CS 5800 Final Project Report

Optimal Maze Solving Using Greedy Best First Search and Floyd-Warshall Algorithms

Group Member: Zheng Zheng, Yu Swe Zin Aung

1) Introduction – Yu Swe Zin Aung

This project outlines a compelling project aimed at addressing the challenges of solving complex mazes through an innovative hybrid approach that combines the Greedy Best First Search algorithm and the Floyd-Warshall algorithm. By capitalizing on the heuristic properties of the Greedy Best First Search and the all-pairs shortest path solution provided by the Floyd-Warshall algorithm, this project seeks to develop a highly efficient and effective maze-solving solution.

The task of solving mazes holds significant relevance in modern computer science as it mirrors real-world challenges and exemplifies the domain of pathfinding. Maze-solving algorithms have diverse applications in robotics, navigation systems, video games, and other fields. The intricacies and constraints presented by mazes serve as a foundation for designing and evaluating various algorithms. In this proposal, we explore the integration of two key algorithms - the Greedy Best First Search and the Floyd-Warshall - to tackle the maze problem effectively.

The primary objective of this project is to explore the complexities of maze-solving by investigating and analyzing two prominent algorithms: the Greedy Best First Search and the Floyd-Warshall algorithms. Through this investigation, we aim to contribute novel insights and concrete advancements to the field of pathfinding algorithms. To achieve this overarching goal, we have set specific objectives:

Objective 1: We will thoroughly examine the Greedy Best First Search algorithm, uncovering its strengths, weaknesses, and optimal use cases in solving mazes. We will conduct an in-depth investigation of the Floyd-Warshall algorithm, exploring its unique characteristics and how it can be effectively applied to maze problems.

Objective 2: Our focus will be on identifying potential areas for improvement within each algorithm, particularly in terms of performance and accuracy when solving mazes. We will develop and implement tailored methods to optimize each algorithm for maze-solving scenarios. To validate the effectiveness of the enhancements, we will rigorously test the improved algorithms on various maze configurations.

By pursuing these objectives, our project aspires to contribute meaningful advancements in the field of maze-solving algorithms, ultimately paving the way for more efficient and reliable pathfinding solutions applicable in diverse domains.

2) Algorithm Overview

a) Greedy Best First Search Algorithm – Yu Swe Zin Aung

The Best-First search algorithm is a method employed in state-space search, aiming to navigate from an initial state towards a goal state efficiently. This algorithm relies on two key lists, the open and closed lists, similar to the Breadth-First search approach. At each step of the process, the algorithm organizes the open list by utilizing a heuristic function, which evaluates the potential of each state. The fundamental principle of Best-First search involves selecting the unvisited state with the most favourable heuristic value for exploration in the subsequent step.

The Best-First search algorithm employs a systematic approach to address the challenge of traversing state spaces, where the objective is to uncover a possible path connecting the initial state with a goal state. This method manages an open list containing states ready for expansion and a closed list containing previously explored states. The algorithm's progression begins with an empty fast list and an open list that exclusively accommodates the initial state. The expansion of a state entails the generation of its potential successors and those successors that have yet to appear in either the open or closed lists are added to the open list. As states are expanded, they transit from the open to the closed list. The algorithm succeeds when a goal state is produced, while a failure outcome occurs when the open list becomes depleted, signifying the inability to continue expanding states further.

When confronted with multiple potential states to expand, the core of the Best-First search's efficiency lies in the selection of the most appropriate state for expansion. The algorithm consistently selects the state with the lowest heuristic value among the available candidates. When multiple states possess equally common heuristic values, the algorithm's behaviour becomes less deterministic, and diverse choices of these minimizers lead to distinct operational outcomes of the Best-First search. This selection process among possible behaviours is called a tie-breaking strategy, shaping the algorithm's behaviour in navigating complex state spaces.

b) Floyd-Warshall Algorithm – Zheng Zheng

The Floyd-Warshall algorithm, conceived by Robert Floyd, Stephen Warshall, and Bernard Roy in the 1960s, is a pivotal algorithm in graph theory for identifying shortest paths within a weighted graph. This algorithm operates effectively on both directed and undirected graphs and has the distinct advantage of accommodating negative weight edges, provided the graph does not contain negative weight cycles. In essence, the algorithm comprises three nested loops that iteratively refine an initially populated matrix, dist, where dist[i][j] represents the shortest known distance from vertex i to vertex j. Specifically, the procedure is as follows:

Initialize matrix dist such that dist[i][j] is set to the weight of the edge between vertices i and j if an edge exists, to infinity if vertices i and j are not directly connected, and to 0 when i equals j.

For each vertex, k, iterate over every pair of vertices (i, j), and systematically update dist[i][j] as min(dist[i][j], dist[i][k] + dist[k][j]). This algorithm, while elegant in its simplicity, incurs a time complexity of O(V^3), where V denotes the number of vertices in the graph. Thus, although not as efficient as some other shortest-path algorithms for sparse graphs, it shines in scenarios involving dense graphs. Turning to the realm of maze pathfinding, the Floyd-Warshall algorithm offers significant utility. Here, the maze is interpreted as a graph in which each cell is a vertex. Adjacent cells are connected by edges, with weights indicative of the cost associated with transitioning from one cell to another. This algorithm excels in scenarios necessitating the pre-computation of shortest paths between all pairs of points within the maze. Such pre-computation proves especially valuable in contexts where the maze remains static, but the start and end points are subject to frequent change. For instance, in gaming environments where the maze configuration is fixed but players' positions are variable, employing the Floyd-Warshall algorithm can markedly expedite subsequent pathfinding queries following the initial pre-computation phase.

3) Implementation and Analysis

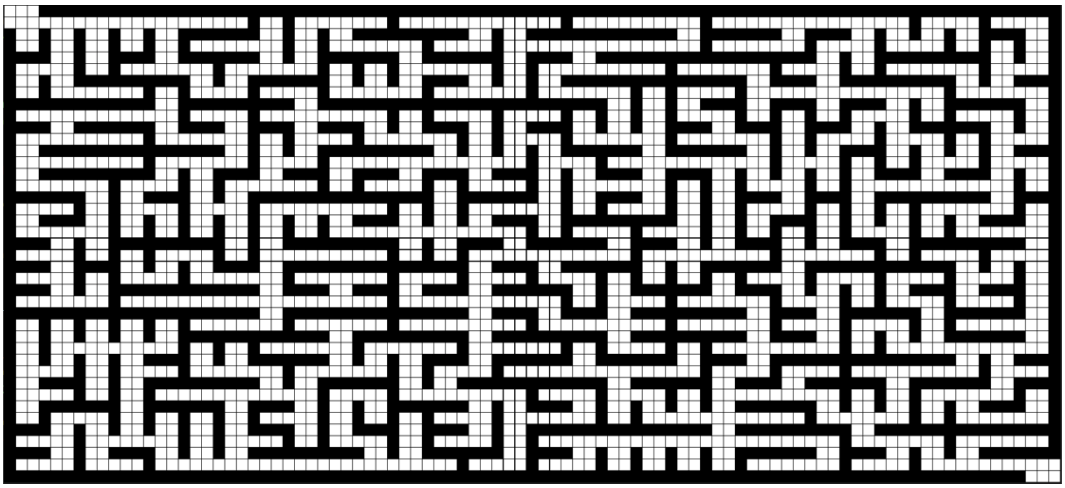
a) Data and Image Importing – Yu Swe Zin Aung

We have a maze represented as a 2D matrix with dimensions 41x90 where each cell in the matrix corresponds to a 20x20 pixel area in a maze single-cell image. This mapping allows you to represent the maze using binary data visually.

1. Matrix Representation:
    - The maze is represented as a 2D matrix, where each cell can have a value of 0 (representing walls or obstacles) or 1 (representing open paths).

2. Maze Image:
    - The maze image visually represents the layout of the maze.

- Each cell in the matrix corresponds to a 20x20 pixel area in this maze image.
- The image itself matches the dimensions of the matrix.



3. Binary Data Plotting:
   - The binary data from the matrix determines the maze layout in the image.

   For each cell in the matrix:

   - If the cell value is 0, it corresponds to a wall in the maze image.
   - If the cell value is 1, it corresponds to an open path in the maze image.

We can visualize the maze's layout by plotting the binary data from the matrix onto the maze image. The binary data guides determine which parts of the image correspond to walls and which parts correspond to open paths.

The Pathfinder class uses a binary matrix to visualize the maze and translate cell coordinates to pixel coordinates, effectively mapping binary data to the maze's visual representation.

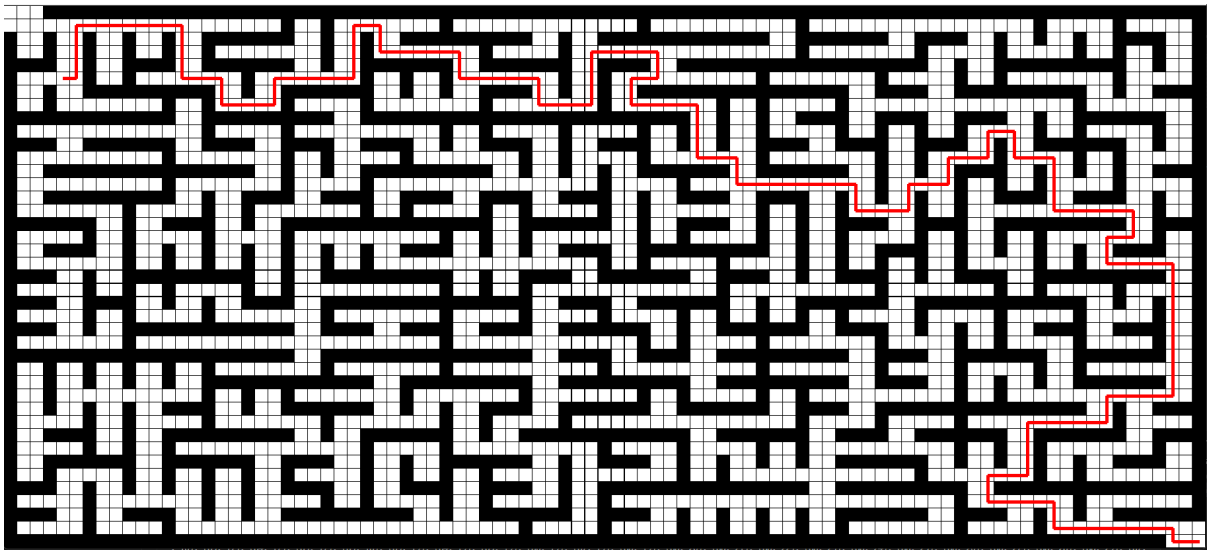
b) Implementing binary Maze – Yu Swe Zin Aung

The implementation of a binary maze involves the creation of a program that can effectively represent, visualize, and solve mazes through the use of binary matrices. We can break down the following steps involved in implementing a binary maze :

1. Maze Representation: The maze is a 2D binary matrix where each cell can be blocked (0) or unblocked (1). The matrix defines the maze layout, where blocked cells represent walls or obstacles, and unblocked cells represent open paths.
2. Pathfinding Algorithms: The code includes the Greedy Best First Search algorithm and the Floyd-Warshall algorithm for finding the shortest path between two points in the maze.

3. Graphical User Interface (GUI): The Pygame library creates a graphical user interface for interacting with the maze. The GUI displays the labyrinth as an image and allows users to set the start and end points by clicking on cells.
4. Path Visualization: Once the start and end points are set, the algorithm calculates the shortest path and visualizes it on the maze. The course is drawn as a red line connecting the cells that form the fastest route.
5. User Interaction: The pathfinding algorithm is triggered to calculate the shortest path when users click cells on the GUI to set the start and end points.
6. Real-time Visualization: The GUI updates in real-time, ensuring that the progress of the pathfinding algorithm and the resulting shortest path are consistently displayed. Users can closely monitor the algorithm as it explores the maze to find the optimal route efficiently.
7. Termination: The Pygame library handles event handling and window management, and the application can be terminated by closing the window or exiting the program.

This implementation provides users with a visual exploration of binary mazes, start and end point settings, and applying the Greedy Best First Search and Floyd-Warshall algorithm to find the shortest path and observe the pathfinding process. It offers an interactive approach to understanding pathfinding algorithms' functionality and their capability to navigate maze-like environments.



```
# Steps to run path finding algorithm
1. `cd` into <code>path_finding_assignment</code> directory
2. Run <code>python pathfinding_maze.py</code>
3. A maze map will appear and first click for the starting point
4. Second click for end/destination point
5. You can click again for the next path finding without restarting the program
```

c) Greedy Best First Search Algorithm – Yu Swe Zin Aung

i) General Implementation

The Greedy Best First Search (GBFS) algorithm is a Best First Search algorithm variant. It uses a heuristic function to evaluate nodes and prioritize exploration based on their heuristic values. GBFS is a greedy algorithm as it makes the most promising choice at each step without considering the complete path's optimality. Here is the implementation of the Greedy Best First Search algorithm:

1. Initialize the Open List:

- The Open List contains the nodes yet to be explored and is initially empty.

2. Add the starting node to the Open List.
3. Loop until the Open List is empty, or the goal node is found:
    a. Choose the node with the best heuristic value from the Open List. This node represents the most promising path based on the heuristic evaluation function.
    b. If the selected node is the goal node, terminate the search; the optimal path has been found.
    c. Move the selected node from the Open List to the Closed List to mark it as visited.
    d. Expand the selected node by generating its successor nodes (neighbors) and evaluating them using the heuristic function.
    e. For each successor node:
        - If the node is not already in the Open or Closed List, add it to the Open List.
        - If the node is already in the Open List, update its heuristic value if the newly calculated value is better.
4. If the Open List becomes empty and the goal node has not been found, no path exists.

The heuristic evaluation function is critical to the Greedy Best First Search algorithm. It estimates the cost from a given node to the goal node. Like in the regular Best First Search, the heuristic function should be admissible (never overestimates the actual cost to the goal) and consistent (satisfies the triangle inequality).

A heuristic function, denoted as h(x), assesses the consecutive nodes by considering their proximity to the target node, favouring options with lower immediate costs. However, despite this approach, it doesn't guarantee identifying the shortest path to the goal.

Consider a scenario where a robotic entity intends to transition from point A to point B. Within the framework of greedy best-first search, the robot opts for moving to a position that brings it nearest to the desired goal, without necessarily considering if an alternative part might ultimately provide a shorter distance. In situations involving obstacles, the algorithm

reevaluates past nodes based on their shortest distances to the goal and consistently selects the node that maintains the closest proximity to the objective.

By using GBFS, following function return the result if parth exist or not.

```python
def greedy_best_first_search(matrix, start, goal):
    visited = set()
    priority_queue = []
    heapq.heappush(priority_queue, (0, start))  # Priority queue format: (heuristic, (x, y))

    while priority_queue:
        #pop the node with the lowest heuristic value
        _, current = heapq.heappop(priority_queue)
        x, y = current

        if current == goal:
            return True  # Path found
        #add current node to the visisted set
        visited.add(current)

        #define the neighboring position fo the current node
        neighbors = [
            (x - 1, y),
            (x + 1, y),
            (x, y - 1),
            (x, y + 1)
        ]


        for neighbor in neighbors:
            nx, ny = neighbor

            #check neighbour is valid and not visited
            if is_valid(nx, ny, matrix) and neighbor not in visited:
                priority = abs(nx - goal[0]) + abs(ny - goal[1])  # Manhattan distance heuristic
                heapq.heappush(priority_queue, (priority, neighbor))
                visited.add(neighbor)

    return False  # Path not found
```

ii) Pseudocode of Greedy Best First Search

Place the starting node into the OPEN list.

If the OPEN list is empty, stop and return failure.

Remvoe the node n from the OPEN list which has the lowest heuristic value, and place it in the CLOSED list.

Expand the node n and generate the successors of node n.

Check each succesor of node n and find whether any node is a goal node or node.

If any success node is goal node,

then return success and terminate the search,

Else for each successor node, algorithm check if the node has been in eighter OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.

iii) Heuristic Function

GBFS algorithm uses a heuristic function to estimate the cost from a node to the goal node, providing an informed guess about node's proximity to the goal. In this project, Manhattan Distance calculation to get the heuristic value.

```
    for neighbor in neighbors:
        nx, ny = neighbor
        if is_valid(nx, ny, matrix) and neighbor not in parent_map:
            priority = abs(nx - goal[0]) + abs(ny - goal[1])  # Manhattan distance heuristic
            heapq.heappush(priority_queue, (priority, neighbor))
            parent_map[neighbor] = current
```

The function has been carefully designed for optimal efficiency and accuracy.

Characteristics of a good heuristic function for GBFS:

1. Admissibility: The heuristic function should never overestimate the actual cost from a node to the goal. In other words, the estimated cost should be less than or equal to the actual cost. GBFS is guaranteed to find the optimal solution if the heuristic is admissible.
2. Consistency: A heuristic function is consistent if the estimated cost from a node to its successor plus the estimated cost from the successor to the goal is no greater than the estimated cost from the node directly to the goal. Consistency ensures that GBFS expands nodes in an optimal order and guarantees that the first goal node found will be the optimal solution.
3. Information Gain: A good heuristic function should provide relevant information about the problem domain. The more accurate the heuristic, the more efficient the search process will be, as it will prioritize exploring nodes closer to the goal.

iv) Analysis of the implementation

We can analyze the implementation of GBFS by examining its advantages, limitations, and performance characteristics. To explore the critical aspects of the implementation:

Efficiency: GBFS is generally more efficient than uninformed search algorithms like Breadth-First Search (BFS) and Depth-First Search (DFS) because it uses a heuristic function to guide the search towards promising paths. This property allows GBFS to focus on exploring nodes closer to the goal, potentially reducing the search space significantly.

Greedy Approach: GBFS is a greedy algorithm that selects the most promising node based on the heuristic evaluation. In certain situations, this approach can lead to finding a solution faster when the heuristic function provides accurate estimates.

Optimality under Certain Conditions: If the heuristic function is admissible and consistent, GBFS is guaranteed to find the optimal solution. Admissibility ensures that the heuristic never overestimates the actual cost to the goal, while consistency ensures that the search proceeds optimally.

Time Complexity: Worst case time complexity: O(bm)

Space Complexity: Worst case space complexity: O(bm)

- b: Branching factor (average number of successors per state)
- m: Maximum depth of the search space

Limitation: Greedy Best-First Search is a method that only sometimes works perfectly. Even when we have a limited number of places to search, this method might need to find the correct answer. It sorts the places to look based on some heuristic values, but the technique might need to be corrected if those heuristic values could be better. It could go in circles or miss the best paths. One reason is that the process doesn't go back to check places it already looked at. So, while this method can be helpful, it only sometimes gives us the answer we need because of its limitations.

d) Floyd-Warshall Algorithm – Zheng Zheng

Problem Definition

The maze solving problem can be defined as follows: given a two-dimensional grid of cells with designated start and end positions, and where each cell is either open (a passage) or blocked (a wall), the task is to find a path from the start position to the end position through the open cells, if such a path exists. This problem can be represented as finding a path between two nodes in a graph where the nodes represent open cells of the maze and the edges represent adjacent open cells.

Objectives

The primary objective of this project is to develop an algorithm that efficiently and effectively solves a maze, i.e., it finds a valid path from a designated start point to a designated endpoint within a given twodimensional matrix representing the maze. Efficiency here refers to the computational resources (time and memory) that the algorithm uses. Effectiveness relates to the algorithm's ability to consistently find the shortest (or optimal) path, if one exists.

Our Approach

To solve this problem, we propose an adaptation of the Floyd-Warshall algorithm, a classic solution for finding the shortest paths between all pairs of vertices in a weighted graph. Traditionally, the Floyd- Warshall algorithm is used in scenarios where the complete set of shortest paths between every pair of vertices is required, as opposed to just a single pair of vertices. However, it can be adapted to the context of maze solving.

In our algorithm, we conceptualize the maze as a graph, where each cell in the maze corresponds to a vertex in the graph. Two vertices are connected by an edge if their corresponding cells are adjacent and open in the maze. The weight of each edge is set to 1, representing the cost of moving from one cell to another. Blocked cells in the maze are not represented in the graph.

```
def floyd_warshall(matrix):
    INF = float('inf')

    num_vertices = 0
    index_to_coords = {}
    coords_to_index = {}
    rows, cols = len(matrix), len(matrix[0])

    for y in range(rows):
        for x in range(cols):
            if matrix[y][x] == 1:
                index_to_coords[num_vertices] = (y, x)
                coords_to_index[(y, x)] = num_vertices
                num_vertices += 1

    dist_matrix = [[INF for _ in range(num_vertices)] for _ in range(num_vertices)]
    pred_matrix = [[None for _ in range(num_vertices)] for _ in range(num_vertices)]
    for i in range(num_vertices):
        dist_matrix[i][i] = 0

    for node, (y, x) in index_to_coords.items():
        neighbors = [(y - 1, x), (y + 1, x), (y, x - 1), (y, x + 1)]
        for ny, nx in neighbors:
            if (ny, nx) in coords_to_index:
                neighbor_node = coords_to_index[(ny, nx)]
                dist_matrix[node][neighbor_node] = 1
                pred_matrix[node][neighbor_node] = node
```

The floyd_warshall function takes as input a two-dimensional matrix, where 1's represent open cells (passages) and 0's represent blocked cells (walls). It computes the shortest path between every pair of open cells in the matrix. The algorithm initializes a distance matrix, where dist_matrix[i][j] represents the shortest path from vertex i to vertex j. It iteratively updates this matrix to ensure that it captures the shortest possible paths between all pairs of vertices.

```
    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                if dist_matrix[i][k] + dist_matrix[k][j] < dist_matrix[i][j]:
                    dist_matrix[i][j] = dist_matrix[i][k] + dist_matrix[k][j]
                    pred_matrix[i][j] = pred_matrix[k][j]

    return dist_matrix, pred_matrix, index_to_coords, coords_to_index
```

Our implementation also constructs a predecessor matrix (pred_matrix), which is used to reconstruct the actual path between any two given points. The reconstruct_path function takes as input the start and end coordinates, the pred_matrix, and a dictionary that maps coordinates to their corresponding vertex indices. It returns a list of vertex indices that make up the shortest path from the start to the end.

In addition to the core logic for finding the shortest path through the maze, our implementation includes a Pathfinder class that interfaces with a graphical representation of the maze using the pygame library. This class allows for visualization of the algorithm in action, as it navigates through the maze from a specified start point to an end point. The Pathfinder class contains methods for initializing and updating the visual representation, drawing the active path, and interacting with user input to set the start and end points for pathfinding.

Evaluation of the Algorithm

- Time Complexity

The time complexity of the original Floyd-Warshall algorithm is

$$O(V3)$$

where V is the number of vertices in the graph. In the context of our maze, V corresponds to the number of open cells in the maze. This cubic time complexity arises from the triple-nested loop structure of the algorithm, where it iteratively updates the shortest path between all pairs of vertices through every possible intermediate vertex.

Given a maze of size n×m, the maximum number of vertices V in the graph can be n×m (in a completely open maze with no walls). Thus, in the worst case, our adapted Floyd-Warshall algorithm runs in

$$O((n \cdot m) 3)$$

- Space Complexity

The primary data structures used by our adapted Floyd-Warshall algorithm are the distance matrix and the predecessor matrix. Each of these is a V×V matrix, where V is the number of vertices in the graph. Consequently, the space complexity of our algorithm is

$$O(V2)$$

In terms of the maze dimensions, this translates to (in the worst case)

$$O((n \cdot m) 2)$$

Practical Performance

Despite its worst-case cubic time complexity, our implementation of the Floyd-Warshall algorithm exhibits practical efficiency on a variety of maze sizes and complexities. This is partly because most realworld mazes have far fewer open cells (and thus vertices in the graph) than the worst-case n×m scenario. Additionally, the algorithm's regular and predictable memory access patterns can lead to effective use of CPU caching, further boosting its practical performance.

Contribution and Scope

This project, therefore, stands as a dual contribution: it adapts a well-established algorithm to a novel context, demonstrating the versatility of classic algorithms when applied to different types of problems. Furthermore, it provides a fully realized, visually engaging demonstration

of algorithmic pathfinding in action. While our focus is on the application of the Floyd-Warshall algorithm for maze solving, the framework we develop is flexible and can be extended to incorporate other pathfinding algorithms.

4) Performance Comparison – Zheng Zheng

a) Comparison of Run-time Across on Same Maze Dataset

4.1 Time Completiy

Greedy Best First Search (GBFS)

The time complexity of GBFS is dependent on the number of vertices V (open cells in the maze) and the size of the priority queue Q. In the worst case, every vertex is visited and added to the queue at least once. The time complexity of inserting an element into a priority queue implemented with a binary heap is O(logQ), and in worst-case, Q can be equal to V. Hence, the worst-case time complexity of GBFS is O(VlogV).

Floyd-Warshall Algorithm

The time complexity of the Floyd-Warshall algorithm is O(V3) , where V represents the number of open cells in the maze. This complexity is due to the triple-nested loop structure of the algorithm.

4.2 Space Complexity

Greedy Best First Search (GBFS)

The primary data structures in GBFS are the priority_queue and visited set. In the worst case, the priority_queue and visited set might need to store all vertices in the graph, which translates to a space complexity of O(V).

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm requires two matrices (the dist_matrix and pred_matrix), each of size V×V. Hence, the space complexity is $O(V^2)$.

4.3 Practical Considerations

Greedy Best First Search (GBFS)

- Optimality: GBFS does not guarantee the shortest path, as it is guided by a heuristic and may prioritize paths that seem promising but may lead to suboptimal solutions.
- Efficiency: GBFS generally exhibits good performance, especially when the heuristic is wellchosen, as it is directed and avoids exploring unnecessary paths.
- Use Case: GBFS is particularly effective for large mazes where an approximate solution is satisfactory, and we want to find a path quickly.

Floyd-Warshall Algorithm

- Optimality: The Floyd-Warshall algorithm guarantees the shortest path between any two nodes, making it optimal for cases where the shortest path is required.

- Efficiency: Despite its cubic time complexity, Floyd-Warshall is highly predictable and can be efficient for small to moderately sized mazes. However, for very large mazes, its performance might become a bottleneck.
- Use Case: It is particularly useful when we need to solve multiple queries for shortest paths in the same maze, as it precomputes all pairs shortest paths.

5) Conclusion – Zheng Zheng

In this study, we have undertaken a comprehensive comparison between the Greedy Best First Search (GBFS) algorithm and the Floyd-Warshall algorithm in the context of maze solving. The findings reveal distinct advantages and limitations associated with each algorithm, which are predominantly tied to their time and space complexities, optimality, and practical efficiency.

The GBFS algorithm stands out for its generally superior efficiency. With a worst-case time complexity of O(VlogV), it tends to be notably faster, especially as the size of the maze increases. This algorithm excels in scenarios where finding a quick, heuristic-guided solution is a priority. However, it is important to highlight that GBFS does not guarantee an optimal solution. The heuristic nature of GBFS, while enabling it to navigate towards the goal swiftly, can sometimes lead it down suboptimal paths. Therefore, GBFS is most apt for applications where the exact shortest path is not a stringent requirement and where rapid computation is more valued.

Conversely, the Floyd-Warshall algorithm, with its time complexity of , is more computationally intensive. Nevertheless, it has the undeniable advantage of producing the exact shortest path between any two points within the maze. This characteristic makes it the algorithm of choice when the optimal solution is non-negotiable. Additionally, Floyd-Warshall's ability to precompute all pairs of shortest paths makes it particularly useful when repeated queries for different start and end points within the same maze are expected, despite its higher computational cost.

In summary, the choice between GBFS and Floyd-Warshall should be guided by the specific requirements of the task at hand. For applications seeking rapid, near-optimal solutions in large mazes, GBFS is likely the superior choice. In contrast, for scenarios where the precise shortest path is paramount, and where computational resources are abundant, the Floyd-Warshall algorithm emerges as the more fitting solution

6) Key Takeaways from this Project

a) Yu Swe Zin Aung's Takeaway

As someone new to algorithms, this project of implementing the Greedy Best First Search algorithm for solving a binary maze has been quite the journey. Navigating the complexities of algorithms and seeing them in action with a real-world problem like maze solving has been exciting and challenging.

One of the key takeaways from this project is understanding the fundamental approach of the Greedy Best First Search algorithm. It is fascinating to see how the algorithm evaluates different paths based on a heuristic function. This heuristic function, which evaluates how promising a path is in reaching the goal, significantly impacts the algorithm's performance. It has been eye-opening to witness how even small changes in the heuristic can drastically affect the outcome.

However, I have realized that the Greedy Best First Search approach only sometimes guarantees the absolute best solution. It is like making decisions based on short-term gains – sometimes, it might miss out on better long-term solutions. The algorithm's inclination towards local optimization and its dependence on the quality of the heuristic function has been particularly intriguing exploration points.

Implementing this algorithm on binary mazes has brought to light the delicate balance between algorithm complexity and solution quality. The algorithm is fast and efficient, but there must be a trade-off regarding potential suboptimal solutions or getting stuck in certain situations. I have faced instances where the algorithm seemed trapped in some maze regions, unable to find a way out.

On a personal note, this project has been a significant learning experience. I have delved into concepts like heuristic functions, algorithm behaviour, and parameter tuning, which were all new to me. Seeing the algorithm work its way through the maze, sometimes finding a solution and others struggling, has given me a deeper appreciation for the nuances of algorithmic problem-solving.

As I reflect on this journey, I'm excited about the potential real-world applications of what I've learned. Route planning, robotics pathfinding, and other navigation-related challenges come to mind. This project has sparked my curiosity to explore more advanced algorithms and hybrid approaches that could overcome some of the limitations I've encountered.

In the end, this project marks just the beginning of my algorithmic exploration. I've learned that algorithms are like tools – each designed for a specific purpose, with strengths and weaknesses. It has been a fulfilling experience to apply these tools to a tangible problem, and I'm looking forward to the next steps in my journey of algorithm discovery.

b) Zheng Zheng's Takeaway

Optimal Path Solution:

The Floyd-Warshall algorithm stands as a robust method for finding the shortest paths between all pairs of nodes within a maze. Unlike some heuristic-based approaches, it guarantees an optimal solution, making it a reliable choice when the shortest path is required.

Time Complexity:

The algorithm has a cubic time complexity, where V is the number of vertices in the graph. For large mazes or frequent computations, this could lead to significant time overhead, and the choice of algorithm might need to be balanced with computational efficiency.

Precomputed Paths:

Floyd-Warshall's design allows it to precompute all pairs of shortest paths, which is beneficial for scenarios that involve repeated queries between different points in a maze. Once computed, the paths can be retrieved without additional computation, enhancing efficiency for such use-cases.

Space Complexity:

With a space complexity of , this algorithm can demand considerable memory, especially for large mazes. This is a key consideration when working with constrained computational resources.

Versatility and Generalizability:

Beyond mazes, the Floyd-Warshall algorithm is a general-purpose solution applicable to various types of weighted graphs. This versatility is an asset, extending its utility beyond the domain of maze solving.

Suitability for Static Environments:

Given its computational demand, the Floyd-Warshall algorithm is best suited for static or infrequently changing environments, where the precomputation of paths is both feasible and beneficial.

Handling of Negative Cycles:

One of the unique strengths of the Floyd-Warshall algorithm is its capability to detect negative cycles in a graph. While this feature may not be commonly required in maze solving, it extends the algorithm's applicability to a broader range of problems.

Predictable Performance:

Unlike heuristic methods, which can have varying performance based on the heuristic used, FloydWarshall's performance is predictable and consistent, irrespective of the input data.

7) Reference

I. Chen, X., & Qin, S. (2017). Approach to high efficient hierarchical pathfinding of indoor mobile service robots based on grid map and Floyd-Warshall algorithm. In 2017 Chinese Automation Congress (CAC) (pp. 6476-6483). Jinan, China. https://doi.org/10.1109/CAC.2017.8243944

II. Johner, R., Lanaia, A., Dornberger, R., & Hanne, T. (2022). Comparing the Pathfinding Algorithms A*, Dijkstra's, Bellman-Ford, Floyd-Warshall, and Best First Search for the Paparazzi Problem. In M. Saraswat, H. Sharma, K. Balachandran, J. H. Kim, & J. C. Bansal (Eds.), Congress on Intelligent Systems. Lecture Notes on Data Engineering and Communications Technologies (Vol. 111). Springer, Singapore. https://doi.org/10.1007/978-981-16-9113-3_41

III. Dela Cruz, J. C., Magwili, G. V., Mundo, J. P. E., Gregorio, G. P. B., Lamoca, M. L. L., & Villaseñor, J. A. (2016). Items-mapping and route optimization in a grocery store using Dijkstra's, Bellman-Ford and Floyd-Warshall Algorithms. In 2016 IEEE Region 10 Conference (TENCON) (pp. 243-246). Singapore. https://doi.org/10.1109/TENCON.2016.7847998

IV. Barnouti, N. H., Mahmood Al-Dabbagh, S. S., Sahib Naser, M. A., & Publishing, S. R. (2016, September 8). Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm. Pathfinding in Strategy Games and Maze Solving Using a* Search Algorithm. https://doi.org/10.4236/jcc.2016.411002

V. University of Basel, Switzerland. (n.d.). Best-Case and Worst-Case Behavior of Greedy Best-First Search. Https://Edoc.Unibas.Ch/64994/1/20180723153346_5b55d93ab3198.Pdf.

VI. L. (2022, January 27). GitHub - Leonardpepa/Pathfinding-Visualizer: Pathfinding Visualizer application that visualizes graph based search algorithms used to find the shortest path. Algorithms used: Breadth first search, Depth first search, Best first search and A* search made with java swing. GitHub. https://github.com/Leonardpepa/Pathfinding-Visualizer

VII. V. (2020, April 5). GitHub - VedantKhairnar/The-Maze-Runner: AI : Use of Greedy Best First Search Traversal to find route from Source to Destination in a Random Maze. GitHub. https://github.com/VedantKhairnar/The-Maze-Runner

VIII. clear-code-projects. (n.d.). Python-Pathfinder: roomba project - map.png. GitHub, from https://github.com/clear-code-projects/Python-Pathfinder/blob/main/roomba%20project/map.png

8) Appendix

a) Greedy Best First Search Algorithm

```python
import heapq


def is_valid(x, y, matrix):
    rows, cols = len(matrix), len(matrix[0])
    return 0 <= y < rows and 0 <= x < cols and matrix[y][x] == 1


def greedy_best_first_search(matrix, start, goal):
    visited = set()
    priority_queue = []
    heapq.heappush(priority_queue, (0, start))  # Priority queue format: (heuristic, (x, y))

    while priority_queue:
        #pop the node with the lowest heuristic value
        _, current = heapq.heappop(priority_queue)
        x, y = current

        if current == goal:
            return True  # Path found
        #add current node to the visisted set
        visited.add(current)

        #define the neighboring position fo the current node
        neighbors = [
            (x - 1, y),
            (x + 1, y),
            (x, y - 1),
            (x, y + 1)
        ]


        for neighbor in neighbors:
            nx, ny = neighbor

            #check neighbour is valid and not visited
            if is_valid(nx, ny, matrix) and neighbor not in visited:
                priority = abs(nx - goal[0]) + abs(ny - goal[1])  # Manhattan distance heuristic
                heapq.heappush(priority_queue, (priority, neighbor))
                visited.add(neighbor)

    return False  # Path not found
```

```python
def find_path(matrix, start, goal):
    #check if start and end nodes are valid on the marix
    if not is_valid(*start, matrix) or not is_valid(*goal, matrix):
        return None

    #return start node if start and end nodes are the same
    if start == goal:
        return [start]

    #path not exist by GBFS
    if not greedy_best_first_search(matrix, start, goal):
        return None

    #Closed List
    parent_map = {}

    #Open List
    priority_queue = []

    #Priority queue with 'Start'position
    heapq.heappush(priority_queue, (0, start))  # Priority queue format: (heuristic, (x, y))

    while priority_queue:
        _, current = heapq.heappop(priority_queue)
        x, y = current

        if current == goal:
            break
        #4 direction neighbours
        neighbors = [
            (x - 1, y),
            (x + 1, y),
            (x, y - 1),
            (x, y + 1)
        ]

        for neighbor in neighbors:
            nx, ny = neighbor
            if is_valid(nx, ny, matrix) and neighbor not in parent_map:
                priority = abs(nx - goal[0]) + abs(ny - goal[1])  # Manhattan distance heuristic
                heapq.heappush(priority_queue, (priority, neighbor))
                parent_map[neighbor] = current
```

```python

    path = []
    current = goal
    while current != start:
        path.append(current)
        current = parent_map[current]
    path.append(start)
    path.reverse()
    return path

```

b) Maze Frame Code

```python
import sys
import pygame
from pathfinding.core.grid import Grid
from greedy_best_first_search import find_path


class Pathfinder:
    def __init__(self, mtrix):
        self.matrix = mtrix
        self.grid = Grid(matrix=mtrix)
        self.path = []

    @staticmethod
    def get_coordinate():
        mouse_pos = pygame.mouse.get_pos()
        x, y = mouse_pos[0] // 20, mouse_pos[1] // 20
        return x, y

    # pathfinder
    def create_path(self, x1, y1, x2, y2, algorithm=None):
        start = (x1, y1)
        end = (x2, y2)

        print("start", start)
        print("end", end)

        if algorithm == "GBFS":
            self.path = find_path(self.matrix, start, end)
        elif algorithm == "FW":
            # TODO: add FW algorithm
            self.path = []

        print("total cells travelled:", len(self.path))
        print(self.path)
        self.grid.cleanup()

    def draw_path(self):
        if self.path:
            points = []
            for point in self.path:
                # plus 16 to draw the line till the centre of rectangle
                x = (point[0] * 20) + 10
                y = (point[1] * 20) + 10
                points.append((x, y))

            pygame.draw.lines(screen, '#FF0000', False, points, 5)

```

```python
    def update(self):
        self.draw_path()

def get_matrix():
    return [
[1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,0],
[0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0],
[0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,0],
[0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0],
[0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0],
[0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0],
[0,1,1,0,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0],
[0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0],
[0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0],
[0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,0],
[0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,0],
[0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,0],
[0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0],
[0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0],
[0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0],
[0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0],
[0,1,1,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0],
[0,1,1,1,1,1,1,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,0],
[0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0],
[0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,0],
[0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0],
[0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0],
[0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0],
[0,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0],
[0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,0],
[0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0],
[0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0],
[0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0],
[0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0],
[0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0],
[0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0],
[0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0],
[0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,0],
[0,0,0,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,1,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
```

```python
# option: GBFS, FW
ALGORITHM = "GBFS"
if __name__ == "__main__":
    # pygame setup
    pygame.init()
    # screen = pygame.display.set_mode((600, 599)) # for maze.png
    screen = pygame.display.set_mode((1821, 821)) # for maze_new.png -> 1920 x 1080
    clock = pygame.time.Clock()

    # game setup
    bg_surf = pygame.image.load('maze.png').convert()
    pathfinder = Pathfinder(get_matrix())

    # algorithm
    # 1. init board
    # 2. set START point
    # 3. set END point
    # 4. find path with algorithm
    # 5. draw path
    # 6. reset upon next mouse down event for start point

    is_start_point = True
    start_x, start_y = 0, 0

    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            if event.type == pygame.MOUSEBUTTONDOWN:
                # start_x, start_y = 0, 0
                if is_start_point:
                    print("setting start point..")
                    start_x, start_y = pathfinder.get_coordinate()
                    # prepare for END point
                    is_start_point = False
                else:
                    print("setting end point..")
                    end_x, end_y = pathfinder.get_coordinate()
                    # prepare for START point again
                    is_start_point = True
                    pathfinder.create_path(start_x, start_y, end_x, end_y, ALGORITHM)

        screen.blit(bg_surf, (0, 0))
        pathfinder.update()
        pygame.display.update()
```

C. Floyd-Warshall Code

```python
def is_valid(x, y, matrix):
    rows, cols = len(matrix), len(matrix[0])
    return 0 <= y < rows and 0 <= x < cols and matrix[y][x] == 1

def floyd_warshall(matrix):
    INF = float('inf')

    num_vertices = 0
    index_to_coords = {}
    coords_to_index = {}
    rows, cols = len(matrix), len(matrix[0])

    for y in range(rows):
        for x in range(cols):
            if matrix[y][x] == 1:
                index_to_coords[num_vertices] = (y, x)
                coords_to_index[(y, x)] = num_vertices
                num_vertices += 1

    dist_matrix = [[INF for _ in range(num_vertices)] for _ in range(num_vertices)]
    pred_matrix = [[None for _ in range(num_vertices)] for _ in range(num_vertices)]
    for i in range(num_vertices):
        dist_matrix[i][i] = 0

    for node, (y, x) in index_to_coords.items():
        neighbors = [(y - 1, x), (y + 1, x), (y, x - 1), (y, x + 1)]
        for ny, nx in neighbors:
            if (ny, nx) in coords_to_index:
                neighbor_node = coords_to_index[(ny, nx)]
                dist_matrix[node][neighbor_node] = 1
                pred_matrix[node][neighbor_node] = node

    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                if dist_matrix[i][k] + dist_matrix[k][j] < dist_matrix[i][j]:
                    dist_matrix[i][j] = dist_matrix[i][k] + dist_matrix[k][j]
                    pred_matrix[i][j] = pred_matrix[k][j]
```

```python
def reconstruct_path(start, end, pred_matrix, coords_to_index):
    start_index = coords_to_index[start]
    end_index = coords_to_index[end]

    path = []
    current = end_index
    while current is not None:
        path.append(current)
        current = pred_matrix[start_index][current]

    path = path[::-1]  # Reverse the path to start-to-goal order

    return path
```