modular code. Future improvments could involve extending the program to allow movement in all four directions. Implementing shortest path Algorithoms like BFS or A*, and visualizing the solution with graphics. this section also includes rebences used durring the preparation of this report.