Members:

Bilal Khan: 2023250
Ahmed Azzam: 2023064

Wazir Awais Haidar: 2023758

Project Report: Data Structures Application in Pathfinding Algorithm (Flood Fill Method)

1. Introduction

In this project, I implemented a *pathfinding algorithm* based on the *Flood Fill* technique. The objective was to find a path through a maze using efficient data structures studied in class — namely *Linked Lists, **Queues, **Stacks, and **Sorting* concepts. These structures were critical for handling movement, storage of positions, and exploration order.

---

2. Application of Class Concepts

A. Queue (Main structure used for Flood Fill)

Flood fill works like a *Breadth-First Search (BFS)* algorithm, which naturally uses a *queue* to explore the maze level-by-level.

- *Implementation Concept:*
  - Push the starting cell into a queue.
  - Repeatedly pop a cell, explore its neighbors (up, down, left, right).
  - Push any unvisited neighbor into the queue.

Sample Code Snippet:

```cpp
#include <queue>

struct Cell {
    int x, y;
};

std::queue<Cell> q;
bool visited[100][100];

void floodFill(int startX, int startY) {
    q.push({startX, startY});
    visited[startX][startY] = true;

    while (!q.empty()) {
        Cell current = q.front();
        q.pop();

        // Check neighbors (up, down, left, right)
        // Push valid neighbors to queue
    }
}
```

Time Complexity:

- O(N × M) for a grid of N rows and M columns.

Space Complexity:

- O(N × M) due to the queue and visited array.

B. Stack (Optional for DFS-style Pathfinding)

If using a *Depth-First Search (DFS)* version of flood fill, a *stack* replaces the queue.

- *Implementation Concept:*
  Stack holds the current cell. When moving forward, push the next cell. When stuck, pop back.

Sample Code Snippet:

```cpp
#include <stack>

std::stack<Cell> s;

void floodFillDFS(int startX, int startY) {
    s.push({startX, startY});
    visited[startX][startY] = true;

    while (!s.empty()) {
        Cell current = s.top();
        s.pop();

        // Explore neighbors
    }
}
```

C. Linked List (Path Reconstruction)

After reaching the destination, we reconstruct the path using *linked lists* to store the sequence of steps.

- *Implementation Concept:*
  - Store each move (cell) in a linked list node while backtracking.
  - Final list shows the shortest or a valid path.

Sample Code Snippet:

```cpp
struct PathNode {
    int x, y;
    PathNode* next;
};

PathNode* pathHead = nullptr;

void addStepToPath(int x, int y) {
    PathNode* newNode = new PathNode{x, y, pathHead};
    pathHead = newNode;
}
```

D. Sorting (Optional for Optimizing Movements)

If multiple paths are available, *sorting* neighbors (for example, based on distance to goal) can make flood fill smarter.

- Implementation Concept:

- Sort neighbors based on heuristic distance (closer cells first).

(No compulsory sorting in simple flood fill, but useful in optimized pathfinding like A)

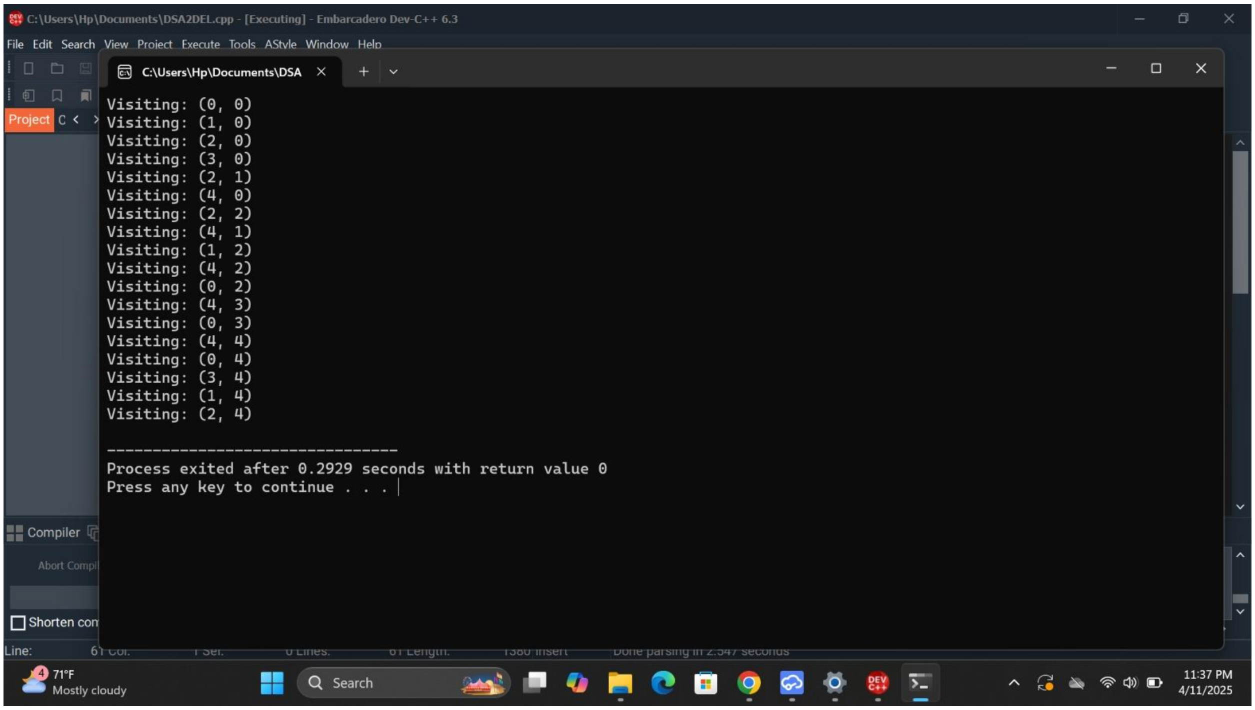
---

3. Efficiency Analysis

Data Structure	Use in Project	Time Complexity	Space Complexity
Queue	BFS traversal of maze	O(N × M)	O(N × M)
Stack	DFS traversal (optional)	O(N × M)	O(N × M)

| Linked List | Path reconstruction storage | O(P) where P is path length | O(P) |

4. Implementation:



| Sorting | Optimizing neighbor order (optional) | Depends on sort (O(k log k)) | O(k) |

4. Conclusion:

Through this project, I practically applied the concepts of *queues, **stacks, **linked lists, and **sorting* studied in class. These structures were critical in efficiently managing the exploration of a maze and constructing a valid path from start to goal. The project not only reinforced my theoretical understanding but also demonstrated how different data structures interact to solve real-world problems.

-