PROJECT BASED LEARNING

On

MAZE SOLVE R

SUBMITTED BY
Jiya Siwach[1/23/SET/BCS/519]
Tushant Mendiratta [1/23/SET/BCS/521]
Vikas Thakran [1/23/SET/BCS/520]

UNDER THE GUIDANCE OF
Dr. Swati Hans

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Manav Rachna International Institute of Research and Studies

ACADEMIC YEAR- 2025-2026

ABSTRACT

Maze solving is one of the classic applications in algorithm design and analysis, commonly used to understand graph traversal concepts in computational problem solving. This project focuses on implementing a Maze Solver using Depth First Search (DFS) and Breadth First Search (BFS) algorithms to navigate from a starting point to a target destination within a 2D maze environment.The maze is represented using a two-dimensional matrix, where valid paths and obstacles are distinctly identified. DFS explores deeper recursive paths, while BFS uses a queue to ensure the shortest path. Both algorithms are implemented using the C programming language. This project strengthens the understanding of algorithmic thinking, recursion, queue-based traversal, and performance comparison of search algorithms under Design and Analysis of Algorithms (DAA).

INTRODUCTION

Maze solving is a well-known computational problem that demonstrates how algorithmic decision-making can be applied to structured environments. A maze consists of interconnected pathways and blocked routes, requiring a systematic approach to navigate from a starting point to a final destination. This makes maze solving an ideal problem for studying graph algorithms, pathfinding logic, and traversal techniques in computer science.

This project focuses on implementing a Maze Solver using Depth First Search (DFS) and Breadth First Search (BFS) in the C programming language. DFS uses a recursive approach to explore deeper paths and backtracks when no further movement is possible, making it suitable for exploring all reachable areas. BFS, in contrast, operates level-by-level using a queue and guarantees the shortest path in an unweighted maze.

The maze is represented as a two-dimensional matrix where walls, free paths, visited nodes, and the final solution path are clearly marked. The implementation strengthens understanding of algorithm design, recursion, queues, data structures, and time-space complexity principles under Design and Analysis of Algorithms (DAA). Overall, the Maze Solver provides practical insight into how theoretical search algorithms can be applied to real-world pathfinding problems.

OBJECTIVES

The primary objective of this project is to design and implement an automated maze-solving system using Depth First Search (DFS) and Breadth First Search (BFS) algorithms in the C programming language. The project aims to represent the maze using a two-dimensional matrix and apply traversal logic to efficiently navigate from a defined starting position to a target endpoint without violating maze boundaries or revisiting already explored paths. Another important objective is to compare the behavior and performance of DFS and BFS in terms of solution accuracy, execution style, and path optimality, highlighting how BFS ensures the shortest route while DFS focuses on exploring all possible routes through recursion and backtracking. Additionally, the project seeks to deepen understanding of core concepts from Design and Analysis of Algorithms (DAA), data structures such as queues, recursion stacks, and visited matrices, and real-world pathfinding strategies. Visualizing the maze and its final solution using console-based output further supports learning by demonstrating the complete traversal process in a clear and interpretable form.

SYSTEM ANALYSIS

System analysis plays an essential role in understanding the functional requirements, constraints, and algorithmic behavior of the Maze Solver. Since maze solving involves navigating structured pathways while avoiding obstacles, the system must employ an efficient

and reliable traversal strategy. The Maze Solver is designed to function on a 2D grid where each cell represents either a walkable path or a blocked region. To achieve efficient navigation, the system utilizes both Depth First Search (DFS) and Breadth First Search (BFS), allowing comparison between exhaustive traversal and shortest-path discovery. DFS is implemented to explore deep routes using recursion and backtracking, while BFS follows a level-wise exploration using a queue to guarantee the shortest solution path in an unweighted maze. The system incorporates boundary validation, visited tracking, and parent tracing to maintain accuracy and prevent repeated exploration. The maze is represented using a two-dimensional array, with characters specifying walls, open paths, the starting cell, and the final goal. Supporting data structures such as queues, recursion stacks, visited matrices, and parent tracking arrays are used to enable traversal and solution reconstruction. The proposed system automates the process of finding valid and optimal paths, eliminating the inefficiency and inaccuracy associated with manual pathfinding, and visualizes the solution clearly through console-based output.

METHODOLOGY

The methodology followed in this project involves a structured approach to designing and developing the Maze Solver using DFS and BFS algorithms. The process begins with representing the maze in the form of a two-dimensional matrix, where each cell is assigned a value indicating whether it is a path, a wall, the start point, or the destination. Once the maze structure is defined, a validation function is implemented to ensure that every movement made by the algorithm remains within the maze boundaries, avoids walls, and does not revisit previously explored cells. After establishing input representation and movement rules, the Depth First Search algorithm is implemented using recursion, allowing the solver to explore one path deeply before backtracking when no further movement is possible. In parallel, the Breadth First Search algorithm is implemented using a queue, enabling the solver to traverse the maze level-by-level and ensuring that the shortest available path is found when the destination is reached. To support BFS path reconstruction, a parent tracking mechanism is included to trace the final route backward from the endpoint to the start. Throughout execution, both algorithms mark visited cells and progressively build the solution path, which is finally displayed through console visualization. This step-by-step methodology ensures that both traversal strategies are accurately implemented, evaluated, and compared in terms of performance, complexity, and solution quality.

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int dr[] = {-1, 1, 0, 0};
int dc[] = {0, 0, -1, 1};

struct Node {
    int row, col;
    struct Node* next;
};

struct Queue {
    struct Node* front;
    struct Node* rear;
};

struct Node* createNode(int row, int col) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->row = row;
    newNode->col = col;
    newNode->next = NULL;
    return newNode;
}

void initQueue(struct Queue* q) {
    q->front = q->rear = NULL;
```

```
}

int isEmpty(struct Queue* q) {
    return q->front == NULL;
}

void enqueue(struct Queue* q, int row, int col) {
    struct Node* newNode = createNode(row, col);
    if (q->rear == NULL) {
        q->front = q->rear = newNode;
        return;
    }
    q->rear->next = newNode;
    q->rear = newNode;
}

struct Node* dequeue(struct Queue* q) {
    if (isEmpty(q)) return NULL;
    struct Node* temp = q->front;
    q->front = q->front->next;
    if (q->front == NULL) q->rear = NULL;
    return temp;
}

int isValid(int row, int col, int visited[N][N], char maze[N][N]) {
    return (row >= 0 && row < N && col >= 0 && col < N && !visited[row][col] &&
maze[row][col] != '1');
}
```

```
void printMaze(char maze[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%c ", maze[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

int dfs(int row, int col, int visited[N][N], char maze[N][N], char solution[N][N], int endRow, int endCol) {
    if (row == endRow && col == endCol) {
        solution[row][col] = 'D';  // Mark the end in the solution
        return 1;
    }

    visited[row][col] = 1;
    solution[row][col] = 'D';  // Mark the current cell in the path

    for (int d = 0; d < 4; d++) {
        int nr = row + dr[d];
        int nc = col + dc[d];
        if (isValid(nr, nc, visited, maze)) {
            if (dfs(nr, nc, visited, maze, solution, endRow, endCol)) {
                return 1;  // Path found
            }
        }
    }
```

```
    solution[row][col] = maze[row][col];
    return 0;  // No path from here
}

int bfs(int startRow, int startCol, int endRow, int endCol, char maze[N][N], char
solution[N][N]) {
    int visited[N][N] = {0};
    int parent[N][N][2];  // To store parent coordinates: parent[row][col][0] = parent row, [1] =
parent col
    memset(parent, -1, sizeof(parent));  // Initialize parent to -1

    struct Queue q;
    initQueue(&q);
    enqueue(&q, startRow, startCol);
    visited[startRow][startCol] = 1;
    int found = 0;
    while (!isEmpty(&q)) {
        struct Node* current = dequeue(&q);
        int row = current->row;
        int col = current->col;
        free(current);

        if (row == endRow && col == endCol) {
            found = 1;
            break;
        }
        for (int d = 0; d < 4; d++) {
            int nr = row + dr[d];
            int nc = col + dc[d];
            if (isValid(nr, nc, visited, maze)) {
```

```
                visited[nr][nc] = 1;
                enqueue(&q, nr, nc);
                parent[nr][nc][0] = row;
                parent[nr][nc][1] = col;
            }
        }
    }

    if (found) {
        int r = endRow, c = endCol;
        while (r != -1 && c != -1) {
            solution[r][c] = 'B';  // Mark the shortest path
            int pr = parent[r][c][0];
            int pc = parent[r][c][1];
            r = pr;
            c = pc;
        }
        return 1;
    }
    return 0;  // No path found
}

int main() {
    // Define the maze: 1 = wall, 0 = open path, S = start, E = end
    char maze[N][N] = {
        {'S', '0', '1', '0', '0'},
        {'1', '0', '1', '0', '1'},
        {'0', '0', '0', '0', '0'},
        {'1', '1', '1', '1', '0'},
```

```
    {'0', '0', '0', '0', 'E'}
};

// Find start and end positions
int startRow, startCol, endRow, endCol;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if (maze[i][j] == 'S') {
            startRow = i;
            startCol = j;
        } else if (maze[i][j] == 'E') {
            endRow = i;
            endCol = j;
        }
    }
}

printf("Original Maze:\n");
printMaze(maze);

int visitedDFS[N][N] = {0};
char dfsSolution[N][N];
memcpy(dfsSolution, maze, sizeof(maze));  // Copy maze to solution
if (dfs(startRow, startCol, visitedDFS, maze, dfsSolution, endRow, endCol)) {
    printf("DFS Path Found:\n");
    printMaze(dfsSolution);
} else {
    printf("No path found using DFS.\n\n");
}
```

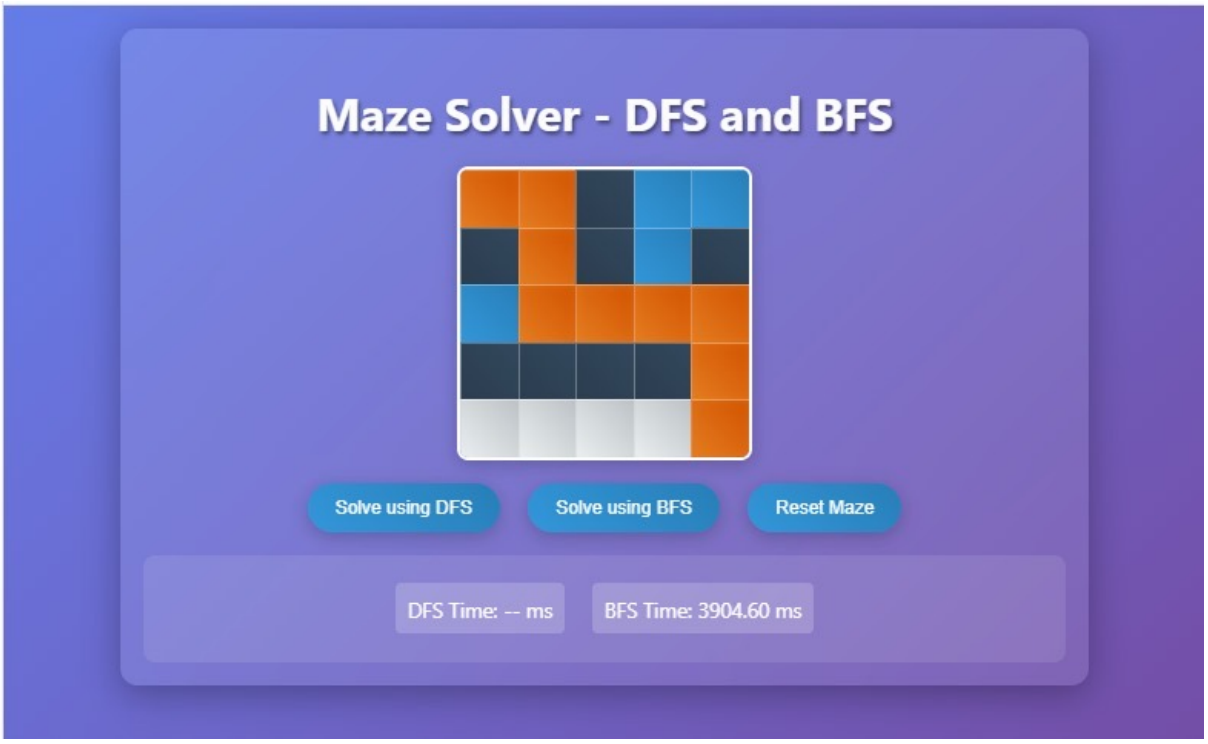
```
    char bfsSolution[N][N];
    memcpy(bfsSolution, maze, sizeof(maze));  // Copy maze to solution
    if (bfs(startRow, startCol, endRow, endCol, maze, bfsSolution)) {
        printf("BFS Shortest Path Found:\n");
        printMaze(bfsSolution);
    } else {
        printf("No path found using BFS.\n");
    }

    return 0;
}
```

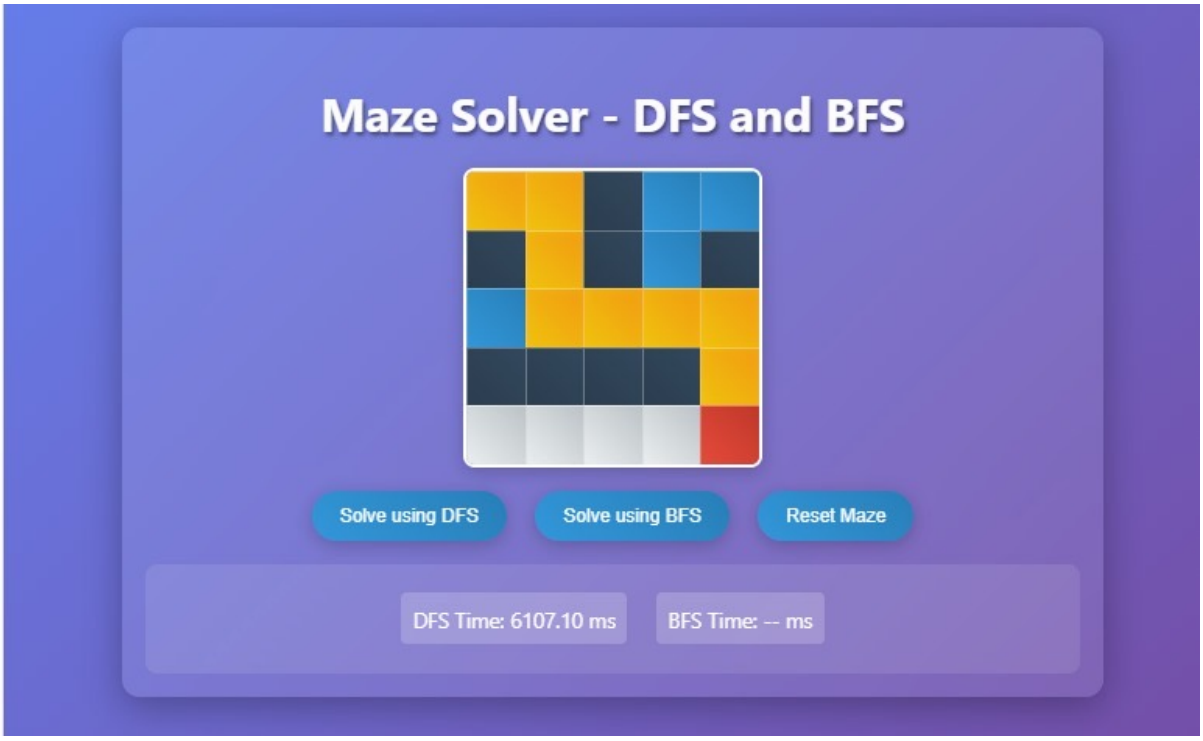
RESULTS AND DISCUSSION

The implementation of the Maze Solver demonstrates successful execution of both Depth First Search (DFS) and Breadth First Search (BFS) algorithms in navigating a maze represented as a two-dimensional grid. The results show that DFS is effective in identifying a valid route from the start cell to the destination by recursively exploring deeper paths before backtracking when faced with dead-ends. This behavior results in DFS finding a correct solution, although not necessarily the shortest one, especially in complex mazes with multiple branching paths. In contrast, BFS consistently identifies the shortest path due to its level-wise exploration strategy. By traversing neighboring cells step by step, BFS ensures that the first time the goal is reached corresponds to the minimum number of movements required.

During execution, both algorithms successfully validated boundaries, avoided walls, and prevented repeated traversal of previously visited cells. The console visualization clearly highlighted walls, free spaces, visited positions, and the final solution path, allowing the behavior of both algorithms to be observed in real time. Comparative analysis indicates that while DFS uses less memory due to its recursive approach, it may require more time and redundant exploration in larger mazes. Conversely, BFS uses additional memory to manage its queue and path-tracking structure, but performs more efficiently in identifying optimized paths. Overall, the results validate the theoretical differences between the two algorithms and highlight their applicability based on the nature and requirements of the maze-solving task.

OUTPUTS


Maze Solver - DFS and BFS
Solve using DFS
Solve using BFS
Reset Maze
DFS Time: -- ms
BFS Time: 3904.60 ms


Maze Solver - DFS and BFS
Solve using DFS
Solve using BFS
Reset Maze
DFS Time: 6107.10 ms
BFS Time: -- ms

CONCLUSION

The Maze Solver project demonstrates the effective implementation of two fundamental graph traversal algorithms—Depth First Search (DFS) and Breadth First Search (BFS)—for solving a structured maze represented as a two-dimensional grid. Through this project, the theoretical concepts learned in the Design and Analysis of Algorithms (DAA) curriculum were applied to a real-world style problem, allowing practical understanding of recursion, queue-based traversal, search optimization, and algorithmic efficiency. During execution, DFS proved suitable for exploring deeper or complex routes and successfully identified a valid solution path, although not always the shortest one. BFS, on the other hand, consistently produced the shortest path due to its systematic level-by-level exploration strategy and parent-tracking mechanism. This difference highlights how algorithm choice impacts performance depending on the complexity and constraints of the problem.

The project also reinforced the importance of data structures such as two-dimensional arrays, visited matrices, recursion stacks, and queues in guiding algorithm flow and ensuring accuracy. Console-based visualization provided a clear representation of traversal order, visited cells, and final routes, enhancing conceptual understanding and aiding in algorithm comparison. Overall, the development of this Maze Solver strengthened computational thinking, problem-solving abilities, and understanding of time and space trade-offs in algorithmic design. The project provides a strong foundation for further advancements, such as GUI-based maze rendering, randomized maze generation, implementation of heuristic-based algorithms like A*, or extending the system into robotics or autonomous path navigation applications.