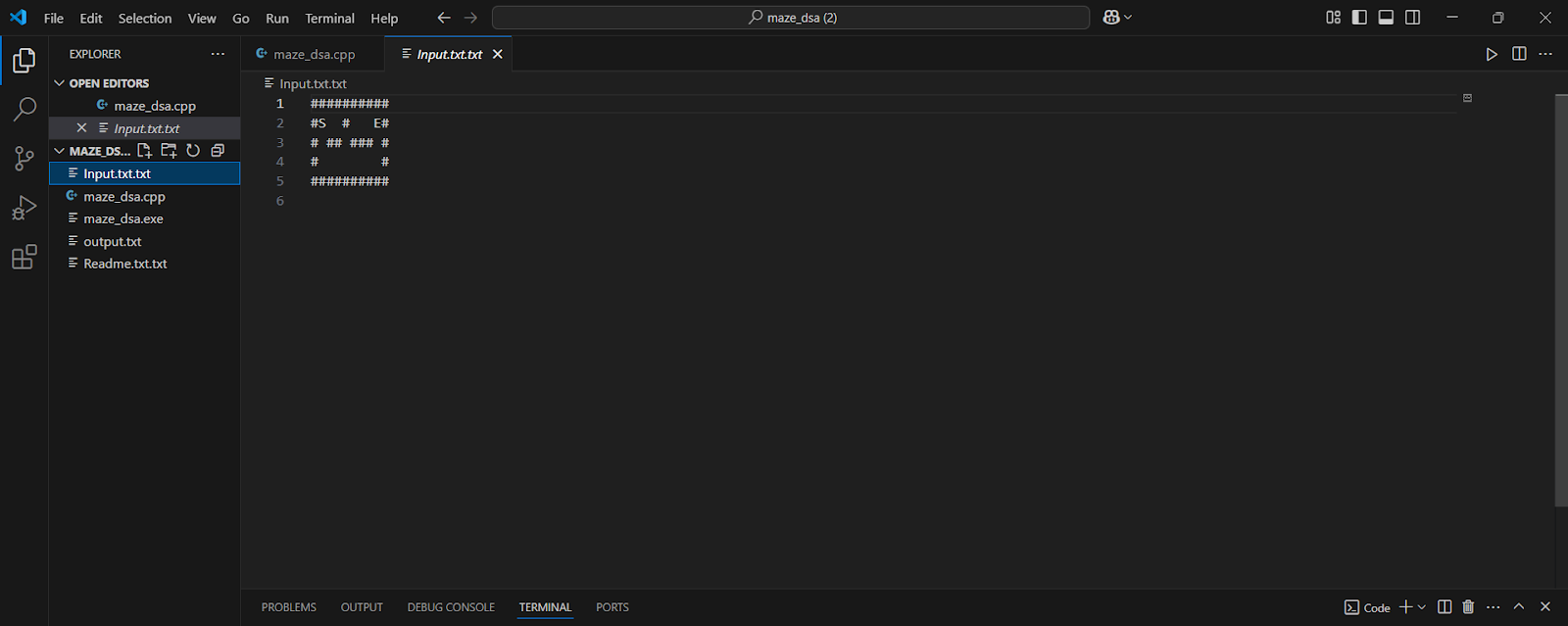
**Maze Solver Game Report**

# Problem Description

The Maze Solver Game operates as a C++ application which imports text-based maze layouts from files to locate paths starting from S through E while writing solution paths using asterisks. The game displays walls through '#' characters and keeps spaces while showing empty areas without symbols. The project shows essential data structures together with algorithmic methods while utilizing real-world examples.



*Figure 1: Input maze as read from file. Start point is marked with 'S' and end point with 'E'. Walls are represented with '#'.*

# System Architecture

* The application implements a basic command-line system structure.

* Input Parser: Reads the maze layout from a .txt file into memory.

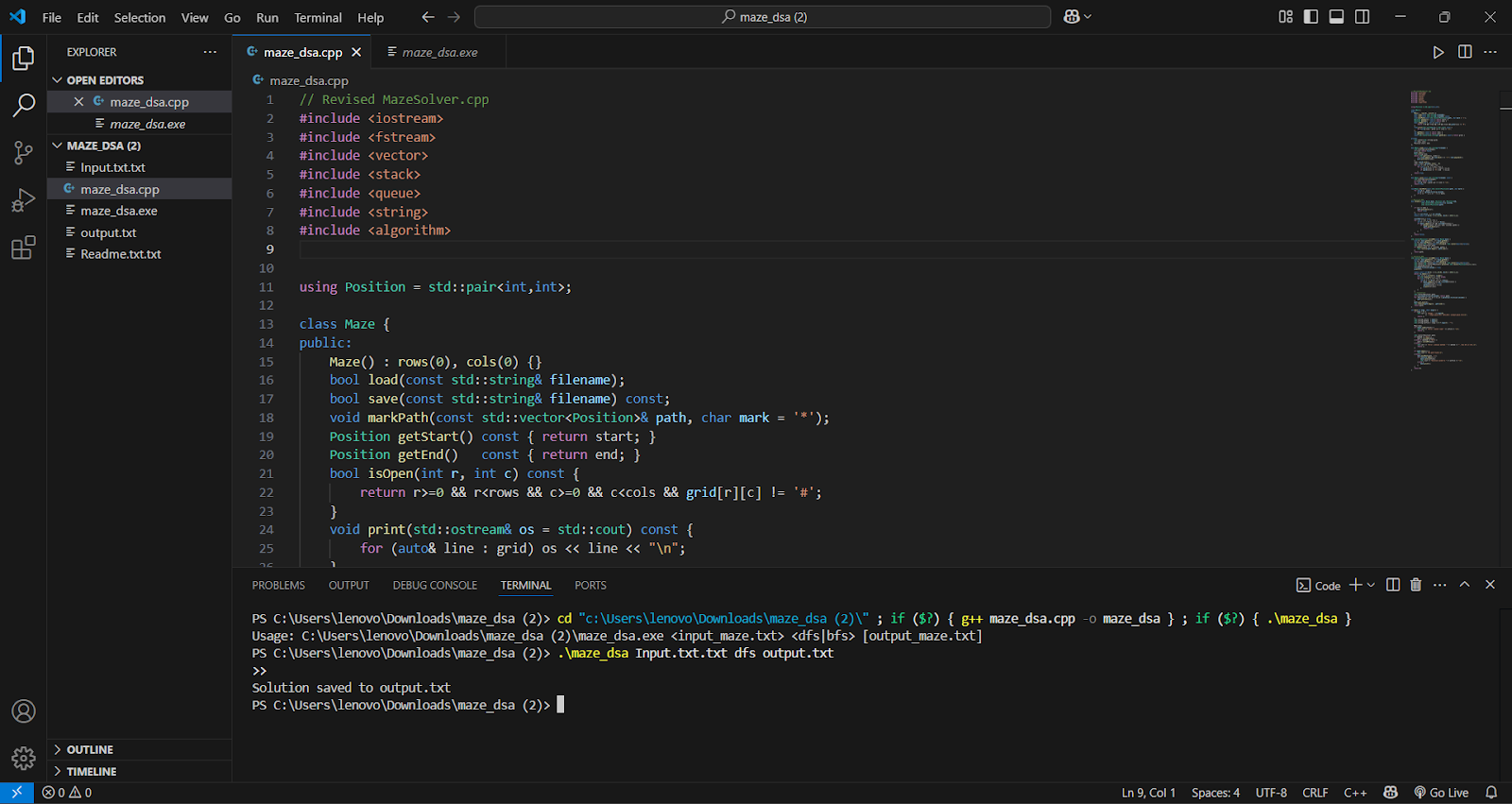
* The maze data is stored within a two-dimensional string array type named vector<string>.

**Solver Module:**

* The algorithm performs recursive backtracking exploration to search paths in DFS Solver.

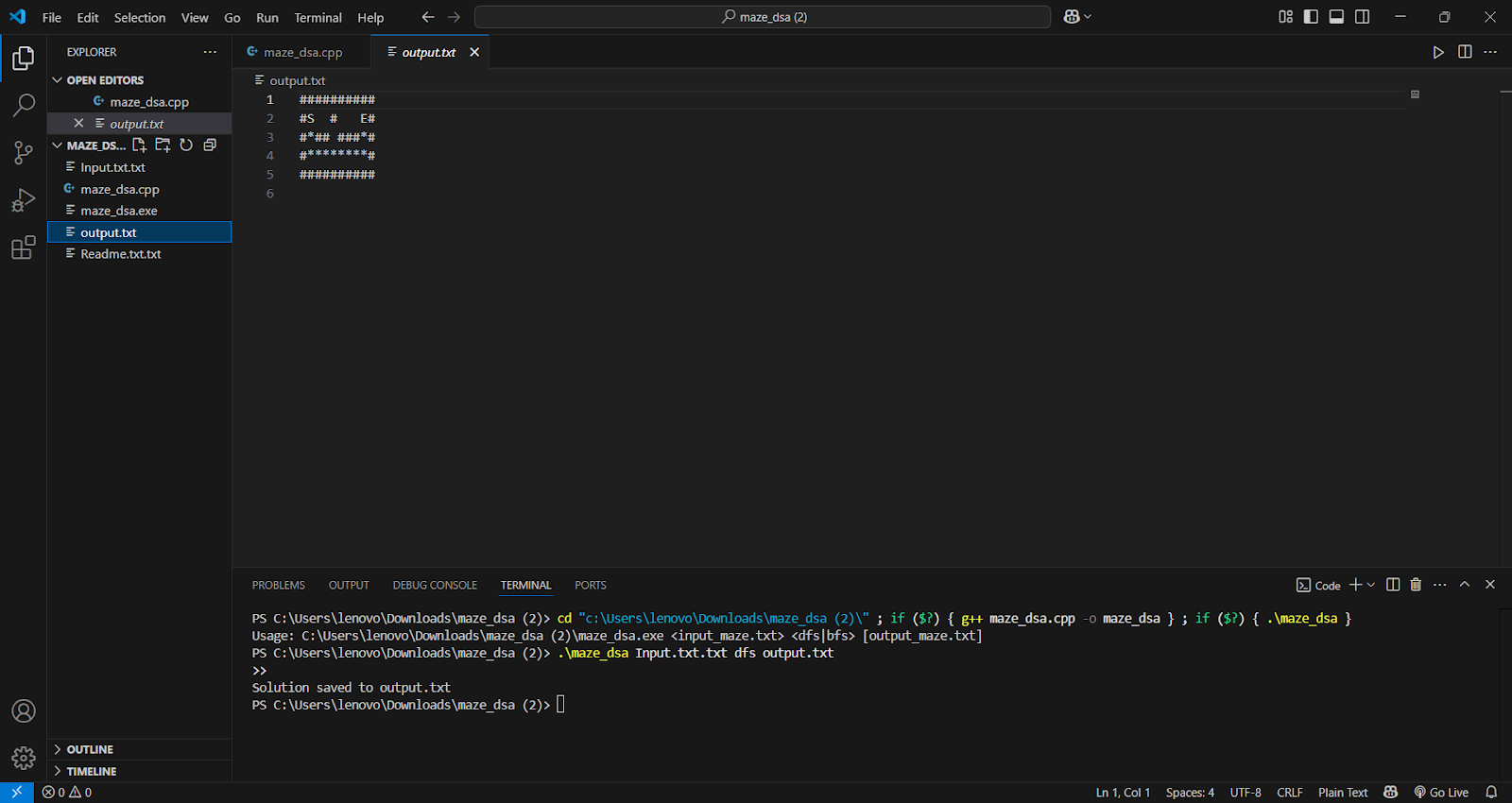
* BFS Solver functions by exploring maze paths sequentially from bottom to top using queue-based implementation.

* Output Handler provides two options to display the path: direct print to screen output or saving the grid to an external file.



*Figure 2: Terminal output showing successful execution of the maze solver using DFS and saving the solution to output.txt.*

[ input.txt ] → [ Maze.load() ] → [ solveDFS() / solveBFS() ] → [ Maze.markPath() ] → [ stdout / file ]



*Figure 3: Solved maze output with path from S to E marked using*

# Data Structure Usage

Each row of the maze appears as Grid: std::vector<std::string>.

The class Position implements pair type to store positions through its pair<int,int> elements.

The visited cells are tracked using std::vector<std::vector<bool>>; while Matrix (DFS) performs its operation.

The BFS requires a Parent Matrix consisting of std::vector<std::vector<Position>> structure to generate path reconstruction.

The depth-first search technique utilizes the call stack as well as recursive approaches for its backtracking operations.

The iterative traversal of breadth-first search depends on std::queue<Position> as its main data structure.

# Key Code Snippets

**Maze Loading:**

|  |
| --- |
| bool Maze::load(const std::string& filename) { std::ifstream in(filename); if (!in) return false;  grid.clear(); std::string line; while (std::getline(in, line)) {  if (!line.empty() && line.back()=='\r') line.pop\_back(); |

grid.push\_back(line);

}

// find S and E for (int i = 0; i < rows; ++i) for (int j = 0; j < cols; ++j) { if (grid[i][j]=='S') start={i,j};

if (grid[i][j]=='E') end={i,j};

}

return true;

}

**Recursive DFS**

|  |
| --- |
| bool dfsUtil(const Maze& m, Position cur, Position end, vector<vector<bool>>& visited, vector<Position>& path) { if (cur==end) { path.push\_back(cur); return true; } visited[cur.first][cur.second] = true; for (auto &dir : directions) {  Position next = {cur.first+dir[0], cur.second+dir[1]};  if (m.isOpen(next.first,next.second) && !visited[next.first][next.second]) { if (dfsUtil(m, next, end, visited, path)) { path.push\_back(cur); return true;  }  }  }  return false;  } |

**Iterative BFS**

|  |
| --- |
| std::vector<Position> solveBFS(const Maze& m) { queue<Position> q; q.push(m.getStart()); parent[r][c] = {-1,-1}; while (!q.empty()) {  auto [r,c] = q.front(); q.pop(); if (make\_pair(r,c)==m.getEnd()) break; for (auto& d : directions) { Position n = {r+d[0],c+d[1]};  if (m.isOpen(n.first,n.second) && !visited[n.first][n.second]) { visited[n.first][n.second]=true; parent[n.first][n.second]={r,c}; q.push(n);  }  } |

}

// reconstruct path...

}

**5. Team Contributions**

