**A**

**Mini Project Report**

**Data Structures and Algorithms Laboratory**

**on**

**“Maze Solver in C++”**

**Submitted to**

****

**Department of Computer Engineering**

Amrutvahini College of Engineering, Sangamner

Submitted By

1. Sarang Deshpande - 2116

2. Uday Dherange - 2117

3. Sai Abhang - 2101

4. Akshad Ambekar - 2103

**Year: 2024-25 (Sem-II)**

**CERTIFICATE**

Amrutvahini Sheti & Shikshan Vikas Sanstha’s

Amrutvahini College of Engineering, Sangamner

Savitribai Phule Pune University

****

This is to certify that the students

Sarang Deshpande – 2116

Uday Dherange – 2117

Akshad Ambekar – 2103

Sai Abhang – 2101

have successfully completed their project, entitled

“Maze Solver in C++”

Under the Guidance of

Prof. K. U. Rahane

During the academic year 2024 – 25

Prof. K. U. Rahane Dr. S. K. Sonkar

**Acknowledgement**

I would like to express my heartfelt gratitude to my guide, [Guide's Name], for their unwavering support, valuable feedback, and insightful suggestions throughout the course of this project. I am also thankful to our project coordinator and all the esteemed faculty members of the department for their encouragement, guidance, and continuous assistance. I extend my appreciation to my colleagues and friends who shared their knowledge and lent their support during critical stages. Last but not least, I am immensely grateful to my family, whose constant motivation and belief in my abilities helped me stay focused and committed to the successful completion of this work.

**Index**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Title of the Content** | **Page No.** |
| 1 | Certificate | 2 |
| 2 | Acknowledgement | 3 |
| 3 | Index | 4 |
| 4 | Abstract | 5 |
| 5 | Objective of the Project | 6 |
| 6 | Tools and Technologies Used | 7 |
| 7 | Problem Statement | 8 |
| 8 | System Design and Architecture | 9 |
| 9 | Data Structures Used | 10 |
| 10 | Algorithm and Flowcharts | 11 |
| 11 | Implementation (Code Snippets & Explanation) | 12 |
| 12 | Sample Input and Output | 14 |
| 13 | Testing and Validation | 15 |
| 14 | Conclusion | 16 |
| 15 | Future Enhancements | 17 |
| 16 | References | 18 |

**Abstract**

This project focuses on implementing a maze solver application using a set of classical pathfinding algorithms: Depth First Search (DFS), Breadth First Search (BFS), Dijkstra's Algorithm, and A\* Search Algorithm. The solver is written in C++ and is designed to load a maze from a specially formatted input file (".in" extension). Each algorithm traverses the maze to discover a valid path from a defined start point to the goal. The implementation also logs key metrics such as the number of steps taken and time consumed. A comparative analysis is conducted to demonstrate how the choice of algorithm impacts performance and efficiency based on maze structure, making it a practical and educational study in algorithm design.

**Objective of the Project**

The primary objective of this mini-project is to design and develop a C++-based program that reads a maze configuration from an external file and solves it using four prominent search algorithms—DFS, BFS, Dijkstra’s, and A\*. The project aims to explore each algorithm’s effectiveness and trade-offs by measuring their runtime, memory usage, and path length. Through this, we hope to build a deeper understanding of pathfinding strategies, analyze their suitability in different maze configurations, and provide insights into optimization methods for future applications such as robotics, game AI, and navigation systems.

**Tools and Technologies Used**

* **Programming Language:** C++
* **Development Environment:** Visual Studio Code
* **Operating System:** Windows or Linux
* **Input Handling:** Text-based maze loaded from “.in” file
* **Data Structures:** Graph representations, 2D arrays/vectors, stack, queue, priority queue, and map/struct for tracking distances and paths
* **Libraries Used:** Standard Template Library (STL) for data management and traversal operations

**Problem Statement**

To develop a console-based C++ application capable of solving a maze provided as a text-based input file using four pathfinding algorithms: Depth First Search, Breadth First Search, Dijkstra’s Algorithm, and A\* Search Algorithm. The software should not only compute the valid paths from the starting point ‘S’ to the endpoint ‘E’ but also provide performance statistics such as time taken and number of steps, enabling a comparative evaluation. The program should handle various maze configurations with different levels of complexity and density of obstacles.

**System Design and Architecture**

* **Input Module:** Handles parsing of the maze from the “.in” file and stores it in a 2D matrix format.
* **Maze Representation:** Maze is modeled as a grid where walls, paths, start and end points are identified using symbols like ‘S’, ‘E’, and ‘#’.
* **Algorithm Module:** Encapsulates implementations of DFS, BFS, Dijkstra’s, and A\* search algorithms. Each algorithm processes the maze and records its route.
* **Output and Logging Module:** Displays the selected path, visualizes the traversal in the console, and logs metrics like steps and execution time.
* **Modularity:** Code is designed in modules for ease of testing and scalability.

**Data Structures Used**

* **2D Array/Vector:** To store maze grid
* **Stack:** Utilized in DFS for backtracking
* **Queue:** Used in BFS for level-order traversal
* **Priority Queue (Min-Heap):** Essential for Dijkstra’s and A\* for efficiently retrieving the next optimal node
* **Map/Struct:** Maintains parent nodes and distances from the start node to reconstruct paths after search completion
* **Pair Structures:** For representing coordinates and directions

**Algorithm and Flowcharts**

Each algorithm is based on well-known graph traversal principles:

* **Depth First Search (DFS):** Uses recursion or an explicit stack to go as deep as possible before backtracking. Often finds a solution quickly, but not guaranteed to be shortest.
* **Breadth First Search (BFS):** Uses a queue to explore level-by-level. Guarantees the shortest path in an unweighted maze.
* **Dijkstra’s Algorithm:** Uses a priority queue to find the least-cost path based on distance from the source. Suitable for weighted mazes.
* **A\* Search Algorithm***:* Enhances Dijkstra’s with heuristics (like Manhattan Distance) to prioritize nodes likely to reach the goal faster.

Flowcharts illustrating the logic and decision-making steps of each algorithm are included in the appendix.

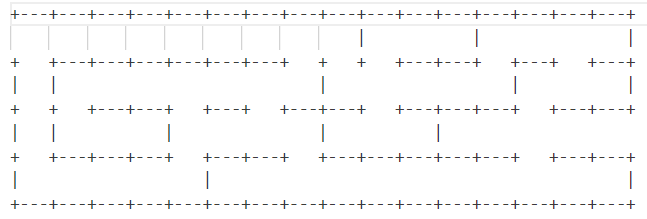
**Implementation (Code Snippets & Explanation)**

One of the four pre-defined path finding algorithms is Breadth First Search algorithm.

BFS Code Snippet

Currently, there are four pre-made mazes in the program, one of which is mentioned below.

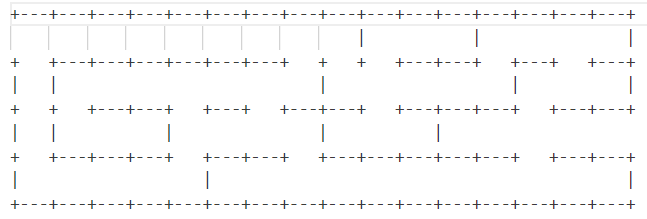
Maze Example (One.in)



Each algorithm is wrapped in its own function. Proper comments and debugging information are included in the full source code to enhance understanding and traceability.

**Sample Input and Output**

**Input**

Maze-1 (one.in)****

**Output:**

* **DFS Path:** Returns a valid path from start to end, but may not be shortest.
* **BFS Path:** Returns the shortest path in terms of steps.
* **Dijkstra’s Path:** Returns the lowest cost path using uniform cost search.
* **A\* Path:** Returns the most optimal path using heuristics.

Output includes a step-by-step traversal log, visual path representation, and a table summarizing algorithm performance.

**Testing and Validation**

The maze solver was thoroughly tested against different maze configurations: small mazes, large grids, complex multi-path layouts, and mazes with no possible solution. Each test case was evaluated for correctness, time taken to find the solution, and number of steps used. Edge cases such as completely blocked mazes and mazes with multiple start or end points were also tested. The robustness and stability of each algorithm under various inputs were analyzed.

**Conclusion**

This project demonstrates that while all four algorithms can solve the maze successfully, their performance varies significantly. DFS is simple and fast but does not guarantee the shortest path. BFS provides the shortest route in unweighted scenarios but can be slow for large mazes. Dijkstra’s is reliable for weighted mazes but slower due to exhaustive path evaluation. A\* outperforms others in terms of efficiency and path optimality when a good heuristic is used. Overall, this maze solver serves as an excellent educational tool for understanding and visualizing graph-based search algorithms.

**Future Enhancements**

* Development of a graphical user interface (GUI) to visualize maze solving in real time
* Dynamic maze generator for generating random mazes of adjustable difficulty
* Integration of additional algorithms like Greedy Best-First Search
* Extension to 3D mazes or real-world mapping systems
* Parallel implementation for simultaneous algorithm execution and performance comparison

**References**

* Cormen, Leiserson, Rivest, Stein – *Introduction to Algorithms*
* GeeksforGeeks.org – Algorithm explanations and examples
* StackOverflow – Developer discussions and solutions
* Wikipedia – Articles on DFS, BFS, Dijkstra’s and A\* algorithms
* Lecture notes and tutorials from academic courses on graph theory and algorithm design