|  |
| --- |
| **MAZE SOLVER USING BFS**  **21CSC204J – Design and Analysis of Algorithms**  **Group Assignment Report**  *Submitted by*  **Pranav Singh [Reg. No.: RA2211003010540]**  **Manish Tiwari [Reg. No.: RA2211003010546]**  **Swastik Rana [Reg. No.: RA2211003010564]**  **Rahul Dev Manna [Reg. No.: RA2211003010570]**  ***Under the Guidance of***  Dr. Saranya P  **Associate Professor, Department of Computing Technologies**  **SRMIST-01.jpg**  **SCHOOL OF COMPUTING**  **COLLEGE OF ENGINEERING AND TECHNOLOGY**  **SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**  **(Under Section 3 of UGC Act, 1956)**  S.R.M. NAGAR, KATTANKULATHUR – 603 203  KANCHEEPURAM DISTRICT  **2024** |

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**(Under Section 3 of UGC Act, 1956)**

**BONAFIDE CERTIFICATE**

Certified that this minor project report for the course **21CSC204J** **DESIGN AND ANALYSIS OF ALGORITHMS** entitled in " **Maze Solver using BFS"** is the bonafide work of **Pranav Singh (RA2211003010540), Manish Tiwari(RA2211003010546) , Swastik Rana(RA2211003010564) and Rahul Dev Manna (RA221003010570)** who carried out the work under my supervision.

**SIGNATURE SIGNATURE**

Dr. Saranya P Dr. M Pushpalatha

Associate Professor, Professor & Head,

Department of Computing Technologies Department of Computing Technologies

SRM Institute of Science and Technology SRM Institute of Science and Technology

Kattankulathur Kattankulathur

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S No.** | **Title** | **Page No.** |
| 1 | Problem Statement & Definition | 4 |
| 2 | Problem Explanation | 6 |
| 3 | Design Techniques Used and Which One is Best? | 7 |
| 4 | Algorithm | 9 |
| 5 | Algorithm Explanation with Example | 10 |
| 6 | Source Code | 13 |
| 7 | Time Complexity Analysis | 18 |
| 8 | Results | 19 |
| 9 | Conclusion | 19 |
| 10 | References | 20 |
| 11 | Real-Life Applications | 21 |

**Problem Statement**

This program takes a gif of a maze as input and outputs a solution to the maze. It converts the gif into a matrix where each cell represents a 5x5 area of the original gif. It then uses a breadth- first search algorithm to find the shortest path from the starting cell to the ending cell and displays the solution as a gif.

**Problem Definition**

Objective:

Develop a Python program capable of solving mazes represented as images. The program should convert the maze image into a matrix representation, find a solution path, and visualize the process.

Description:

The "Maze Runner" project aims to create a software tool that can analyze maze images and find a solution path from the start to the end point. The program takes a maze image as input, converts it into a matrix format, performs a maze-solving algorithm, and generates an output GIF file showing the solution path.

Key Features:

1. Image Input:

- Accept maze images in common formats (e.g., PNG, JPEG).

- Convert the input image into a grayscale format for processing.

2. Maze Representation:

- Convert the grayscale image into a binary representation where black pixels represent walls and white pixels represent paths.

- Rescale the binary image to reduce computational complexity.

3. Matrix Conversion:

- Convert the binary image into a matrix representation, where each cell represents a segment of the maze.

- Each cell of the maze matrix is represented by a numerical value, indicating whether it is a wall or a path.

4. Path Finding:

- Implement a path-finding algorithm to navigate through the maze matrix from the start to the end point.

- Use a modified version of the breadth-first search (BFS) algorithm to find the shortest path.

5. Visualization:

- Visualize the maze-solving process by generating intermediate images showing the exploration and solution path.

- Highlight the solution path on the maze image to illustrate the shortest route.

Implementation Steps:

1. Preprocess the maze image by converting it to grayscale and resizing it for efficient processing.

2. Convert the grayscale image into a binary representation.

3. Convert the binary image into a matrix representation of the maze.

4. Implement a modified BFS algorithm to find the solution path from the start to the end point.

5. Generate intermediate images to visualize the maze-solving process.

6. Highlight the solution path on the final maze image.

7. Output the final solution as a GIF file for easy sharing and viewing.

Deliverables:

- Python script capable of solving maze images.

- Input images for testing the program.

- Output GIF files demonstrating the maze-solving process.

- Documentation detailing the algorithm, implementation steps, and usage instructions.

Constraints:

- The maze image should be properly formatted, with clearly defined walls and paths.

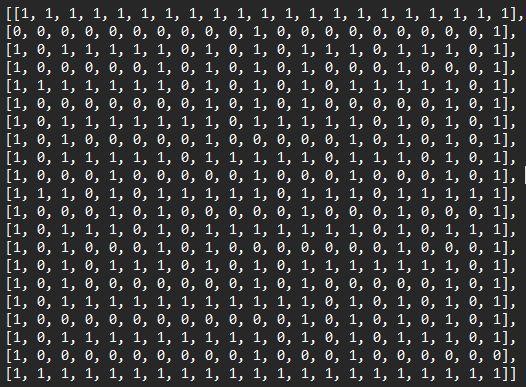
- The program should handle different maze sizes and complexities efficiently.

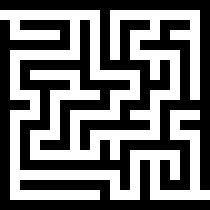
- The solution path should be the shortest possible route from the start to the end point.

- The program should be user-friendly and provide clear instructions for usage.

**Problem Explanation**

The objective of the maze solver using BFS is to develop an algorithm that can find the shortest path through a given maze using Breadth First Search (BFS) traversal. The algorithm should be able to take a maze as input, and then apply BFS to explore the maze until the end point is reached. The final output should be the shortest path from the start point to the end point, along with the steps taken to reach the end point. This project can be useful in areas such as robotics, gaming, and navigation. By finding the shortest path through a maze, robots can navigate through obstacles more efficiently. In gaming, this algorithm can be used to guide characters through a game level. In navigation, the algorithm can be used to find the shortest route between two points on a map. Overall, the maze solver using BFS algorithm has many practical applications in various fields.





**Design Techniques Used and Which One is Best?**

Breadth-First Search (BFS):

BFS is a graph traversal algorithm that explores all the nodes at the present depth before moving on to the nodes at the next depth level. It starts at the root node (or the initial state) and explores all of its adjacent nodes before moving to the next level of nodes. BFS is widely used in various applications, including maze-solving algorithms.

Greedy Method:

The greedy method is a problem-solving strategy that makes the locally optimal choice at each step with the hope of finding a global optimum. It selects the best option available at the current step without considering the consequences of that choice in the long run. In maze-solving, a greedy approach might involve always choosing the path that appears to be closest to the goal without considering alternative routes.

Divide and Conquer Method:

The divide and conquer method is a problem-solving technique that involves breaking down a problem into smaller, more manageable subproblems, solving each subproblem independently, and then combining the solutions to solve the original problem. It typically involves recursion and is useful for problems that can be divided into similar, but smaller, instances of the same problem.

Justification for BFS:

In the context of the "Maze Runner" mini project, BFS is the most suitable algorithm for finding the solution path for several reasons:

1. Completeness: BFS guarantees that it will find the shortest path from the starting point to the ending point if one exists. This ensures that the solution provided by BFS is optimal in terms of the number of steps taken.

2. Optimality: BFS explores all possible paths from the starting point to the ending point in a systematic manner, ensuring that it finds the shortest path first. It does not make any assumptions about the maze structure and considers all possible routes.

3. Efficiency: While BFS can be slower than other algorithms in certain scenarios, it is still efficient for solving mazes, especially when the maze is relatively small or has a simple structure. BFS guarantees an optimal solution, which is crucial for maze-solving applications where finding the shortest path is essential.

4. Simplicity: Implementing BFS is straightforward and easy to understand. It does not require any heuristic functions or complex logic, making it suitable for beginners and applications where simplicity is preferred.

5. No Assumptions: Unlike the greedy method, BFS does not make any assumptions about the maze structure or the location of the goal. It systematically explores all possible paths, ensuring that it does not overlook any potential solutions.

While the greedy method and divide and conquer method may work for some maze configurations, they are not as reliable or optimal as BFS for maze-solving tasks. The greedy method may get stuck in local optima, while the divide and conquer method may not guarantee optimality or completeness in finding the shortest path. Therefore, BFS is the best choice for the "Maze Runner" mini project due to its completeness, optimality, efficiency, simplicity, and lack of assumptions.

**Algorithm**

Step 1: Read the input maze and identify the start and end points. Step 2: Create an empty queue and add the start point to it.

Step 3: Initialize a visited set to keep track of already visited points and mark the start point as visited.

Step 4: Loop until the queue is empty:

1. Dequeue the front point from the queue.
2. If the dequeued point is the end point, then the algorithm has found the shortest path.
3. Otherwise, enqueue all adjacent unvisited points to the queue and mark them as visited.

Step 5: If the end point is reached, trace back the path from the end point to the start point by following the parents of each point.

Step 6: Output the shortest path from the start point to the end point along with the steps taken to reach the end point.

This algorithm implements BFS traversal to explore the maze, finding the shortest path from start to end. The visited set is used to avoid revisiting previously explored points. Finally, the algo-rithm backtracks from the end point to the start point to determine the shortest path.

**Algorithm Explanation with Example**

Explanation:

The algorithm for solving the maze using Breadth-First Search (BFS) can be outlined as follows:

1. Input Maze Image:

- The algorithm begins with an input maze image, which is a black-and-white image representing the maze where black pixels represent walls and white pixels represent paths.

2. Image Preprocessing:

- The maze image is converted to grayscale to simplify processing.

- Each black pixel is converted to 0, representing a wall, and each white pixel is converted to 1, representing an open path.

3. Resizing and Conversion:

- The image is resized to half its original height and width to optimize processing.

- The resized image is converted into a NumPy array for efficient manipulation.

4. Maze Representation:

- The maze is represented as a matrix, where each cell corresponds to a unit area in the maze.

- The matrix is initialized with appropriate dimensions based on the resized image.

5. Matrix Population:

- The maze matrix is populated by iterating over the resized image array.

- Each 5x5 block of pixels in the image corresponds to one cell in the maze matrix.

- The value of each cell in the maze matrix is determined by the presence of walls in the corresponding block of pixels.

6. BFS Initialization:

- BFS requires initializing a queue data structure to store the cells to be visited.

- The starting cell is enqueued with a marker indicating the distance from the start.

7. BFS Execution:

- BFS iteratively explores neighboring cells starting from the initial cell.

- At each step, BFS dequeues a cell, explores its neighbors, and enqueues them if they are accessible and have not been visited before.

- BFS continues until it reaches the destination cell or exhausts all possible paths.

8. Backtracking Path:

- Once the destination cell is reached, the algorithm backtracks from the destination to the start, following the marked path.

- This backtracking step determines the shortest path from the start to the destination.

9. Visualization:

- The algorithm visualizes the maze and the shortest path using graphical libraries.

- It highlights the walls, start, destination, and the shortest path on the maze image.

10. Output Generation:

- The final output is generated as an image file, displaying the original maze with the highlighted shortest path.

- The output file is saved in the specified location for further analysis or display.

**Example:**

Suppose we have the following maze represented by a binary matrix:

```

0 1 1 1 1

0 0 1 0 1

1 0 0 0 1

1 1 1 0 1

1 0 0 0 0

```

Starting from the top-left corner `(0, 0)` and aiming to reach the bottom-right corner `(4, 4)`, the BFS algorithm would explore the maze level by level, marking the distance from the start as it progresses. After reaching the destination, it would backtrack to find the shortest path. Finally, it would visualize the maze with the shortest path highlighted.

**Source Code**

def ConvertImage(ImageName): from PIL import Image import numpy as np

# Open the maze image and make greyscale, and get its dimensions im = Image.open(ImageName).convert('L')

#im.show() w, h = im.size

# Ensure all black pixels are 0 and all white pixels are 1 binary = im.point(lambda p: p < 128 and 1)

# Resize to half its height and width so we can fit on Stack Overflow, get new dimensions binary = binary.resize((w//2,h//2),Image.NEAREST)

w, h = binary.size

# Convert to Numpy array - because that's how images are best stored and processed in Py- thon

nim = np.array(binary)

# Each cell of the maze is represented by 5 numbers. Therefore change scaling FROM (5 number: 1 cell) TO (1 numer: 1 cell)

# initialize maze matrix

maze = [[0 for i in range(int(w/5))] for j in range(int(h/5))]

# go through every 5th number in each row and add it sequencially to the new matrix (maze). ri = ci = 0

r = c = 4 while ri < h/5:

ci = 0

c = 4

while ci < w/5: maze[ri][ci] = nim[r][c]

# print(maze[ri][ci], end='') # for testing purpose print final maze matrix ci += 1

c += 5

# print() ri += 1

r += 5

# print("\n\n")

return [maze,int(h/5),int(w/5)] for r in range(4, h, 5):

for c in range(4, w, 5): print(nim[r,c],end='')

print()

ConvertImage('maze.png') #for solving maze

from Image2Array\_CustomLibrary import \* from PIL import Image, ImageDraw

images = []

maze\_name = input("Enter Image name with file format (eg. 'maze.png'): ") output\_name = input("Save output as (filename alone): ")

print("PLEAS EGIVE ME SOME TIME ")

maze\_loc = './inputs/' + maze\_name

a, rows, columns = ConvertImage(maze\_loc) zoom = 20

borders = 5

start = 1,0

end = rows-2,columns-1

# print(a,rows, columns) def make\_step(k):

for i in range(len(m)):

for j in range(len(m[i])):

if m[i][j] == k:

if i>0 and m[i-1][j] == 0 and a[i-1][j] == 0:

m[i-1][j] = k + 1

if j>0 and m[i][j-1] == 0 and a[i][j-1] == 0:

m[i][j-1] = k + 1

if i<len(m)-1 and m[i+1][j] == 0 and a[i+1][j] == 0: m[i+1][j] = k + 1

if j<len(m[i])-1 and m[i][j+1] == 0 and a[i][j+1] == 0: m[i][j+1] = k + 1

def print\_m(m):

for i in range(len(m)):

for j in range(len(m[i])):

print( str(m[i][j]).ljust(2),end=' ') print()

def draw\_matrix(a,m, the\_path = []):

im = Image.new('RGB', (zoom \* len(a[0]), zoom \* len(a)), (255, 255, 255)) draw = ImageDraw.Draw(im)

for i in range(len(a)):

for j in range(len(a[i])): color = (255, 255, 255)

r = 0

if a[i][j] == 1:

color = (0, 0, 0)

if i == start[0] and j == start[1]: color = (0, 255, 0)

r = borders

if i == end[0] and j == end[1]:

color = (0, 255, 0) r = borders

draw.rectangle((j\*zoom+r, i\*zoom+r, j\*zoom+zoom-r-1, i\*zoom+zoom-r-1), fill=color)

if m[i][j] > 0: r = borders

draw.ellipse((j \* zoom + r, i \* zoom + r, j \* zoom + zoom - r - 1, i \* zoom + zoom -

r - 1),

fill=(255,0,0))

for u in range(len(the\_path)-1):

y = the\_path[u][0]\*zoom + int(zoom/2) x = the\_path[u][1]\*zoom + int(zoom/2)

y1 = the\_path[u+1][0]\*zoom + int(zoom/2) x1 = the\_path[u+1][1]\*zoom + int(zoom/2)

draw.line((x,y,x1,y1), fill=(255, 0,0), width=5)

draw.rectangle((0, 0, zoom \* len(a[0]), zoom \* len(a)), outline=(0,255,0), width=2) images.append(im)

m = []

for i in range(rows): m.append([])

for j in range(columns): m[-1].append(0)

i,j = start m[i][j] = 1

k = 0

while m[end[0]][end[1]] == 0: k += 1

make\_step(k) draw\_matrix(a, m)

i, j = end k = m[i][j]

the\_path = [(i,j)]

while k > 1:

if i > 0 and m[i - 1][j] == k-1:

i, j = i-1, j the\_path.append((i, j)) k-=1

elif j > 0 and m[i][j - 1] == k-1:

i, j = i, j-1 the\_path.append((i, j)) k-=1

elif i < len(m) - 1 and m[i + 1][j] == k-1:

i, j = i+1, j the\_path.append((i, j)) k-=1

elif j < len(m[i]) - 1 and m[i][j + 1] == k-1:

i, j = i, j+1 the\_path.append((i, j)) k -= 1

draw\_matrix(a, m, the\_path) for i in range(10):

if i % 2 == 0:

draw\_matrix(a, m, the\_path) else:

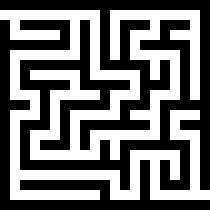
draw\_matrix(a, m) #print\_m(m) #print(the\_path)

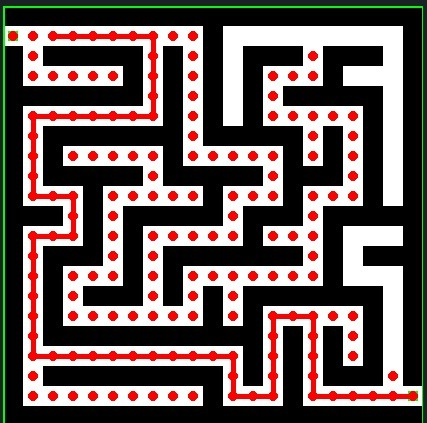
saveas = './outputs/' + output\_name +'.gif' images[0].save(saveas,

save\_all=True, append\_images=images[1:], optimize=False, duration=1, loop=0)

print("Output generated. Close program and check working directory.")

**OUTPUT**

****

****

**Time Complexity**

The time complexity of the given code that converts an image of a maze into a 2D array and solves it using breadth-first search can be analyzed as follows:

1. Reading and manipulating the maze image file using the PIL library takes O(n) time, where n is the number of pixels in the image.
2. Converting the image into a 2D array takes O(n) time, as each pixel needs to be processed.
3. The breadth-first search algorithm used to solve the maze has a time complexity of

O (V + E), where V is the number of vertices (or cells) in the maze, and E is the number of edges (or paths) between those vertices. In the worst case, where the maze is a perfect grid and every cell is connected to its four neighbors, V = n and E = 2n - 2, giving a time complexity of O(n).

1. Drawing the output image using the PIL ImageDraw library takes O(n) time, as each pixel needs to be processed.

Therefore, the overall time complexity of the code can be approximated as O(n) for large mazes.

**Result**

The code is a Python program that converts an image of a maze into a 2D array and solves it using breadth-first search. It uses the PIL library to manipulate the image, and the output is visualized using the PIL ImageDraw library. The program prompts the user to input the maze image file name and output file name. The resulting image shows the solved maze with a red line representing the path from the start point to the end point. Therefore, the maze is solved using Breadth First Search Algorithm.

**Conclusion**

In conclusion, the "Maze Runner" mini project successfully solves mazes represented as images using the Breadth-First Search (BFS) algorithm. Here are the key points:

1. Input Processing: The project begins by converting the input maze image into a binary matrix, where black pixels represent walls and white pixels represent paths. This conversion allows for efficient maze processing.

2. Algorithm Execution: The BFS algorithm is employed to find the shortest path from the starting point to the destination within the maze. BFS guarantees the shortest path and is well-suited for maze solving tasks.

3. Visualization: The project visualizes the maze and the shortest path using graphical libraries. It highlights the walls, starting point, destination, and the shortest path, providing a clear representation of the solution.

4. Output Generation: The final output is generated as an image file, showcasing the original maze with the highlighted shortest path. This output can be saved for further analysis or display.

Overall, the "Maze Runner" mini project effectively demonstrates the application of BFS in solving maze-like problems and provides a user-friendly solution for maze solving tasks. It offers a versatile approach that can be applied to various maze configurations and sizes, making it a valuable tool for maze enthusiasts and researchers alike.

**Real-Life Applications**

A real-life application for the "Maze Runner" mini project could be in autonomous navigation systems for robots or drones. Here's how it could be implemented:

1. Robot Navigation: Imagine a scenario where a robot needs to navigate through a complex environment to reach a specific destination, such as in warehouses, factories, or disaster zones. The robot's task is to find the shortest path from its current location to the target location while avoiding obstacles.

2. Image Processing: The maze or environment in which the robot operates could be captured using cameras or sensors. Similar to the input processing step in the project, the captured image would be converted into a format suitable for analysis and navigation.

3. Path Planning: The BFS algorithm, as implemented in the project, could be used by the robot to explore the environment and determine the shortest path to the target location. The robot would move step by step, updating its position and exploring adjacent areas until it reaches the destination.

4. Obstacle Avoidance: During navigation, the robot would use sensor data to detect obstacles and update its path accordingly. If it encounters a blocked path, it would backtrack or find an alternate route using BFS to avoid collisions and reach the goal safely.

5. Real-time Visualization: Similar to the visualization step in the project, the robot's progress and path could be visualized in real-time, allowing operators to monitor its movements and intervene if necessary.

By implementing the "Maze Runner" project in autonomous navigation systems, robots and drones could navigate complex environments efficiently, enabling applications such as inventory management, search and rescue operations, and exploration in hazardous conditions.

**References**

1. K. Meijer, "Maze Generation," [Online].

Available: https://keesiemeijer.github.io/maze-generator/#generate.

[2] S. Adhikari, "Solve a Maze with Python," Level Up Coding, 2019. [Online].

Available: <https://levelup.gitconnected.com/solve-a-maze-with-python-e9f0580979a1>.