MAZE SOLVER USING BFS

MINOR PROJECT REPORT

By

KENISHA SURANA (RA2211027010078)
PRATYUSH KUMAR SINGH (RA2211027010088)
AKSHAT SHARMA (RA2211027010121)

Under the guidance of

Dr. Rajkumar K

In partial fulfilment for the Course

of

21CSC204J – DESIGN AND ANALYSIS OF ALGORITHMS

in Data Science and Business Systems

B. TECH
COMPUTER SCIENCE WITH SPECIALIZATION
IN BIG DATA ANALYTICS

FACULTY OF ENGINEERING AND TECHNOLOGY

SCHOOL OF COMPUTING

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR

MARCH 2024

SRM INSTITUTION OF SCIENCE AND TECHNOLOGYKATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that this Course Project Report titled **"MAZE SOLVER USING BFS"** is the bonafide work done by **Kenisha Surana (RA2211027010078), Pratyush Kumar Singh (RA2211027010088)** and **Akshat Sharma (RA2211027010121)** who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

SIGNATURE

Dr. Rajkumar K
Assistant Professor
Data Science and Business System
SRM Institute of Science and Technology
Kattankulathur

SIGNATURE

Dr. M. Lakshmi
Head of the Department
Data Science and Business Systems
SRM Institute of Science and Technology
Kattankulathur

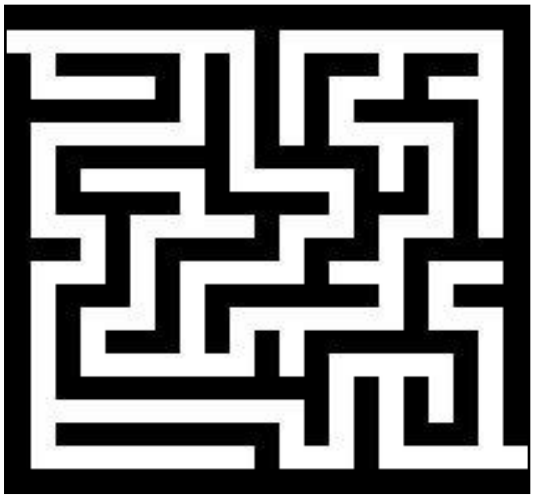
TABLE OF CONTENTS

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 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.



```
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1],
[1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1],
[1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1],
[1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1],
[1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1],
[1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
[1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1],
[1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1],
[1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1],
[1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1],
[1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
[1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1],
[1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1],
[1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

ALGORIT
HM

1. Read the input maze and identify the start and end points.

2. Create an empty queue and add the start point to it.

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

4. Loop until the queue is empty:
   a. Dequeue the front point from the queue.
   b. If the dequeued point is the end point, then the algorithm has found the shortest path.
   c. Otherwise, enqueue all adjacent unvisited points to the queue and mark them as visited.

5. If the end point is reached, trace back the path from the end point to the start point byfollowing the parents of each point.

6. Output the shortest path from the start point to the end point along with the steps taken toreach 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 algorithm backtracks from the end point to the start point to determine the shortest path.

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 Python
    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')

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("PLEASE GIVE 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)
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.")
```

TIME COMPLEXITY ANALYSIS

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 neighbours, V = n and E = 2n - 2, giving a timecomplexity of O(n).

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

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

COMPARISON OF ALGORITHMS AND TIME COMPLEXITY

5.1 DEPTH FIRST SEARCH (DFS):

Depth-First Search (DFS) is a widely used algorithm for traversing or searching trees or graph data structures. When applied to maze solving, DFS explores as far as possible along each branch before backtracking.

Time Complexity Analysis of DFS in a Maze Solver:

1. Graph Representation: Let's assume the maze is represented as a graph, where each cell is a node, and there are edges between adjacent cells (up, down, left, right). If the maze has (N) cells, the graph representation will have O(N) vertices and edges.

2. Visited Cells: During the DFS traversal, each cell is visited at most once. Therefore, the time complexity of visiting all cells is O(N).

3. Branching Factor: The branching factor of the DFS traversal depends on the maze structure. In the worst-case scenario, if each cell has b neighboring cells (i.e., the maximum branching factor), the DFS algorithm can potentially explore b^d nodes, where d is the maximum depth of the DFS tree.

4. Worst-case Time Complexity: In the worst-case scenario, DFS might visit all nodes in the graph before finding the solution. If the maze has (N) cells and the maximum branching factor is (b), then the worst-case time complexity of DFS for maze solving is O(b^N).

However, in practice, the actual performance of DFS in maze solving can vary significantly based on several factors:

- Maze Structure: The shape and complexity of the maze influences the number of nodes visited and the depth of the DFS traversal.

- Search Strategy: DFS doesn't guarantee finding the shortest path in the maze. Depending on the starting point and the order in which branches are explored, DFS might find a solution that is not the shortest path.

- Implementation Optimization: There are various optimizations possible in DFS implementation, such as pruning dead-end paths, using iterative DFS instead of recursive DFS, or employing techniques like memorization to avoid redundant computations.

In summary, while the worst-case time complexity of DFS for maze solving is exponential O(b^N), the actual performance depends on several factors and can often be improved with optimization techniques.

5.2 DIJKSTRA'S:

Dijkstra's Algorithm is another popular algorithm for solving maze-like problems, particularly for finding the shortest path from a source node to all other nodes in a weighted graph. Here's how Dijkstra's Algorithm compares to DFS in the context of maze solving:

1. Graph Representation: Similar to DFS, Dijkstra's Algorithm requires the maze to be represented as a graph. Each cell is a node, and there are edges between adjacent cells, but Dijkstra's Algorithm also considers edge weights, which represent the cost of moving from one cell to another.

2. Visited Cells: Dijkstra's Algorithm also visits each cell once, similar to DFS. However, it keeps track of the shortest distance from the source to each cell.

3. Priority Queue: Dijkstra's Algorithm utilizes a priority queue to select the next node to explore based on the shortest distance discovered so far. This ensures that nodes closer to the source are explored first.

4. Time Complexity: The time complexity of Dijkstra's Algorithm depends on the data structure used to implement the priority queue. Using a binary heap, the time complexity is O((V + E)log V), where V is the number of vertices (cells) and E is the number of edges (possible moves between cells).

- → In the case of a maze, where V represents the number of cells and E represents the number of edges between cells, the overall time complexity of Dijkstra's Algorithm is significantly influenced by the density of the maze and the number of possible moves from each cell.
- → In a sparse maze (few edges per cell), Dijkstra's Algorithm can be relatively efficient, potentially finding the shortest path quickly.
- → In a dense maze (many edges per cell), Dijkstra's Algorithm may take longer to compute due to the larger number of edges to explore.

5. Optimality: Dijkstra's Algorithm guarantees finding the shortest path from the source to all other reachable nodes in the maze, making it suitable for maze solving tasks where finding the shortest path is crucial.

In summary, while Dijkstra's Algorithm can find the shortest path efficiently in many maze-like problems, its time complexity can vary based on the density of the maze and the chosen data structure for implementing the priority queue. It guarantees optimality but may be computationally expensive in dense mazes.

5.3 RECURSIVE DIVISION:

Recursive Division is a maze generation algorithm rather than a maze-solving algorithm like DFS or Dijkstra's Algorithm. It works

by recursively dividing a maze into smaller sections until it reaches a desired level of complexity. However, it can indirectly impact the solving process if the generated maze is subsequently solved using DFS or Dijkstra's Algorithm. Here's how Recursive Division compares:

1. Maze Generation: Recursive Division creates a maze by recursively dividing the space into smaller sections using walls. This process typically involves creating horizontal or vertical walls at random positions and then recursively dividing the sections created by those walls until a desired level of complexity is achieved.

2. Structure: The maze generated by Recursive Division can vary in structure depending on factors such as the initial configuration and the specific implementation of the algorithm. It may have open spaces, dead ends, loops, and corridors.

3. Maze Solving: Once a maze is generated using Recursive Division, it can be solved using various algorithms like DFS or Dijkstra's Algorithm. The structure of the maze, including the presence of corridors, dead ends, and loops, will influence the performance and efficiency of the solving algorithm.

4. Time Complexity: The time complexity of Recursive Division for maze generation depends on the size of the maze and the complexity of the division process. It typically has a time complexity of O(n^2 \log n), where n is the size of the maze grid. This complexity arises from the recursive nature of the algorithm and the overhead of dividing the maze.

5. Effect on Maze Solving Algorithms: The structure of the maze generated by Recursive Division can affect the performance of maze-solving algorithms. For example, DFS might get stuck in infinite loops if the maze contains cycles, while Dijkstra's Algorithm might find suboptimal paths if the maze has multiple paths with different lengths.

In summary, Recursive Division is primarily a maze generation

algorithm, and while it indirectly influences the maze-solving process, it is not directly comparable to maze-solving algorithms like DFS or Dijkstra's Algorithm. It generates mazes with various structures, which in turn affect the performance and efficiency of solving algorithms.

Thus, on comparison

S. No	Algorithm	Time Complexity
1.	BFS	O(V+E)
2.	DFS	O(V+E)
3.	Dijkstra	O((V+E)logV)
4.	Recursion	O(n^2)

RESULT

The code is a Python program that converts an image of a maze into a 2D array and solvesit using breadth-first search. It uses the PIL library to manipulate the image, and the outputis 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.

REFRENCES

[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.

[3] Muhammad Ahsan Naeem, "A Python Module For Maze Search Algorithms". Available: https://towardsdatascience.com/a-python-module-for-maze-search-algorithms-64e7d1297c96

[4] Aryan Abed-Esfahani, "Maze Generation – Recursive Backtracking". Available: https://aryanab.medium.com/maze-generation-recursive-backtracking-5981bc5cc766