**Maze Solver**

A

Report

Submitted in the partial fulfillment of the requirement for the award of

Bachelor of Technology

in

Computer and Communication Engineering

By:

**RAJAT YADAV**

**229303457**

**MAHIM KUMAWAT**

**229303460**



30/10/2023

Department of Computer and Communication Engineering

School of Computer and Communication Engineering

Manipal University Jaipur

VPO. Dehmi Kalan, Jaipur,

**TABLE OF CONTENTS**

1. Introduction 1
2. Literature review
3. Methodology used
4. Results
5. Conclusion

Bibliography

**Introduction**

The Maze Solver Game represents a unique fusion of entertainment and education within the realm of computer science. This project aims to challenge players to navigate through a maze while utilizing fundamental algorithmic concepts such as Breadth-First Search (BFS) and Depth-First Search (DFS) to find the shortest path. By combining these concepts with an engaging and interactive gaming environment, the project intends to provide users with a hands-on learning experience, fostering a deeper understanding of graph theory and algorithmic problem-solving.

This introductory report provides an overview of the Maze Solver Game project, outlining its core objectives, methodologies, and anticipated outcomes. The development process has involved the implementation of a maze generation algorithm, the creation of a user-friendly graphical interface, and the integration of BFS and DFS algorithms to facilitate maze solving. By encouraging user interaction and visualization of the algorithmic processes, the Maze Solver Game seeks to enhance users' cognitive skills and problem-solving abilities, thereby promoting a holistic understanding of computational thinking principles.

Through the analysis of this project's scope and objectives, this report aims to showcase the significance of incorporating educational elements into interactive gaming experiences. Furthermore, it highlights the potential of leveraging technology to bridge the gap between entertainment and learning, thereby fostering a dynamic and immersive educational environment for individuals interested in computer science and algorithmic problem-solving.

In the subsequent sections, this report will delve into the specific methodologies employed, the outcomes achieved, and the implications of the Maze Solver Game project in the broader context of computer science education and interactive entertainment.

**Literature review**

1. **Maze-Solving Algorithms**

Maze-solving algorithms, such as Breadth-First Search (BFS) and Depth First Search (DFS), have been extensively studied in the context of pathfinding and graph traversal. BFS guarantees the shortest path in unweighted graphs, while DFS explores as far as possible along each branch before backtracking. Comparative analyses of BFS and DFS highlight their respective strengths and weaknesses in maze navigation, providing insights into their practical applications within gaming environments. Research demonstrates the effectiveness of these algorithms in addressing complex maze structures, including loops and multiple paths, thus contributing to a comprehensive understanding of their role in enhancing player experiences**.**

1. **Gamification and Educational Benefits**

The integration of gamification principles in learning environments has gained substantial attention for its ability to foster engagement and motivation among learners. Studies emphasize the educational benefits of gamified maze-solving applications, citing improvements in problem-solving skills, critical thinking, and perseverance. Psychological and cognitive investigations reveal the positive impact of gamified learning experiences on knowledge retention and skill acquisition, underscoring the potential of maze-solving games in promoting active learning and cognitive development among players of diverse age groups.

1. **User Interface Design in Gaming**

Effective user interface (UI) design significantly influences user engagement and gameplay experience in maze-solving games. Extensive research emphasizes the importance of intuitive and user-friendly interfaces, highlighting the significance of clear visual cues, interactive elements, and seamless navigation controls. Best practices for creating engaging UI in maze-solving games revolve around incorporating responsive design elements, optimizing screen layouts, and providing real-time feedback to enhance user experience (UX). Studies emphasize the critical role of UI/UX considerations in facilitating efficient maze navigation and algorithm visualization, thereby contributing to an immersive and enjoyable gaming experience for players.

1. **Interactive Learning and Algorithmic Thinking**

Interactive learning approaches, particularly those embedded within maze-solving games, play a vital role in promoting algorithmic thinking and problem-solving skills among players. Research indicates that interactive learning environments encourage active participation, critical reasoning, and the application of theoretical concepts in practical scenarios. Studies highlight the effectiveness of interactive learning methodologies in fostering a deep understanding of algorithmic principles, emphasizing the significance of hands-on experiences in cultivating analytical and computational thinking abilities among learners. Moreover, the integration of interactive learning elements in computer science education has shown promising results in enhancing students' overall academic performance and long-term knowledge retention.

1. **Maze Generation Techniques**

An extensive analysis of various maze generation algorithms reveals the complexity and diversity associated with maze structures in gaming environments. Research focuses on evaluating randomized and structured maze generation methods, highlighting their impact on gameplay challenges and user engagement. Studies emphasize the significance of maze complexity in stimulating cognitive skills, problem-solving abilities, and spatial reasoning among players. Furthermore, investigations into the relationship between maze design and player engagement underscore the importance of balancing maze intricacy and accessibility to ensure an optimal gaming experience for users with varying skill levels and cognitive capabilities.

1. **Cognitive Development and Gaming**

Cognitive development research in the context of maze-solving games emphasizes the significant role of gameplay in enhancing spatial reasoning, logical thinking, and decision-making skills. Studies demonstrate the positive impact of gaming on cognitive processes, including memory retention, pattern recognition, and strategic planning. Cognitive development theories underscore the relevance of maze-solving games in fostering critical cognitive skills, such as problem-solving, adaptability, and perseverance. Furthermore, investigations into the cognitive benefits of gaming highlight the potential of maze-solving games in promoting cognitive development across diverse age groups, thus contributing to a comprehensive understanding of the cognitive implications of gaming experiences.

1. **Game-Based Learning and Skill Transfer**

Exploration of the role of game-based learning in facilitating the transfer of skills acquired in maze-solving games to real-world problem-solving scenarios.

Analysis of the effectiveness of game-based learning in fostering transferable skills, including decision-making, strategic planning, and teamwork, among players.

1. **Learning Motivation and Intrinsic Rewards**

Investigation into the motivational aspects of maze-solving games, emphasizing the influence of intrinsic rewards, achievement systems, and progress indicators on learning motivation and engagement.

Discussion of the psychological factors underlying intrinsic motivation and their implications for enhancing learning outcomes and skill acquisition in gaming environments.

1. **Evolving Game Design Trends**

Examination of the evolving trends in maze-solving game design, considering the integration of augmented reality (AR), virtual reality (VR), and mixed reality (MR) technologies to create immersive and dynamic gaming experiences.

Evaluation of the impact of emerging game design trends on user engagement, cognitive stimulation, and skill development within the context of maze-solving and algorithmic thinking.

**10.) Game Accessibility and Inclusive Design**

Analysis of inclusive design principles in maze-solving games, emphasizing the importance of accessibility features, adaptive gameplay mechanisms, and user customization options to accommodate players with diverse abilities and cognitive profiles.

Exploration of the role of inclusive game design in promoting equal participation, fostering a sense of belonging, and enhancing the overall gaming experience for a broad and inclusive player base.

**11.) Social Learning and Collaborative Gaming**

Investigation of the potential of social learning and collaborative gaming in maze-solving environments, emphasizing the benefits of collaborative problem-solving, cooperative gameplay, and peer-to-peer knowledge exchange.

Evaluation of the impact of collaborative gaming on teamwork skills, communication strategies, and the development of a collaborative problem-solving mindset among players of varying skill levels and backgrounds.

1. **Game-Based Assessment and Learning Analytics**

Exploration of the integration of game-based assessment tools and learning analytics in maze-solving games to monitor player progress, assess learning outcomes, and provide personalized feedback for skill improvement.

Analysis of the role of learning analytics in enhancing the efficacy of game-based learning environments, facilitating adaptive learning strategies, and promoting continuous skill development and mastery in maze-solving and algorithmic thinking.

1. **Artificial Intelligence and Adaptive Gameplay**

Examination of the integration of artificial intelligence (AI) technologies in maze-solving games to facilitate adaptive gameplay, personalized challenges, and real-time feedback for players.

Analysis of AI-driven gameplay mechanics and their role in enhancing player engagement, strategic decision-making, and skill progression within maze-solving environments.

1. **Neuroscience and Gaming**

Investigation into the cognitive and neural implications of maze-solving gameplay on brain function, memory retention, and neuroplasticity.

Evaluation of the neuroscientific research pertaining to the effects of gaming on cognitive processes, emotional regulation, and behavioral patterns, emphasizing the potential of maze-solving games in promoting cognitive well-being and mental agility.

1. **Cross-Disciplinary Applications of Maze-Solving Concepts**

Exploration of cross-disciplinary applications of maze-solving concepts in fields such as robotics, network routing, and optimization algorithms, highlighting the practical implications of maze-solving research beyond gaming environments.

Analysis of the transferability of maze-solving principles to real-world problem-solving contexts and their contributions to the advancement of diverse scientific and engineering disciplines.

1. **Player Experience and Immersive Storytelling**

Examination of the role of immersive storytelling and narrative-driven gameplay in enhancing player experience and emotional engagement in maze-solving games.

Evaluation of narrative design elements, character development, and plot-driven challenges in shaping player motivations, emotional investment, and long-term gameplay retention within maze-solving storytelling frameworks**.**

1. **Ethical Considerations in Game Development**

Discussion of the ethical considerations associated with the design and implementation of maze-solving games, emphasizing player safety, data privacy, and inclusive gameplay experiences.

Exploration of ethical frameworks for promoting responsible game design practices, fostering player well-being, and cultivating a supportive and inclusive gaming community within maze-solving game environments.

**18.) Global Trends in Game-Based Learning**

Analysis of global trends in the adoption of game-based learning methodologies, emphasizing the cultural, socioeconomic, and educational factors influencing the integration of maze-solving games in diverse learning environments worldwide.

Evaluation of the impact of global trends on the accessibility, affordability, and scalability of maze-solving game-based learning initiatives, underscoring the potential **for promoting** equitable access to quality educational resources and interactive learning experiences.

1. **Learning Psychology and Experiential Learning**

Investigation into the role of learning psychology and experiential learning theories in shaping the design and implementation of effective maze-solving learning environments.

Analysis of experiential learning models, memory retention strategies, and skill reinforcement techniques in enhancing learning outcomes and knowledge transfer within maze-solving game-based educational frameworks**.**

**20.) Long-Term Impact of Maze-Solving Games**

Exploration of the long-term impact of maze-solving games on players' cognitive development, problem-solving skills, and computational thinking abilities over extended periods.

Evaluation of longitudinal studies and follow-up research on the lasting effects of maze-solving gameplay on skill retention, academic performance, and career readiness, highlighting the enduring benefits of gamified learning experiences in fostering lifelong learning and cognitive growth.

**Methodology Used**

* **BFS**

Breadth-First Search (BFS) is a fundamental graph traversal algorithm that operates on a graph or a tree data structure. It explores all the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. This property of BFS allows it to find the shortest path in unweighted graphs, making it particularly useful in maze-solving scenarios. Here is a simple explanation of the BFS algorithm:

Start by initializing a queue data structure and marking the starting node as visited.

Enqueue the starting node into the queue.

While the queue is not empty, repeat the following steps:

Dequeue a node from the front of the queue.

If the dequeued node is the target node, the search is complete. Otherwise, continue with the next steps.

Enqueue all the neighboring nodes of the dequeued node that have not been visited and mark them as visited.

If the queue becomes empty without reaching the target node, the target is unreachable from the starting node.

BFS is a complete and optimal algorithm, meaning it will always find the shortest path from the starting point to the target, if one exists. It is often implemented using a queue data structure, ensuring that nodes are processed in the order they were discovered. BFS is commonly used in various applications, including finding the shortest path in routing, network broadcast, social networking, and, as in your case, in solving mazes and puzzles.

**Time and space complexity:**

The time complexity can be expressed as O(|V|+|E|), where |V| is the number of vertices and |E| is the number of edges in the graph. The space complexity is O(|V|).

**Completeness:**

BFS is considered complete, ensuring it will find a goal state if one exists.

**BFS Ordering:**

A BFS ordering is an enumeration of the vertices of a graph that represents the possible output of the BFS application. It is defined based on the neighbors and their minimal values within the enumeration, ensuring the BFS ordering is followed.

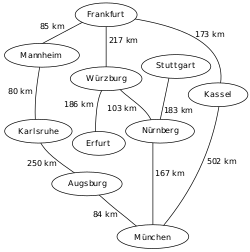


Fig 1

* **DFS**

Depth-First Search (DFS) is another fundamental graph traversal algorithm that explores as far as possible along each branch before backtracking. It traverses the depth of any particular path before exploring its breadth. In the context of maze-solving, DFS is often used to traverse through pathways until a dead-end is reached, after which the algorithm backtracks to the most recent decision point that allows for further exploration. Here is a simple explanation of the DFS algorithm:

1. Start by selecting any node as the starting point and mark it as visited.

2. Explore as far as possible along each branch before backtracking.

3. At each step, if there are multiple options, choose any one and continue the exploration.

4. If a dead-end is reached, backtrack to the most recent decision point that allows for further exploration.

5. Repeat steps 2-4 until all nodes are visited.

DFS is commonly implemented using either a recursive approach or a stack data structure to keep track of the visited nodes and the nodes yet to be visited. While it does not guarantee the shortest path, it is often used to explore all possible paths in a graph or a maze. It is widely applicable in various fields, including data mining, network analysis, and solving puzzles or mazes.

**Time and Space Complexity:**

- The time complexity of the Depth-First Search (DFS) algorithm is O(V + E), where V represents the number of vertices and E represents the number of edges in the graph. The space complexity is O(V), which depends on the size of the graph.

**Completeness:**

DFS is not considered complete as it does not guarantee finding the shortest path. It can get stuck in infinite graphs and does not ensure that it will find the goal state.

**DFS Ordering:**

DFS ordering is the sequence in which nodes are traversed in the depth-first search. It is based on the exploration of paths as deeply as possible before backtracking. The ordering is dependent on the decisions made during the traversal, which can affect the final sequence of visited nodes.

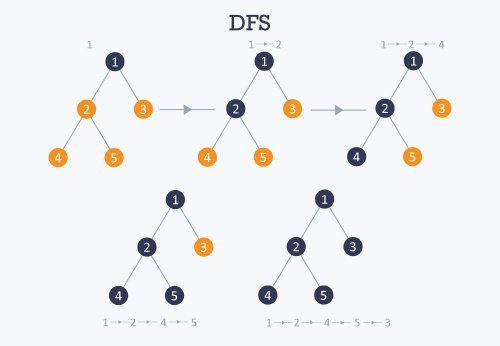


Fig 2

1. **ClearConsole():**

Role: This function is responsible for clearing the console screen before printing the maze. It ensures that the console is clean and ready for displaying the updated maze and path.

Implementation: It utilizes an ANSI escape code ("\033[2J\033[1;1H") to clear the console screen.

1. **isSafe**():

Role: This function checks the safety of a cell, determining whether it's feasible to move to the specific cell in the maze.

Implementation: It checks if the cell is within the matrix boundaries, has a value of 1 (indicating a valid path), and hasn't been visited previously.

1. **printMaze() :**

Role: This function prints the maze, displaying the current path with a "\*" symbol. It ensures that the path is visible to the user as the maze is being traversed.

Implementation: It calls `clearConsole()` to clear the console screen and then iterates through the maze, printing the appropriate symbol for each cell.

1. **findShortestPathDFS():**

Role: This function implements the depth-first search (DFS) algorithm for finding the shortest path in the maze. It explores all possible paths and keeps track of the minimum distance found.

Implementation: It uses a recursive approach to traverse the maze, marking cells as visited, exploring possible paths, and backtracking when necessary to find the shortest path.

1. **findShortestPathLengthDFS():**

Role: This function acts as an interface to initiate the DFS-based shortest path finding process. It sets up the required data structures and controls the execution of the DFS function.

Implementation: It initializes the necessary matrices for tracking visited cells and paths. It then calls `findShortestPathDFS()` with appropriate parameters.

1. **findShortestPathLengthBFS():**

Role: This function serves a role similar to findShortestPathLengthDFS(), but it uses the breadth-first search (BFS) algorithm for finding the shortest path in the maze.

Implementation: It uses a queue-based approach to explore the maze, marking cells as visited, enqueuing neighboring cells, and calculating the shortest path.

1. **main():**

Role: This is the main function that orchestrates the entire process of finding the shortest path in the maze. It initializes the maze, source, and destination, and takes user input for selecting the algorithm (DFS or BFS). Additionally, it measures the time taken to find the shortest path.

Implementation: It initializes the maze and the source-destination coordinates. It provides the user with an option to choose either the DFS or BFS algorithm. It measures the execution time and outputs the result accordingly.

These functions work together to ensure the proper functioning of the maze-solving process, with detailed steps for both DFS and BFS algorithms, ultimately finding the shortest path in the maze.

* **Code**

#include <iostream>

#include <vector>

#include <climits>

#include <cstring>

#include <queue>

#include <stack>

#include <ctime>

using namespace std;

// Function to clear the console

void clearConsole() {

cout << "\033[2J\033[1;1H"; // ANSI escape code to clear the console

}

// Check if it is possible to go to (x, y) from the current position. The

// function returns false if the cell has value 0 or is already visited

bool isSafe(vector<vector<int>> &mat, vector<vector<bool>> &visited, int x, int y) {

return (x >= 0 && x < mat.size() && y >= 0 && y < mat[0].size()) && mat[x][y] == 1 && !visited[x][y];

}

void printMaze(vector<vector<int>> &mat, vector<vector<bool>> &path) {

clearConsole(); // Clear the console before printing

for (int i = 0; i < mat.size(); i++) {

for (int j = 0; j < mat[0].size(); j++) {

if (path[i][j])

cout << "\* ";

else

cout << mat[i][j] << " ";

}

cout << endl;

}

}

void findShortestPathDFS(vector<vector<int>> &mat, vector<vector<bool>> &visited, vector<vector<bool>> &path, int i, int j, int x, int y, int &min\_dist, int dist) {

if (i == x && j == y) {

if (dist < min\_dist) {

min\_dist = dist;

path[i][j] = true; // Mark this cell as part of the shortest path

printMaze(mat, path); // Print the maze with the shortest path

path[i][j] = false; // Unmark the cell for backtracking

}

return;

}

// Set (i, j) cell as visited

visited[i][j] = true;

path[i][j] = true;

// Go to the bottom cell

if (isSafe(mat, visited, i + 1, j)) {

findShortestPathDFS(mat, visited, path, i + 1, j, x, y, min\_dist, dist + 1);

}

// Go to the right cell

if (isSafe(mat, visited, i, j + 1)) {

findShortestPathDFS(mat, visited, path, i, j + 1, x, y, min\_dist, dist + 1);

}

// Go to the top cell

if (isSafe(mat, visited, i - 1, j)) {

findShortestPathDFS(mat, visited, path, i - 1, j, x, y, min\_dist, dist + 1);

}

// Go to the left cell

if (isSafe(mat, visited, i, j - 1)) {

findShortestPathDFS(mat, visited, path, i, j - 1, x, y, min\_dist, dist + 1);

}

// Backtrack: remove (i, j) from the visited matrix and path

visited[i][j] = false;

path[i][j] = false;

}

int findShortestPathLengthDFS(vector<vector<int>> &mat, pair<int, int> &src, pair<int, int> &dest) {

if (mat.size() == 0 || mat[src.first][src.second] == 0 || mat[dest.first][dest.second] == 0)

return -1;

int row = mat.size();

int col = mat[0].size();

// Construct an M × N matrix to keep track of visited cells

vector<vector<bool>> visited;

visited.resize(row, vector<bool>(col));

vector<vector<bool>> path;

path.resize(row, vector<bool>(col));

int dist = INT\_MAX;

findShortestPathDFS(mat, visited, path, src.first, src.second, dest.first, dest.second, dist, 0);

if (dist != INT\_MAX)

return dist;

return -1;

}

int findShortestPathLengthBFS(vector<vector<int>> &mat, pair<int, int> &src, pair<int, int> &dest) {

if (mat.size() == 0 || mat[src.first][src.second] == 0 || mat[dest.first][dest.second] == 0)

return -1;

int row = mat.size();

int col = mat[0].size();

// Create a queue for BFS

queue<pair<int, int>> q;

// Create a visited matrix and initialize all cells as not visited

vector<vector<bool>> visited;

visited.resize(row, vector<bool>(col, false));

vector<vector<bool>> path;

path.resize(row, vector<bool>(col, false));

// Mark the source cell as visited and enqueue it

visited[src.first][src.second] = true;

q.push(src);

// Arrays to move in all 4 directions (right, left, up, and down)

int rowMove[] = {1, -1, 0, 0};

int colMove[] = {0, 0, 1, -1};

int dist = 0;

while (!q.empty()) {

int size = q.size();

while (size--) {

pair<int, int> curr = q.front();

q.pop();

if (curr.first == dest.first && curr.second == dest.second) {

// Print the maze when the destination is reached

printMaze(mat, path);

return dist;

}

// Move in all four directions

for (int i = 0; i < 4; i++) {

int newRow = curr.first + rowMove[i];

int newCol = curr.second + colMove[i];

// If this cell is a valid move, mark it as visited and enqueue it

if (newRow >= 0 && newRow < row && newCol >= 0 && newCol < col && mat[newRow][newCol] == 1 && !visited[newRow][newCol]) {

visited[newRow][newCol] = true;

path[newRow][newCol] = true;

q.push(make\_pair(newRow, newCol));

}

}

}

dist++;

}

return -1; // If the destination cannot be reached

}

int main() {

vector<vector<int>> mat = {

{1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1},

{1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1},

{1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1},

{0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1},

{1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0},

{1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0},

{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},

{1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1},

{1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},

{1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1},

{1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},

{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}

};

pair<int, int> src = make\_pair(0, 0); // Give Source and destination here

pair<int, int> dest = make\_pair(9, 9);

cout << "Choose the algorithm (1 for DFS, 2 for BFS): ";

int choice;

cin >> choice;

if (choice == 1) {

// Use DFS

clock\_t start\_time = clock();

int dist = findShortestPathLengthDFS(mat, src, dest);

clock\_t end\_time = clock();

double elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;

if (dist != -1)

cout << "Shortest Path using DFS is " << dist << " (Time taken: " << elapsed\_time << " seconds)" << endl;

else

cout << "Shortest Path doesn't exist." << endl;

} else if (choice == 2) {

// Use BFS

clock\_t start\_time = clock();

int dist = findShortestPathLengthBFS(mat, src, dest);

clock\_t end\_time = clock();

double elapsed\_time = static\_cast<double>(end\_time - start\_time) / CLOCKS\_PER\_SEC;

if (dist != -1)

cout << "Shortest Path using BFS is " << dist << " (Time taken: " << elapsed\_time << " seconds)" << endl;

else

cout << "Shortest Path doesn't exist." << endl;

} else {

cout << "Invalid choice. Please select 1 for DFS or 2 for BFS." << endl;

}

    return 0