DEPARTMENTOF

COMPUTERSCIENCE&ENGINEERING

Discover. Learn. Empower.

CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

Project Report:

Maze Solver using A* Algorithm

Subject Name – Design and Analysis of Algorithms

Subject Code – 23CSH-301

Submitted To:	Submitted By:
Er. Mohammad Shaqlain (E17211)	Name: Sanampreet Singh UID: 23BCS13053 Section: KRG_2-B

An implementation of the Randomized Depth-First Search (DFS) algorithm to generate a maze, and the A algorithm to find the shortest path.*

1. Introduction / Constraints

The program generates a random 10 X 10 maze using the Depth-First Search (DFS) algorithm and subsequently finds the shortest path from the Start cell (top-left, index 0) to the Goal cell (bottom-right, index 99) using the A* Search Algorithm.

Constraints:

- **Grid:** N X N grid (where N=10).
- **Movement:** Only orthogonal movement is allowed (North, East, South, West).
- **Solution:** The path must exist only through broken walls (paths) and must be the shortest possible route between the start and goal.

2. Input / Apparatus Used

- **Programming Language:** C++
- **IDE Used:** [Code::Blocks / VS Code / Visual Studio]
- **Input:** 10 X 10 array representing the maze structure and walls (generated internally by DFS).

3. Procedure / Algorithm

Algorithm: A* Search Algorithm (Best-First Search)

The A* algorithm finds the shortest path by evaluating a cost function f(n) for every node n:

$$f(n) = g(n) + h(n)$$

Term	Description
g(n)	The actual cost (distance/steps) from the **Start** node to the current node n.
h(n)	The estimated cost (heuristic) from the current node n to the **Goal** node.
f(n)	The total estimated cost of the path through node n. The algorithm prioritizes the node with the lowest f(n) value.

Heuristic Used: The program uses the **Manhattan Distance** as the heuristic h(n), which calculates the sum of the absolute differences in the X and Y coordinates between the current cell and the goal cell. This is an admissible heuristic, guaranteeing the shortest path is found.

Procedure Steps:

1. Initialize the Start cell's g=0 and calculate its initial f score.
2. Set the current cell to the Start cell and add it to the Closed List (or mark as visited in mazeBlocks[n][3]).
3. Identify all valid, unvisited neighbors of the current cell.
4. For each neighbor, calculate its new g (current g+1), h, and f scores.
5. If the new **f** score is lower than the previously recorded f score, update the cell's scores and set the current cell as its Parent (mazeBlocks[n][4]).
6. Select the unvisited node with the lowest f score (breaking ties with the lowest h score) and make it the new current cell.
7. Repeat steps 3-6 until the Goal is reached.
8. Reconstruct the shortest path by backtracking from the Goal to the Start using the recorded Parent pointers.

4. C++ Program:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <vector>
#include <numeric>
#include <string>
#include <algorithm>

const int size = 10;

bool visited[size * size];
```

```
bool walls[size * size][2] = { 0 };

const int start = 0;
const int goal = size * size - 1;

int path[size * size] = { 0 };

int mazeBlocks[size * size][5];

int pos = start;
int rot = 0;
int pathIndex = 0;

int emptyWalls[4] = { 0 };
int emptyWallsCount = 0;

/**
 * @brief Finds the delta to an unvisited neighbor for DFS maze generation.
 * @param currentPos The index of the current cell.
 * @param size The size of the maze grid (side length).
 * @return The delta (+/- 1 or +/- size) to the next cell, or 0 if no unvisited neighbors exist.
 */
int nextCell(int currentPos, int size) {
   std::vector<int> options;

   if (currentPos >= size && !visited[currentPos - size]) { options.push_back(-size); }
   if ((currentPos + 1) % size != 0 && !visited[currentPos + 1]) { options.push_back(1); }
   if (currentPos < size * (size - 1) && !visited[currentPos + size]) { options.push_back(size); }
   if (currentPos % size != 0 && !visited[currentPos - 1]) { options.push_back(-1); }

   if (options.empty()) { return 0; }

   return options[rand() % options.size()];
}

/**
```

```
 * @brief Breaks the wall between two adjacent cells.
 * @param pos1 Index of the first cell.
 * @param pos2 Index of the second cell.
 */
void connect(int pos1, int pos2) {
   if (pos2 > pos1) {
      if (pos2 == pos1 + 1) {
         walls[pos2][0] = 1;
      } else {
         walls[pos2][1] = 1;
      }
   } else {
      if (pos1 == pos2 + 1) {
         walls[pos1][0] = 1;
      } else {
         walls[pos1][1] = 1;
      }
   }
}

/**
 * @brief Generates the maze using Randomized Depth-First Search (DFS).
 * @param currentPos The index of the current cell.
 * @param size The size of the maze grid.
 */
void randomdfs(int currentPos, int size) {
   visited[currentPos] = true;
   int nextDelta;

   while ((nextDelta = nextCell(currentPos, size)) != 0) {
      int nextPos = currentPos + nextDelta;
      connect(currentPos, nextPos);
      randomdfs(nextPos, size);
   }
}
```

```
/**
 * @brief Initializes and runs the DFS maze generation.
 */
void createMazeDFS() {
   std::cout << "Starting DFS maze generation..." << std::endl;
   for (int i = 0; i < size * size; ++i) {
      visited[i] = false;
      walls[i][0] = 0;
      walls[i][1] = 0;
      path[i] = 0;
   }

   randomdfs(start, size);
   std::cout << "Maze generation complete." << std::endl;
}

/**
 * @brief Finds open walls from the current position. Used for A*.
 */
void getEmptyWalls() {
   emptyWallsCount = 0;

   if (pos >= size && walls[pos][1]) {
      emptyWalls[emptyWallsCount++] = -size;
   }
   if ((pos + 1) % size != 0 && walls[pos + 1][0]) {
      emptyWalls[emptyWallsCount++] = 1;
   }
   if (pos < size * (size - 1) && walls[pos + size][1]) {
      emptyWalls[emptyWallsCount++] = size;
   }
   if (pos % size != 0 && walls[pos][0]) {
      emptyWalls[emptyWallsCount++] = -1;
   }
}
```

```
/**
 * @brief Updates G, H, F, and Parent values for a neighbor in A* search.
 */
void aStarCalculate(int neighborPos) {
   int g = mazeBlocks[pos][0] + 1;

   int dx = std::abs(neighborPos % size - goal % size);
   int dy = std::abs((neighborPos / size) - (goal / size));
   int h = dx + dy;

   int f = g + h;

   if (mazeBlocks[neighborPos][2] == 0 || f < mazeBlocks[neighborPos][2]) {
      mazeBlocks[neighborPos][0] = g;
      mazeBlocks[neighborPos][1] = h;
      mazeBlocks[neighborPos][2] = f;
      mazeBlocks[neighborPos][4] = pos;
   }
}

/**
 * @brief Solves the maze using the A* search algorithm (guaranteed shortest path).
 * @return The length of the shortest path found.
 */
int solveMazeAstar() {
   for (int i = 0; i < size * size; i++) {
      for (int j = 0; j < 5; j++) {
         mazeBlocks[i][j] = 0;
      }
      path[i] = 0;
   }

   pos = start;
   mazeBlocks[pos][0] = 0;
   mazeBlocks[pos][1] = std::abs(pos % size - goal % size) + std::abs((pos / size) - (goal /
size));
```

```
mazeBlocks[pos][2] = mazeBlocks[pos][1];

while (pos != goal) {
   mazeBlocks[pos][3] = 1;

   getEmptyWalls();

   for (int i = 0; i < emptyWallsCount; i++) {
      int neighborPos = pos + emptyWalls[i];
      if (mazeBlocks[neighborPos][3] == 0) {
         aStarCalculate(neighborPos);
      }
   }
   int minf = size * size * 2;
   int minh = size * size;
   int nextCell = -1;

   for (int i = 0; i < size * size; i++) {
      if (mazeBlocks[i][3] == 0 && mazeBlocks[i][2] != 0) {
         if (mazeBlocks[i][2] < minf) {
            minf = mazeBlocks[i][2];
            minh = mazeBlocks[i][1];
            nextCell = i;
         } else if (mazeBlocks[i][2] == minf && mazeBlocks[i][1] < minh) {
            minh = mazeBlocks[i][1];
            nextCell = i;
         }
      }
   }

   if (nextCell == -1) {
      return -1;
   }

   pos = nextCell;
}
```

```
    int currentCell = goal;
    std::vector<int> shortestPathCells;

    while (currentCell != -1) {
        shortestPathCells.push_back(currentCell);
        if (currentCell == start) break;
        currentCell = mazeBlocks[currentCell][4];
    }
    int pathLength = shortestPathCells.size() - 1;
    for (size_t i = 0; i < shortestPathCells.size(); ++i) {
        int cellIndex = shortestPathCells[shortestPathCells.size() - 1 - i];
        path[cellIndex] = i + 1;
    }
    return pathLength;
}

/**
 * @brief Prints the maze representation with the corresponding solution path markers.
 * @param title The title for the maze output.
 */
void printMaze(const std::string& title) {
    std::cout << "\n================================================" <<
std::endl;
    std::cout << "--- " << title << " (" << size << "x" << size << ") ---" << std::endl;
    std::cout << "================================================" << std::endl;

    bool is_unsolved = title.find("Unsolved") != std::string::npos;
    bool is_astar_solved = title.find("A* Shortest Path") != std::string::npos;

    for (int j = 0; j < size; ++j) {
        for (int i = 0; i < size; ++i) {
            int currentPos = j * size + i;

            std::cout << "+";

            bool wallOpen = false;
```

```
    if (j > 0) {
        wallOpen = walls[currentPos][1] == 1;
    } else {
        if (currentPos == start) {
            wallOpen = true;
        }
    }
    if (wallOpen) {
        std::cout << "   ";
    } else {
        std::cout << "---";
    }
}
std::cout << "+\n";

for (int i = 0; i < size; ++i) {
    int currentPos = j * size + i;

    bool wallOpen = false;
    if (i == 0) {
        if (currentPos == start) {
            wallOpen = true;
        } else {
            wallOpen = false;
        }
    } else {
        wallOpen = walls[currentPos][0] == 1;
    }
    if (wallOpen) {
        std::cout << " ";
    } else {
        std::cout << "|";
    }

    if (is_unsolved) {
        if (currentPos == start) std::cout << " S ";
```

```
            else if (currentPos == goal) std::cout << " G ";
            else std::cout << "   ";
        }
        else if (is_astar_solved && path[currentPos] > 0) {
            int step = path[currentPos];
            std::string s = std::to_string(step);

            if (s.length() == 1) std::cout << "  " << s;
            else if (s.length() == 2) std::cout << " " << s;
            else if (s.length() >= 3) std::cout << s.substr(0, 3);
        }
        else {
            std::cout << "   ";
        }
    }

    if (j == size - 1) {
        if (goal % size == size - 1) {
            bool pathExists = (is_astar_solved && path[goal] > 0);

            if (pathExists || is_unsolved) {
                std::cout << " \n";
            } else {
                std::cout << "|\n";
            }
        } else {
            std::cout << "|\n";
        }
    } else {
        std::cout << "|\n";
    }
}
for (int i = 0; i < size; ++i) {
    std::cout << "+";
    if (i == size - 1) {
        bool pathExists = (is_astar_solved && path[goal] > 0);

        if (pathExists || is_unsolved) {
            std::cout << " ";
        } else {
```

```
                std::cout << "---";
            }
        } else {
            std::cout << "---";
        }
    }
    std::cout << "+\n";

    if (is_unsolved) {
        std::cout << "Legend: S=Start, G=Goal\n";
    } else if (is_astar_solved) {
        std::cout << "Legend: 1,2,3...=A* Shortest Path Steps\n";
    }
}

int main(){
    srand(time(NULL));
    createMazeDFS();

    for (int i = 0; i < size * size; i++)
        path[i] = 0;

    printMaze("Unsolved Maze");

    std::cout << "\nStarting A* search solution..." << std::endl;
    int aStarLength = solveMazeAstar();

    if (aStarLength != -1) {
        std::cout << "A* solution complete. Shortest path length: " << aStarLength << " steps." <<
std::endl;
        printMaze("A* Shortest Path");
    } else {
        std::cout << "A* FAILED: No path found." << std::endl;
    }

    std::cout << "\nExecution finished. Maze size reduced to " << size << "x" << size << " for console
visualization." << std::endl;

    return 0;
}
```

5. Sample Output:

```
PS C:\Users\HP\Desktop\DAA project> cd "c:\Users\HP\Desktop\DAA project\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
Starting DFS maze generation...
Maze generation complete.

================================================
--- Unsolved Maze (10x10) ---
================================================
+   +---+---+---+---+---+---+---+---+---+---+
  S     |           |                       |
+---+   +---+   +   +---+---+   +   +   +   +
|   |       |   |           |   |   |   |   |
+   +---+   +   +---+---+   +---+   +   +   +
|       |   |   |               |   |   |   |
+   +   +   +   +---+---+---+   +   +   +   +
|   |   |   |           |   |       |   |   |
+   +   +   +   +---+   +   +---+---+   +   +
|   |   |   |       |   |               |   |
+   +   +   +---+---+   +---+   +---+---+   +
|   |   |           |       |           |   |
+   +---+---+---+   +   +   +---+---+   +   +
|           |   |   |   |           |   |   |
+   +   +   +   +   +   +---+---+   +   +   +
|   |   |       |   |           |   |   |   |
+---+   +   +---+   +   +---+---+   +   +   +
|       |   |       |   |       |   |   |   |
+   +---+---+   +---+---+   +   +   +   +   +
|                           |       | G
+---+---+---+---+---+---+---+---+---+---+ +
Legend: S=Start, G=Goal

Starting A* search solution...
A* solution complete. Shortest path length: 60 steps.

================================================
--- A* Shortest Path (10x10) ---
================================================
+   +---+---+---+---+---+---+---+---+---+---+
    1   2|     36  37|             47  48|
+---+   +---+   +   +---+---+   +   +   +   +
|   |  3   4| 35| 38  39  40|   | 46| 49|
+   +---+   +   +---+---+   +---+   +   +   +
|       |  5| 34|         41  42| 45| 50|
+   +   +   +   +---+---+---+   +   +   +   +
|   |   |   6| 33  32  31|   | 43  44| 51|
+   +   +   +   +---+   +   +---+---+   +   +
|   |   |   7|       | 30|     54  53  52|
+   +   +   +---+---+   +---+   +---+---+---+
|   |   |   8   9  10| 29  28| 55  56  57|
+   +---+---+---+   +   +   +---+---+   +   +
|           |   | 11|   | 27  26  25| 58|
+   +   +   +   +   +   +---+---+   +   +   +
|   |   |       | 12|           | 24| 59|
+---+   +   +---+   +   +---+---+   +   +   +
|       |   | 14  13|   | 19  20| 23| 60|
+   +---+---+   +---+---+   +   +   +   +   +
|             15  16  17  18| 21  22| 61
+---+---+---+---+---+---+---+---+---+---+ +
Legend: 1,2,3...=A* Shortest Path Steps

Execution finished. Maze size reduced to 10x10 for console visualization.
PS C:\Users\HP\Desktop\DAA project>
```

DEPARTMENTOF COMPUTERSCIENCE&ENGINEERING

```
cd "c:\Users\HP\Desktop\DAA project\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
Starting DFS maze generation...
Maze generation complete.

================================================
--- Unsolved Maze (10x10) ---
================================================
+   +---+---+---+---+---+---+---+---+---+
  S |           |                       |
+   +   +---+---+---+   +   +---+---+---+
|   |       |         |                 |
+   +---+   +   +---+---+---+---+---+   +
|       |       |                   |   |
+---+   +---+   +---+   +   +---+   +   +
|   |       |       |   |   |       |   |
+   +---+   +---+   +---+   +   +---+   +
|       |   |               |   |   |   |
+   +   +   +---+---+---+---+   +   +   +
|   |   |           |               |   |
+   +   +---+---+   +   +---+---+---+   +
|   |       |   |   |   |           |   |
+---+   +   +   +   +---+   +---+   +   +
|       |       |           |   |   |   |
+   +---+---+---+---+---+---+   +   +   +
|           |                   |   |   |
+   +---+   +   +---+---+   +---+   +   +
|       |           |             G
+---+---+---+---+---+---+---+---+---+---+ +
Legend: S=Start, G=Goal

Starting A* search solution...
A* solution complete. Shortest path length: 20 steps.

================================================
--- A* Shortest Path (10x10) ---
================================================
+   +---+---+---+---+---+---+---+---+---+
   1|           |                       |
+   +   +---+---+---+   +   +---+---+---+
|  2|       |         |                 |
+   +---+   +   +---+---+---+---+---+   +
|  3   4|       |                   |   |
+---+   +---+   +---+   +   +---+   +   +
|   |  5   6|       |   |   |       |   |
+   +---+   +---+   +---+   +   +---+   +
|       |  7|               |   |   |   |
+   +   +   +---+---+---+---+   +   +   +
|   |   |  8   9  10|               |   |
+   +   +---+---+   +   +---+---+---+   +
|   |       |   | 11|   | 15  16  17|   |
+---+   +   +   +   +---+   +---+   +   +
|       |       | 12  13  14|   | 18|   |
+   +---+---+---+---+---+---+---+   +   +
|           |                   | 19|   |
+   +---+   +   +---+---+   +---+   +   +
|       |           |           20  21
+---+---+---+---+---+---+---+---+---+---+ +
Legend: 1,2,3...=A* Shortest Path Steps
```

4. Complexity Analysis:

Let N be the side length of the grid (10), and **V** be the total number of cells, $V = N^2$ (100).

- **Worst Case Time:** $O(V^2)$ or $O((N^2)^2)$
    - The current implementation finds the next node with the minimum F score by iterating linearly through all V cells in every step. In the worst case, this leads to V iterations, repeated up to V times.
- **Space Complexity:** $O(V)$ or $O(N^2)$
    - Space is dominated by the storage for the maze grid (walls), the visited flags (visited), and the A* data (mazeBlocks), all proportional to the total number of cells V.

5. Result

The program successfully **generates a random maze and solves** the puzzle using the A* Shortest Path Algorithm. The resulting path is guaranteed to be optimal (minimal number of steps) and is clearly visualized using sequential step numbers from 1 (Start) to L (Goal).

6. Conclusion

1. **Project Success and Algorithm Validation:** The implementation successfully combined the **Randomized DFS** for robust maze generation and the **A* Search Algorithm** to reliably find the path, demonstrating proficiency in graph traversal and informed search techniques.
2. **Optimality through Heuristic:** The use of the **admissible Manhattan Distance heuristic** ensures that the A* algorithm is optimally guided and guarantees the discovery of the absolute shortest path, validating the core principle of the search strategy.

3. **Complexity Bottleneck:** The current A* implementation exhibits a worst-case time complexity of $O(V^2)$. This is primarily due to the inefficient linear search used to select the next node with the minimum f(n) score in each iteration.
4. **Future Optimization:** To enhance performance for larger maze sizes, future work should focus on integrating a **Priority Queue** data structure. This optimization is expected to improve the worst-case time complexity to a more efficient O(VlogV).