আন্তর্জাতিক ইসলামী বিশ্ববিদ্যালয় চট্টগ্রাম

الجامعة الإسلامية العالمية شيتاغونغ

International Islamic University Chittagong

## Department Of Computer Science And Engineering

***Project Report***

## Student Information :


**Name: Jannatul Ferdous Tania**

**ID: C241486**

**Semester : 3rd**

**Section : 3CF**


## Project Information:

**Project Name: Maze Generator And Solver**

**Course title: Data structures Lab**

**Couse Title: CSE-2322**

## Teacher Information:


**Asmaul Hosna Sadika**

**Assistant Lecturer,**

**Dept. of CSE, IIUC**

## Abstract

A **maze generator and solver** is a program that creates a path puzzle (maze) and then finds a way to go from the start to the end.
In this project, I built a maze generator and solver using C++. The program can take user input for a custom maze or generate one randomly using DFS. Then, it solves the maze using BFS and shows the shortest path. Users can also insert or remove walls and re-solve the maze to see the changes. This helps understand how graph algorithms work in real time. The program is text-based but interactive. It can be expanded in future with graphics, more smart algorithms (like A*), and game-like features.In this project I works that:

- I created a program to generate mazes automatically using DFS.
- I added a maze solver using BFS to find the shortest path.
- I allowed the user to input custom mazes manually.
- I added options to insert or delete walls in the maze.
- I displayed the solution path and allowed users to see the updated maze after changes.

**Real-life uses:**

- **Robots** finding the shortest path in a building
- **Game AI** solving levels or puzzles
- **Maps/GPS** finding best route
- **Emergency exits** planning in buildings

# Table of contents (with page no)

# Maze Solver and Generator

## Introduction

This project develops a program in C++ to generate and solve mazes. A maze is a puzzle made of paths and walls. The program can either generate a maze randomly or take maze input from the user. Users can add or remove walls and search cell walls. The maze solver uses a Breadth First Search (BFS) to find a path from the start to the end. The program also displays the maze and the solution path visually using ASCII characters.

## Objectives

- To create a maze either by user input or automatic generation.
- To allow modification of maze walls.
- To implement a maze-solving algorithm to find a path.
- To visually display the maze and the solution path.
- To practice C++ programming with recursion, arrays, and queues.

## Motivation

I got the idea for this project from YouTube tutorials and coding websites where maze problems are used to teach algorithms. The idea came from the classic maze puzzles and games. I wanted to understand how mazes can be generated programmatically and how computers can solve puzzles. This project helps learn important programming concepts like recursion and graph traversal. It is also fun to see how algorithms work visually.

## Related Work / Literature Review

1. Maze Generation Using Depth-First Search (DFS)
   Many researchers and developers use the DFS backtracking algorithm for maze generation. This method creates perfect mazes with one unique path between any two points. It is simple and efficient for generating complex mazes.
   *Reference: Eller's Maze Generation Algorithm and DFS - Recursive Backtracking*

2. Maze Solving with Breadth-First Search (BFS)
   BFS is widely used in maze solving due to its ability to find the shortest path in unweighted grids. It systematically explores all possible paths layer by layer until it reaches the goal.
   *Reference: Introduction to Algorithms by Cormen et al.*

3. *A Algorithm for Maze Solving**
   A* is a popular heuristic-based pathfinding algorithm that improves search efficiency by

estimating the cost to the goal, thus finding optimal paths faster than BFS. It is common in game development and robotics.
*Reference: Hart, Nilsson, and Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," 1968*

4. Graphical Maze Games and Interactive Maze Editors
Several projects develop graphical user interfaces for maze generation and solving, allowing users to interactively edit mazes and visualize solutions dynamically, enhancing learning and engagement.
*Reference: Maze game projects on GitHub and open-source platforms*

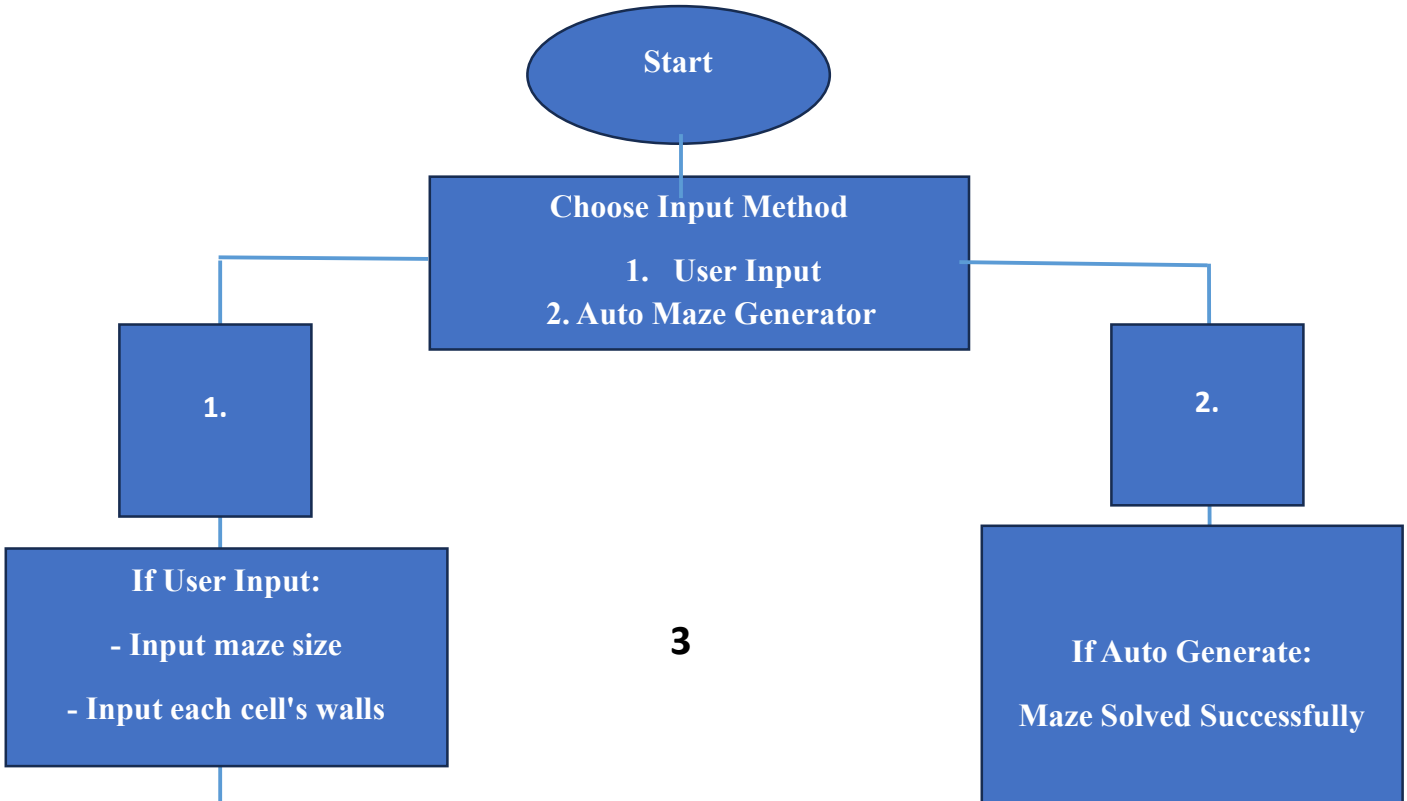
---

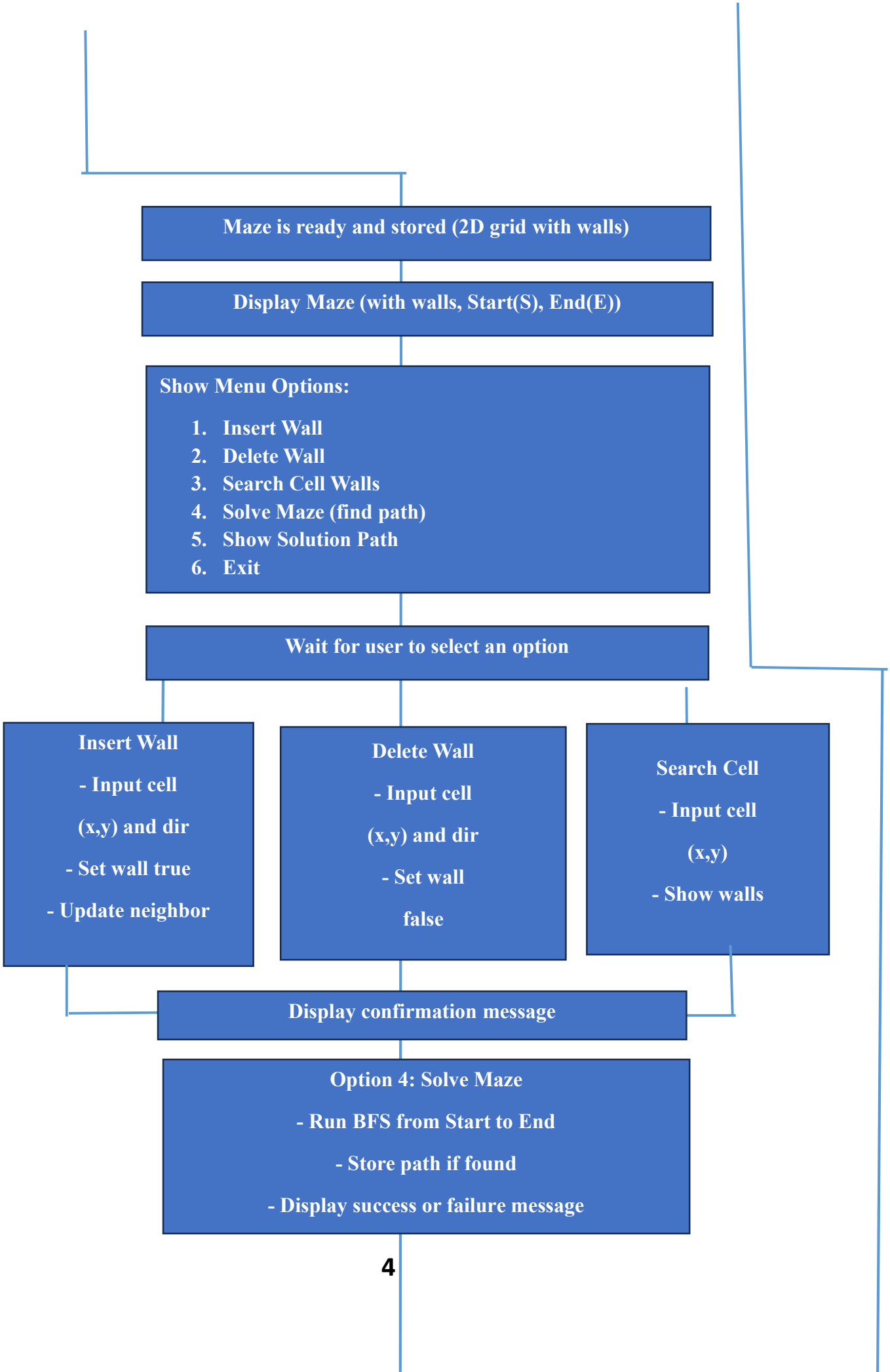
## References


1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
2. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*.
3. Eller, J. (Year unknown). *Maze generation algorithm*. Available at multiple online sources.
4. Various Open-source Maze Projects on GitHub. (n.d.). Retrieved from https://github.com


## Methodology

The project is divided into the following steps:




Maze is ready and stored (2D grid with walls)
Display Maze (with walls, Start(S), End(E))
Show Menu Options:
1. Insert Wall
2. Delete Wall
3. Search Cell Walls
4. Solve Maze (find path)
5. Show Solution Path
6. Exit
Wait for user to select an option
Insert Wall
- Input cell
(x,y) and dir
- Set wall true
- Update neighbor
Delete Wall
- Input cell
(x,y) and dir
- Set wall
false
Search Cell
- Input cell
(x,y)
- Show walls
Display confirmation message
Option 4: Solve Maze
- Run BFS from Start to End
- Store path if found
- Display success or failure message

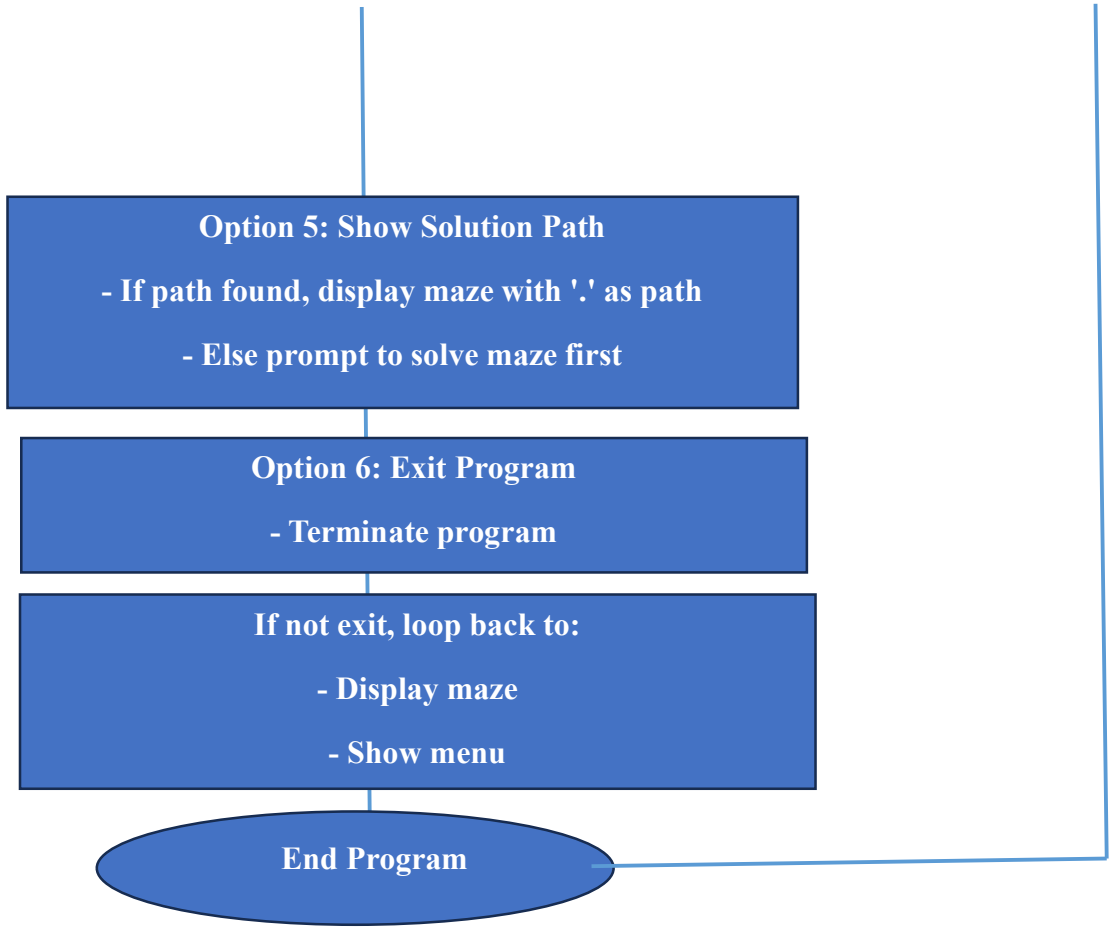



**Figure 01 : Flow Chart (Maze Solver and Generator)**

**Description of Workflow**

1. Start Program – The program starts and prepares needed variables.
2. Choose Input Method – User selects:
    - User Input Maze – Enter maze size and walls for each cell.
    - Auto Maze Generator – Program creates a random maze automatically.
3. Maze Ready – Maze is stored in a 2D grid with wall information.
4. Show Maze – The maze is displayed with Start (S) and End (E).
5. Menu Options – User can:
    - Insert Wall
    - Delete Wall
    - Search Cell Walls
    - Solve Maze (BFS)
    - Show Solution Path
    - Exit Program

6. Perform Action – The program runs the selected option.
7. Repeat Menu – After each action (except Exit), the maze and menu are shown again.
8. End Program – Program stops when the user chooses Exit.

## Source Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <ctime>
using namespace std;
int width, height;
int dx[] = {0, 1, 0, -1};   // Up, Right, Down, Left
int dy[] = {-1, 0, 1, 0};
struct cell {
   bool wall[4] = {true, true, true, true};
   bool visited = false;};
vector<vector<cell>> maze;
vector<vector<cell>> autoGeneratedMazeBackup; // auto generated maze backup
bool isvalid(int x, int y) {
   return (x >= 0 && x < width && y >= 0 && y < height);}
// User Input Maze
void inputMazeFromUser() {
   cout << "Enter maze size (width height): ";
   cin >> width >> height;
   maze.assign(height, vector<cell>(width));
   cout << "Enter each cell walls as number (0-15), bitwise: Up=1, Right=2, Down=4, Left=8\n";
   for (int y = 0; y < height; y++) {
```

```
        cout << "Row " << y+1 << ": ";
        for (int x = 0; x < width; x++) {
            int w; cin >> w;
            while (w < 0 || w > 15) {
                cout << "Invalid! Enter 0 to 15: "; cin >> w;}
            maze[y][x].wall[0] = (w & 1);
            maze[y][x].wall[1] = (w & 2);
            maze[y][x].wall[2] = (w & 4);
            maze[y][x].wall[3] = (w & 8);}}}
void removeWalls(int x1, int y1, int x2, int y2) {
    if (x1 == x2) {
        if (y1 < y2) { // down
            maze[y1][x1].wall[2] = false;
            maze[y2][x2].wall[0] = false;
        } else { // up
            maze[y1][x1].wall[0] = false;
            maze[y2][x2].wall[2] = false;}
    } else if (y1 == y2) {
        if (x1 < x2) { // right
            maze[y1][x1].wall[1] = false;
            maze[y2][x2].wall[3] = false;
        } else { // left
            maze[y1][x1].wall[3] = false;
            maze[y2][x2].wall[1] = false;}}}
// Generate maze using DFS backtracking
void generateMaze(int x, int y) {
    maze[y][x].visited = true;
```

```
    int dirs[] = {0,1,2,3};
    for (int i=3; i>0; i--) {
        int j = rand() % (i+1);
        swap(dirs[i], dirs[j]);}
    for (int i=0; i<4; i++) {
        int dir = dirs[i];
        int nx = x + dx[dir];
        int ny = y + dy[dir];
        if (isvalid(nx, ny) && !maze[ny][nx].visited) {
            removeWalls(x, y, nx, ny);
            generateMaze(nx, ny);}}}
// Print maze
void printMaze() {
    cout << "\n*******  My  Maze  ********\n";
    for (int y=0; y<height; y++) {
        // Print top walls
        for (int x=0; x<width; x++) {
            cout << "+";
            cout << (maze[y][x].wall[0] ? "---" : "   ");}
        cout << "+\n";
        // Print left walls and cell contents
        for (int x=0; x<width; x++) {
            cout << (maze[y][x].wall[3] ? "|" : " ");
            if (y == 0 && x == 0) cout << " S ";
            else if (y == height-1 && x == width-1) cout << " E ";
            else cout << "   ";}
        cout << "|\n";}
```

```
    for (int x=0; x<width; x++) cout << "+---";
    cout << "+\n";}

// Insert wall
void insertWall(int x, int y, int dir) {
    if (!isvalid(x,y)) {
        cout << "Invalid cell!\n";
        return;}
    maze[y][x].wall[dir] = true;
    int nx = x + dx[dir];
    int ny = y + dy[dir];
    if (isvalid(nx, ny)) {
        int opp = (dir + 2) % 4;
        maze[ny][nx].wall[opp] = true;}
    cout << "Wall inserted at (" << x << "," << y << ") towards ";
    if(dir == 0) cout << "Up (dx=0, dy=-1)";
    else if(dir == 1) cout << "Right (dx=1, dy=0)";
    else if(dir == 2) cout << "Down (dx=0, dy=1)";
    else cout << "Left (dx=-1, dy=0)";
    cout << "\n";}
// Delete wall
void deleteWall(int x, int y, int dir) {
    if (!isvalid(x,y)) {
        cout << "Invalid cell!\n";
        return;}
    maze[y][x].wall[dir] = false;
    int nx = x + dx[dir];
```

```
    int ny = y + dy[dir];
    if (isvalid(nx, ny)) {
        int opp = (dir + 2) % 4;
        maze[ny][nx].wall[opp] = false;}
    cout << "Wall deleted at (" << x << "," << y << ") towards ";
    if(dir == 0) cout << "Up (dx=0, dy=-1)";
    else if(dir == 1) cout << "Right (dx=1, dy=0)";
    else if(dir == 2) cout << "Down (dx=0, dy=1)";
    else cout << "Left (dx=-1, dy=0)";
    cout << "\n";}
// Search cell walls
void searchCell(int x, int y) {
    if (!isvalid(x,y)) {
        cout << "Invalid cell!\n";
        return;}
    cout << "Cell (" << x << "," << y << ") walls: ";
    cout << "Up=" << maze[y][x].wall[0] << ", ";
    cout << "Right=" << maze[y][x].wall[1] << ", ";
    cout << "Down=" << maze[y][x].wall[2] << ", ";
    cout << "Left=" << maze[y][x].wall[3] << "\n";}
// Maze Solver with BFS + path reconstruction
struct point { int x,y; };
bool solveMazeWithPath(vector<point> &path) {
    vector<vector<bool>> visited(height, vector<bool>(width, false));
    vector<vector<point>> parent(height, vector<point>(width, {-1,-1}));
    queue<point> q;
    q.push({0,0});
```

```
    visited[0][0] = true;
    bool found = false;

    while (!q.empty()) {
        point p = q.front(); q.pop();
        int x = p.x, y = p.y;
        if (x == width-1 && y == height-1) {
            found = true;
            path.clear();
            for (point cur = p; cur.x != -1 && cur.y != -1; cur = parent[cur.y][cur.x])
                path.push_back(cur);
            reverse(path.begin(), path.end());
            break;}
        for (int dir=0; dir<4; dir++) {
            int nx = x + dx[dir];
            int ny = y + dy[dir];
            if (isvalid(nx, ny) && !visited[ny][nx] && !maze[y][x].wall[dir]) {
                visited[ny][nx] = true;
                parent[ny][nx] = {x,y};
                q.push({nx, ny});}}}
    return found;}
// Print maze with solution path
void printMazeWithPath(const vector<point> &path) {
    cout << "\n#####  Maze with Solution Path  #####\n";
    vector<vector<bool>> isPath(height, vector<bool>(width, false));
    for (auto &p : path) isPath[p.y][p.x] = true;
    for (int y=0; y<height; y++) {
```

```
        for (int x=0; x<width; x++) {
            cout << "+";
            cout << (maze[y][x].wall[0] ? "---" : "   ");}
        cout << "+\n";
        for (int x=0; x<width; x++) {
            cout << (maze[y][x].wall[3] ? "|" : " ");
            if (y == 0 && x == 0) cout << " S ";
            else if (y == height-1 && x == width-1) cout << " E ";
            else if (isPath[y][x]) cout << " . ";
            else cout << "   ";}
        cout << "|\n";}
    for (int x=0; x<width; x++) cout << "+---";
    cout << "+\n";}
int main() {
    srand(time(0));
    cout << "Choose input method:\n1. User Input Maze\n2. Auto Generate Maze\nChoice: ";
    int choice; cin >> choice;
    vector<point> solutionPath;
    bool mazeSolved = false;
    bool allowEdit = false;
    if (choice == 1) {
        inputMazeFromUser();
        allowEdit = true;
        mazeSolved = solveMazeWithPath(solutionPath);
        if (!mazeSolved) {
            cout << "Maze could not be solved with given input.\n";
            cout << "Showing current maze:\n";
```

```
    } else {
        cout << "Maze solved!\n";}
    printMaze();
} else {
    // Auto generate
    cout << "Enter maze size (width height): ";
    cin >> width >> height;
    maze.assign(height, vector<cell>(width));
    generateMaze(0,0);
    // Backup original auto maze to show if needed
    autoGeneratedMazeBackup = maze;
    mazeSolved = solveMazeWithPath(solutionPath);
    cout << "\nAuto generated maze:\n";
    printMaze();
    if (mazeSolved) cout << "Maze is SOLVED.\n";
    else cout << "Maze is NOT solved or unsolvable.\n";}
while (true) {
    cout << "\nMenu:\n";
    if (allowEdit) {
        cout << "1. Insert Wall\n2. Delete Wall\n3. Search Cell\n";}
    cout << "4. Solve Maze\n5. Show Solution Path\n6. Exit\nChoice: ";
    int ch; cin >> ch;
    if (ch == 1 && allowEdit) {
        int x,y,dir;
        cout << "Enter cell x y direction(0=Up,1=Right,2=Down,3=Left): ";
        cin >> x >> y >> dir;
        if (!isvalid(x,y) || dir < 0 || dir > 3) {
```

```
        cout << "Invalid input!\n";
        continue;}
    insertWall(x,y,dir);
    mazeSolved = solveMazeWithPath(solutionPath);
    if (mazeSolved) cout << "After insertion, Maze is SOLVED.\n";
    else cout << "After insertion, Maze is NOT solved or unsolvable.\n";
    printMaze();
} else if (ch == 2 && allowEdit) {
    int x,y,dir;
    cout << "Enter cell x y direction(0=Up,1=Right,2=Down,3=Left): ";
    cin >> x >> y >> dir;
    if (!isvalid(x,y) || dir < 0 || dir > 3) {
        cout << "Invalid input!\n";
        continue;}
    deleteWall(x,y,dir);
    mazeSolved = solveMazeWithPath(solutionPath);
    if (mazeSolved) cout << "After deletion, Maze is SOLVED.\n";
    else cout << "After deletion, Maze is NOT solved or unsolvable.\n";
    printMaze();
} else if (ch == 3 && allowEdit) { int x,y;
    cout << "Enter cell x y to search walls: ";
    cin >> x >> y;
    if (!isvalid(x,y)) {
        cout << "Invalid cell!\n"; continue;}
    searchCell(x,y);
    printMaze();
    mazeSolved = solveMazeWithPath(solutionPath);
```

```
    if (mazeSolved) cout << "Maze is currently SOLVED.\n";
    else cout << "Maze is NOT solved yet or unsolvable.\n";
} else if (ch == 4) {
    mazeSolved = solveMazeWithPath(solutionPath);
    if (mazeSolved) cout << "Maze solved successfully!\n";
    else cout << "Maze is NOT solved or unsolvable.\n";
} else if (ch == 5) {
    if (!mazeSolved) cout << "Solve the maze first (option 4).\n";
    else printMazeWithPath(solutionPath);
} else if (ch == 6) {
    cout << "Exiting...\n";
    break;
} else {
    cout << "Invalid choice!\n";}}}
```

**Result**:

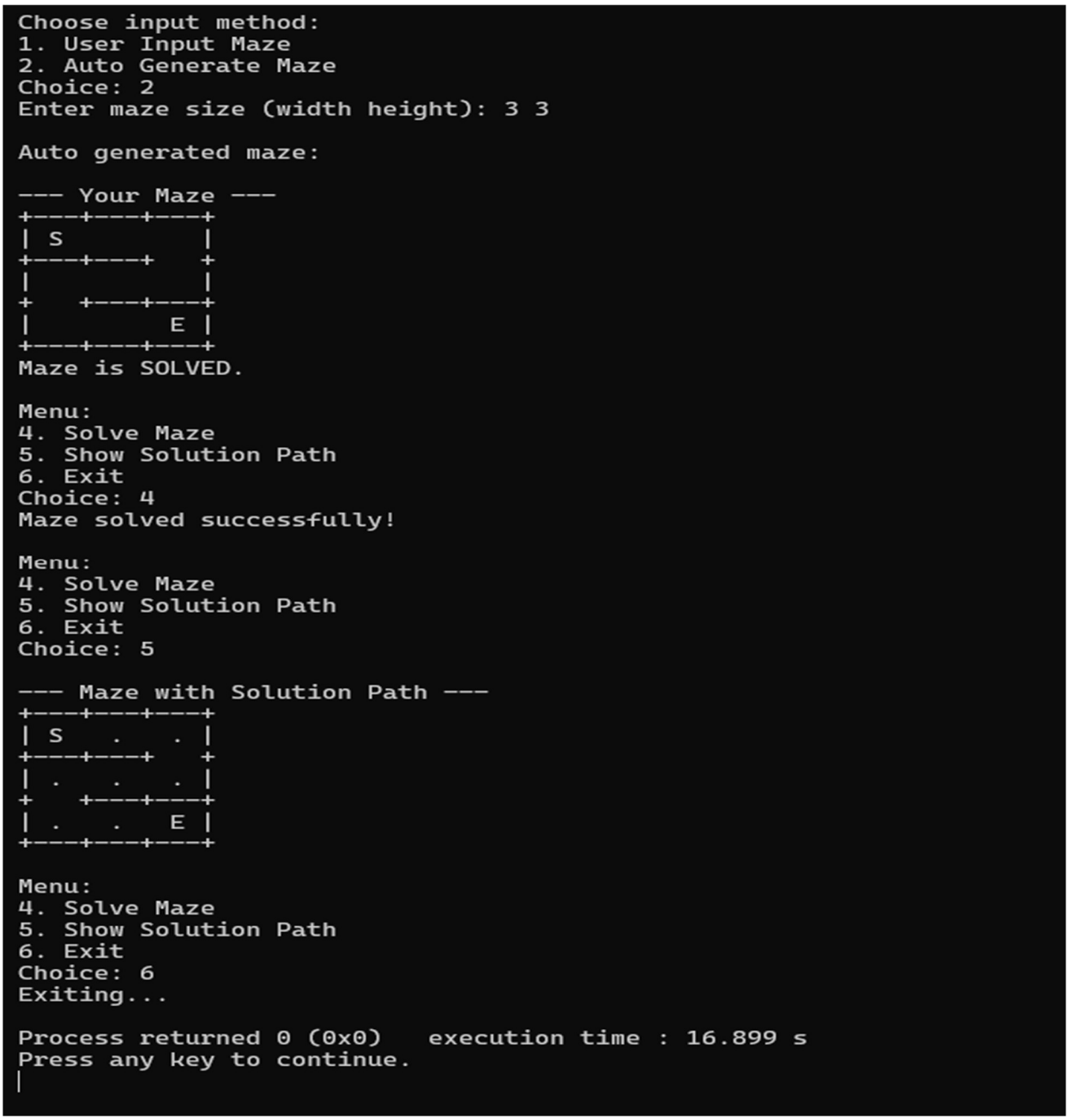



**Figure 02: Auto-generated maze solved using BFS**

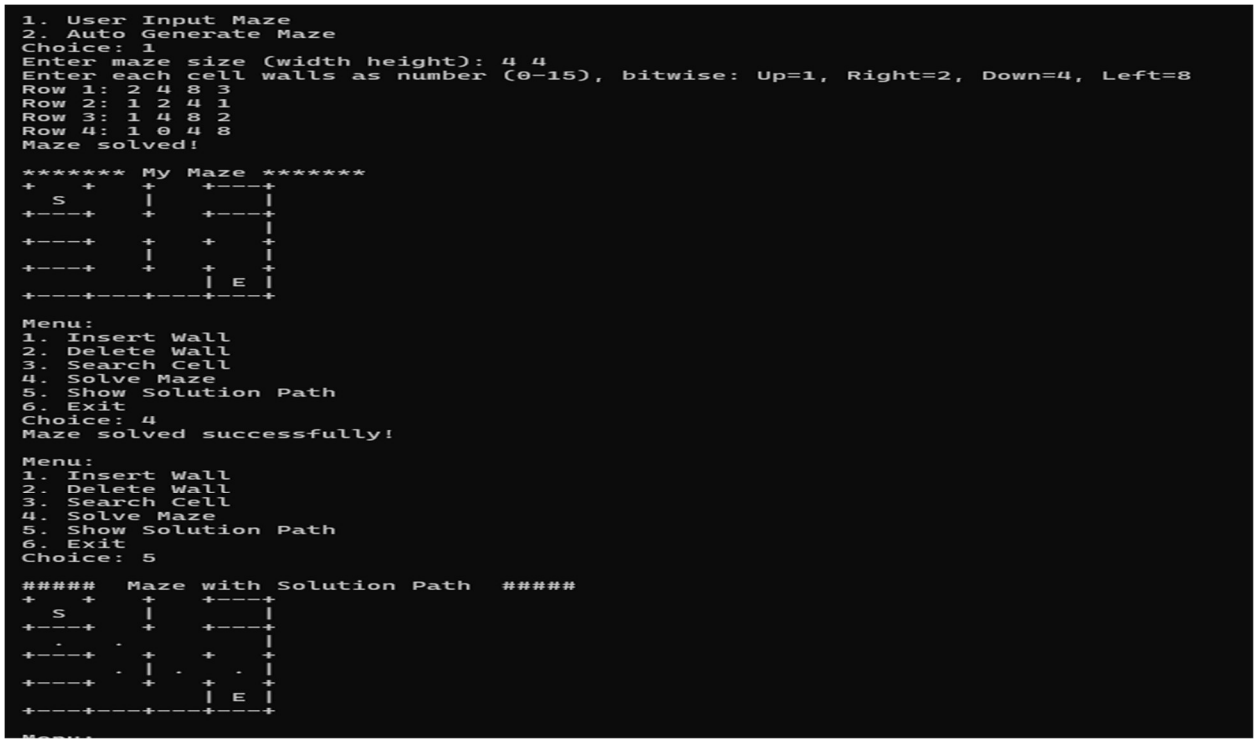



**Figure 03: User-input maze walls**

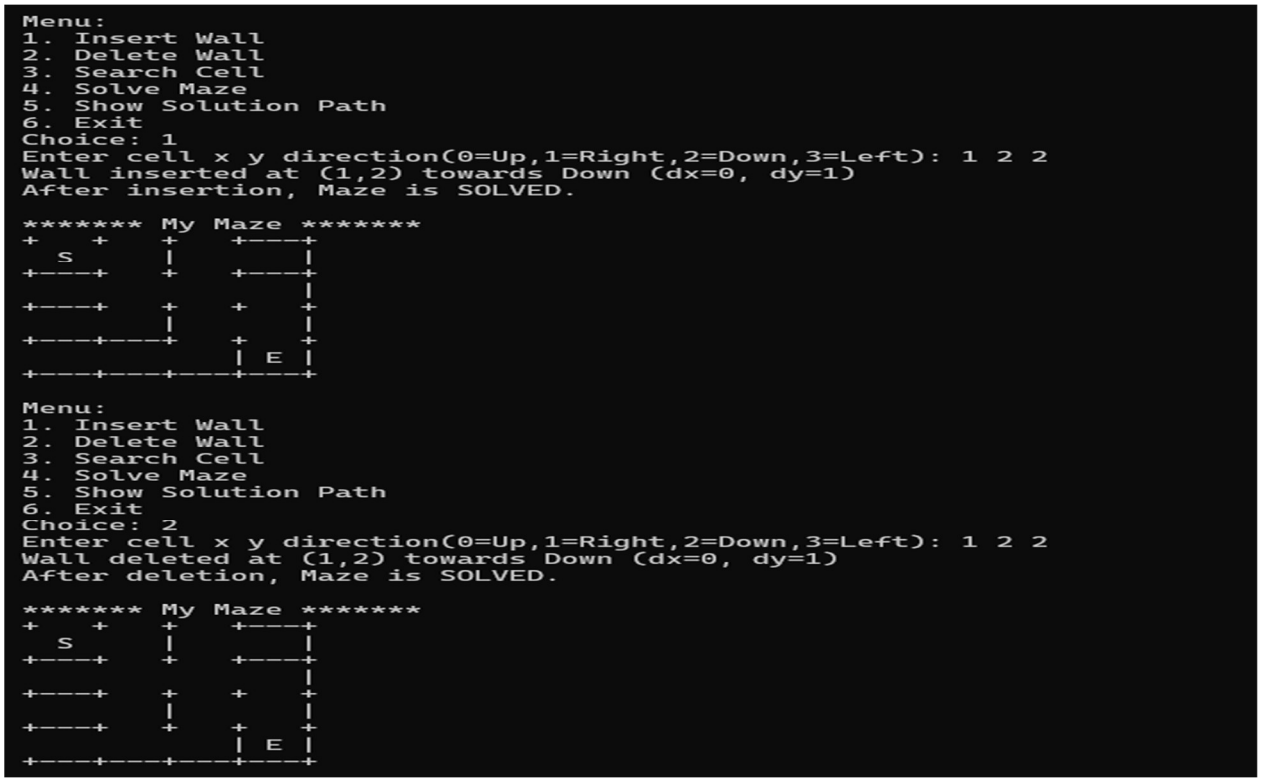



**Figure 04: User-input maze with inserted and deleted walls**

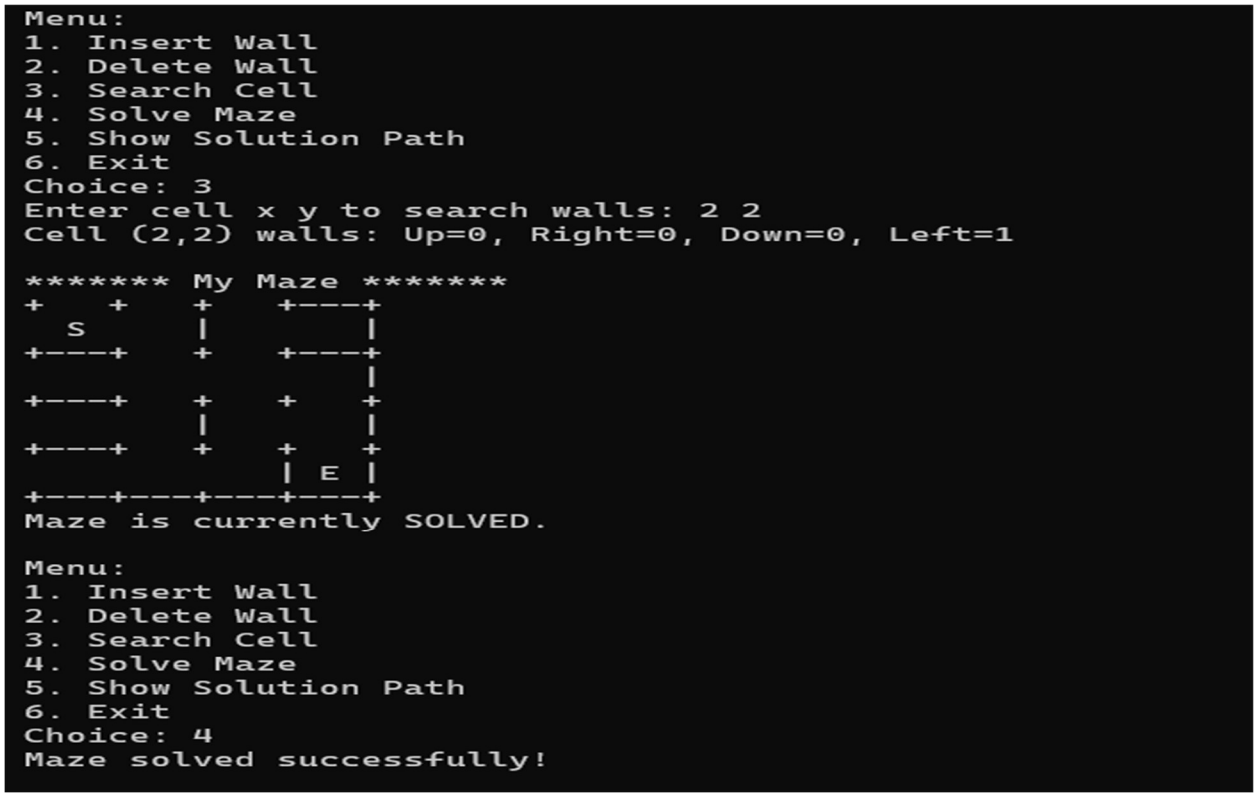



**Figure 05 : Search Cells**

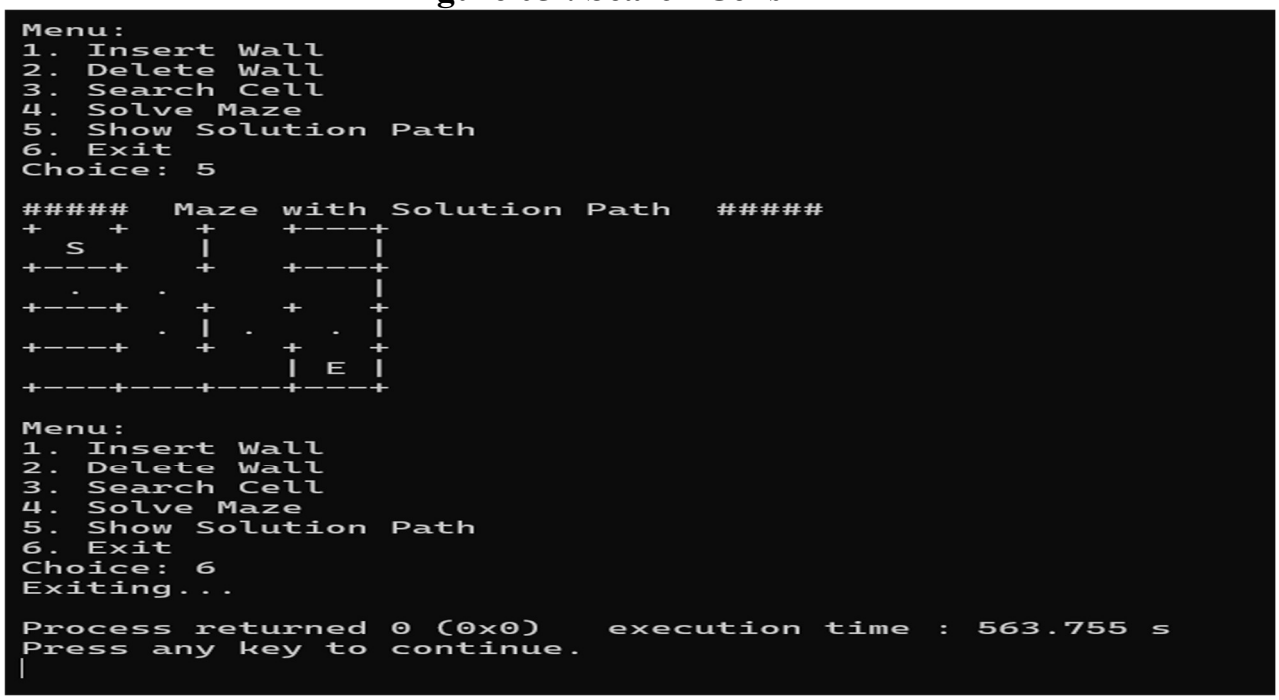



**Figure 06 : Maze Generator And Solver (with solution path(.))**

## Conclusion

This project successfully implements a Maze Generator and Solver using C++. It allows the maze to be created either by user input or automatically using the recursive backtracking method. The user can modify the maze by inserting or deleting walls, search for walls in any cell, and solve the maze using the Breadth-First Search (BFS) algorithm to find the shortest path. The program also displays the solution path visually in the console. In the future, a graphical view can be added, more smart solving methods like A* (pathfinding algorithm) can be used, or the maze can be turned into a fun game or learning tool.

## *References*


- Daniel Shiffman. *Maze Generation with Recursive Backtracking* (The Coding Train). (Accessed: Aug 3, 2025)
https://thecodingtrain.com/challenges/10-maze-generator

- GeeksforGeeks. *Find shortest path in a maze*. (Accessed: Aug 3, 2025)
https://www.geeksforgeeks.org/shortest-path-in-a-binary-maze/

- Wikipedia. *Maze solving algorithm*. (Accessed: Aug 3, 2025)
https://en.wikipedia.org/wiki/Maze_solving_algorithm

- Baeldung on Computer Science. *A* Search Algorithm in C++*. (Accessed: Aug 3, 2025)
https://www.baeldung.com/cs/a-star-algorithm.