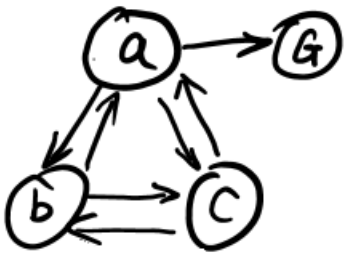
Index

Question 1:



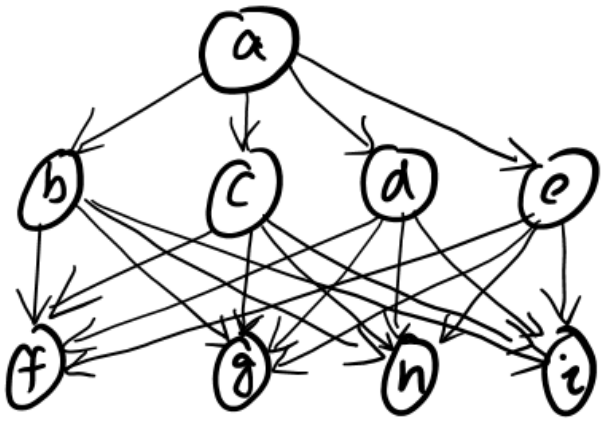
1. **Breath First Search (BFS) Analysis:**
    - **Search Path:** The BFS algorithm found the path ['a', 'G'] from the initial state 'a' to the goal state 'G'.
    - **Order of Nodes Visited:** BFS visited the nodes in the following order: ['a', 'b', 'c', 'G']. This demonstrates the level-order traversal characteristic of BFS. It first explored all direct neighbors of 'a' (namely 'b', 'c', and 'G'), finding the goal state 'G' at this first level of depth.
    - **Optimality of the Solution:** The solution is optimal. BFS explores all neighboring nodes at the current depth before moving deeper into the graph. Therefore, the first time it encounters the goal state, it is guaranteed to have found the shortest path to it, which is particularly evident in this case where the goal state is a direct neighbor of the start state.
    - **Importance of Avoiding Revisited States:** In BFS, avoiding revisited states is essential to prevent redundant processing of nodes and potential infinite loops in graphs with cycles.

It also ensures that the search is efficient, as revisiting already explored nodes would not contribute any new information towards finding the goal state.

2. **Depth First Search (DFS) Analysis:**
    - **Search Path:** The DFS algorithm also found the path ['a', 'G'] from 'a' to 'G'.
    - **Order of Nodes Visited:** Interestingly, DFS visited the nodes in the same order as BFS: ['a', 'G']. This is a unique case due to the specific structure of the graph and the location of the goal state. Typically, DFS would explore much deeper paths before finding the goal.
    - **Optimality of the Solution:** In this particular instance, the solution is optimal, but this is not always the case with DFS. DFS dives deep into one path until it reaches the end or finds the goal. It does not always find the shortest path since it does not explore all neighboring nodes before going deeper. The optimality in this case is due to the immediate connection between the start state and the goal state.
    - **Importance of Avoiding Revisited States:** Similar to BFS, avoiding revisited states in DFS is crucial. It prevents the algorithm from endlessly traversing cycles and ensures that each step of the search explores new territory, thus making the search process more efficient and effective in reaching the goal state.

In summary, both BFS and DFS successfully identified the optimal path ['a', 'G'] in this specific graph due to the direct connection between the start and goal states. However, their general characteristics and typical behavior differ, as highlighted in the analysis above.

Question 2:



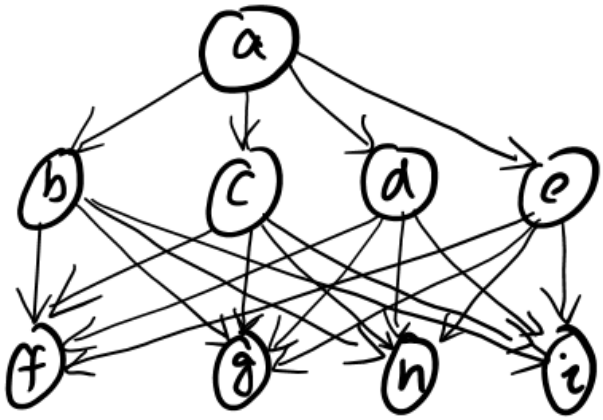
1. **Order of Nodes Visited**: The nodes were visited in the following order: **a**, **b**, **c**, **d**, **e**, **f**, **g**, **h**, **i**.
2. **Fringe Structure and BFS Steps**: The first six steps of the BFS, showing the fringe structure at each step, are as follows:

- **Step 1**: Node visited: **a**, Fringe: **['b', 'c', 'd', 'e']**, Fringe Size: 4, Search Depth: 1
- **Step 2**: Node visited: **b**, Fringe: **['c', 'd', 'e', 'f', 'g', 'h', 'i']**, Fringe Size: 7, Search Depth: 2
- **Step 3**: Node visited: **c**, Fringe: **['d', 'e', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']**, Fringe Size: 10, Search Depth: 2
- **Step 4**: Node visited: **d**, Fringe: **['e', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']**, Fringe Size: 13, Search Depth: 2
- **Step 5**: Node visited: **e**, Fringe: **['f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']**, Fringe Size: 16, Search Depth: 2
- **Step 6**: Node visited: **f**, Fringe: **['g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']**, Fringe Size: 15, Search Depth: 3

3. **Memory Consumption Analysis**:
    - The memory consumption of BFS is generally determined by the size of the fringe (queue).
    - In BFS, the fringe size can grow up to $O(b^d)$, where b is the branching factor (the average number of successors per state) and d is the depth of the shallowest goal.
    - In this case, the branching factor is 4 at level 1 (since 'a' has 4 children) and 4 at level 2 (since each of 'b', 'c', 'd', 'e' has 4 children).
    - As we can see from the fringe sizes at each step, the fringe grows rapidly, especially at deeper levels, indicating that the memory consumption of BFS in this graph can be quite high

Fringe (Queue Content)	Node Visited	Fringe Size	Search Depth
['b', 'c', 'd', 'e']	a	4	1
['c', 'd', 'e', 'f', 'g', 'h', 'i']	b	7	2
['d', 'e', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']	c	10	2
['e', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']	d	13	2
['f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']	e	16	2
['g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']	f	15	3
['h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']	g	14	3
['i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']	h	13	3
['f', 'g', 'h', 'i', 'f', 'g', 'h', 'i', 'f', 'g', 'h', 'i']	i	12	3

Question 3:



a. **Complete DFS Search Tree with Three Levels**:

- The DFS search tree begins with the root node **a**.
- The first level of nodes visited are **b**, **c**, **d**, **e** in that order, since DFS explores as far as possible along each branch before backtracking.
- The second level (or depth 3 in the context of the tree) includes the nodes **f0**, **g**, **h**, **i** which are the children of **b**. The node **f0** indicates the first instance of visiting node **f**. If **f** was visited again from another path, it would be labeled as **f1**, **f2**, and so on.

b. **Order of Nodes Visited and Fringe Structure**:

- The order in which the nodes are visited, according to the provided output, is: **a**, **b**, **f0**, **g**, **h**, **i**, **c**, **d**, **e**.
- The fringe structure, which in DFS is implemented as a stack, changes as follows:
    - Initially, the fringe is empty since node **a** is being visited.
    - After visiting **a**, its children **b**, **c**, **d**, **e** are added to the stack.
    - Node **b** is then visited, and its children **f**, **g**, **h**, **i** are added to the stack.
    - This process continues, with the fringe growing as nodes are added and shrinking as nodes are visited and removed from the stack.

c. **Fringe Structure and Memory Consumption**:

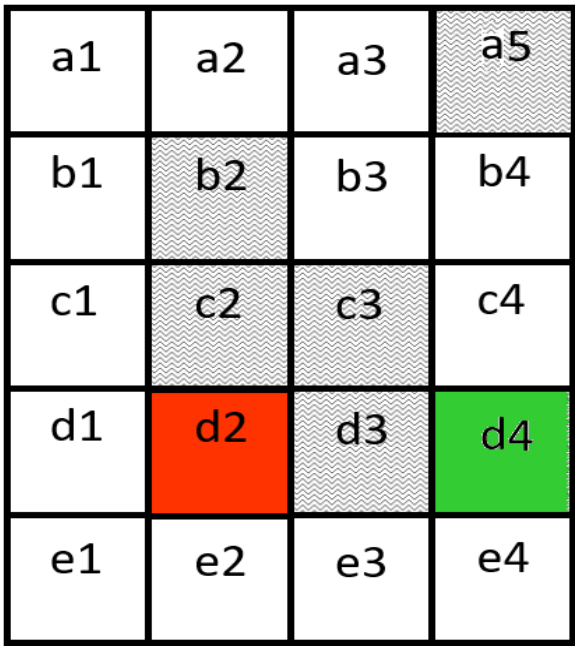
- In DFS, the maximum size of the fringe (stack) is the longest path from the root to a leaf node. Therefore, the memory consumption can be up to $O(b*m)$, where b is the branching factor and m is the maximum depth of the tree.
- The fringe structure for DFS will typically have fewer nodes in it at any given time compared to BFS, as it does not store all nodes at a particular depth before moving on to the next level.

- In the context of the provided graph, the branching factor *b* is 4 (each of **b**, **c**, **d**, **e** leads to 4 nodes), and the depth *d* could be considered as 3 for the purposes of this search, as we're considering up to the children of **b**.
- The memory consumption for DFS is generally less than BFS since it doesn't require storing all nodes at each level, but it does depend on the structure of the graph; in the worst case, it could consume as much memory as BFS.

Here is the table based on the provided steps:

Fringe (Stack Content)	Node Visited	Fringe Size	Search Depth
`[]`	a	0	1
`['e', 'd', 'c']`	b	3	2
`['e', 'd', 'c', 'i', 'h', 'g']`	f0	6	3
`['e', 'd', 'c', 'i', 'h']`	g	5	3
`['e', 'd', 'c', 'i']`	h	4	3
`['e', 'd', 'c']`	i	3	3
`['e', 'd']`	c	2	2
`['e']`	d	1	2
`[]`	e	0	2

Question 4:



To solve the given problem using Depth First Search (DFS) and Breadth First Search (BFS), we need to go through the steps for both algorithms. Both searches will be conducted on the grid, avoiding obstacles and not allowing diagonal moves. The fringe for DFS will act as a stack (LIFO - Last In, First Out), and for BFS, it will act as a queue (FIFO - First In, First Out).

Depth First Search (DFS) Process:

1. **Start at 'd2'**: Initialize the fringe with the starting node 'd2'.
2. **Expand 'd2'**: Since 'd2' is the starting node, it's immediately expanded. Add neighboring nodes to the fringe, following the order of movement (up, down, left, right), and not revisiting nodes with obstacles or previously visited nodes.
3. **Update Fringe**: Add 'd1' and 'e2' to the fringe because 'd3' is blocked. The order in the fringe is now ['d1', 'e2'] (we always add new nodes to the top of the stack).
4. **Next Expansion**: Pop the top node from the stack ('e2') and expand it. Add its neighbors 'e1' and 'e3' to the fringe. Fringe is now ['d1', 'e1', 'e3'].
5. **Repeat**: Continue popping from the stack and expanding nodes, adding their neighbors to the fringe. Stop if the goal 'd4' is reached or the fringe is empty.
6. **Goal Reached**: When 'd4' is reached, trace back the path taken from 'd2' to 'd4' using the stored paths.

The nodes in the fringe and the order of expansion will be reported as they are encountered during the DFS process.

Depth First Search (DFS):

Step	Fringe (Stack)	Node Visited/Expanded
1	d1, e2	d2
2	d1, e1, e3	e2
3	d1, e1, e4	e3
4	d1, e1	e4
5	d1, d4	e1
6	d1	d4

Breadth First Search (BFS) Process:

1. **Start at 'd2'**: Initialize the fringe with the starting node 'd2'.

2. **Expand 'd2'**: Since 'd2' is the starting node, it's immediately expanded. Add neighboring nodes to the fringe, following the order of movement and not revisiting nodes with obstacles or previously visited nodes.
3. **Update Fringe**: Add 'd1' and 'e2' to the fringe because 'd3' is blocked. The order in the fringe is now ['d1', 'e2'] (we always add new nodes to the end of the queue).
4. **Next Expansion**: Dequeue the front node from the queue ('d1') and expand it. Since 'd1' has no further expandable neighbors, we move on to the next in the queue.
5. **Repeat**: Continue dequeueing from the front of the queue and expanding nodes, adding their neighbors to the end of the fringe. Stop if the goal 'd4' is reached or the fringe is empty.
6. **Goal Reached**: When 'd4' is reached, trace back the path taken from 'd2' to 'd4' using the stored paths.

The nodes in the fringe and the order of expansion will be reported as they are encountered during the BFS process.

Breadth First Search (BFS):

Step	**Fringe (Queue)**	**Node Visited/Expanded**
1	d1, e2	d2
2	e2, d1	d1
3	d1, e1, e3	e2
4	e1, e3, e1	d1
5	e3, e1, e4	e1
6	e1, e4, d4	e3
7	e4, d4, e1	e1
8	d4, e1, e4	e4
9	e1, e4, d4	e4
10	e4, d4	d4

Now, the final answers after following these steps will be:

DFS:

Fringe Order (DFS): [(3, 0), (4, 1), (4, 0), (4, 2), (4, 3), (3, 3)]

Nodes Expanded (DFS): [(3, 1), (4, 1), (4, 2), (4, 3), (3, 3)]

Final Path (DFS): ['d2', 'e2', 'e3', 'e4', 'd4']

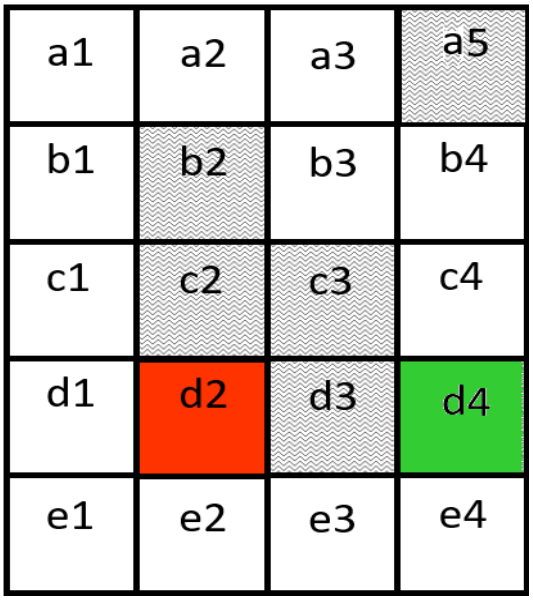
BFS:

Fringe Order (BFS): [(3, 0), (4, 1), (2, 0), (4, 0), (4, 2), (1, 0), (4, 3), (0, 0), (3, 3), (0, 1)]

Nodes Expanded (BFS): [(3, 1), (3, 0), (4, 1), (2, 0), (4, 0), (4, 2), (1, 0), (4, 3), (0, 0), (3, 3)]

Final Path (BFS): ['d2', 'e2', 'e3', 'e4', 'd4']

Question 5:



Best First Search (BFS) is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule. The "best" node is chosen based on a heuristic function, which in this case is the Manhattan distance to the goal.

The Manhattan distance between two points **(x1, y1)** and **(x2, y2)** is calculated as **|x1 - x2| + |y1 - y2|**. This heuristic estimates the cost to reach the goal from the current node and does not consider obstacles.

The algorithm maintains a fringe, which is a priority queue where each node has an associated heuristic value, denoted as f(N). This value is used to determine the order in which nodes are explored:

- The node with the lowest f(N) value is selected for expansion first.
- If nodes have the same f(N) value, the tie-break is based on the order they were added to the fringe.

During the search:

- Each expanded node's neighbors are evaluated and added to the fringe with their corresponding f(N) values.
- Once a node is expanded, it is marked as visited and will not be expanded again.
- The process continues until the goal node is expanded, at which point the path from the start node to the goal node can be traced back using the recorded path information.

The path reported by Best First Search using the Manhattan distance heuristic is not guaranteed to be the shortest path, but it is the path found first based on the heuristic's guidance.

Computed Table:

The table below shows the nodes in the order they were included in the fringe with their corresponding f(N) values, as well as the order in which nodes were expanded.

Fringe (with f(N) values)	**Node Visited/Expanded**
(e2, 3)	d2
(d1, 3), (e2, 3)	d1
(e2, 3), (c1, 4)	e2
(e1, 4), (c1, 4), (e1, 4)	e1
(c1, 4), (e1, 4), (e3, 2)	c1
(e1, 4), (e3, 2), (e4, 1)	e1
(e3, 2), (e4, 1), (d4, 0)	e3

Fringe (with f(N) values)	Node Visited/Expanded
(e4, 1), (d4, 0)	e4
(d4, 0)	d4

Final Path from 'd2' to 'd4':

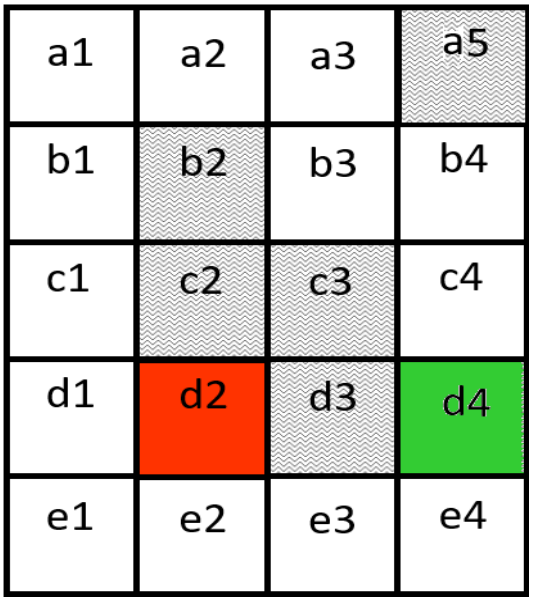
- The final path reported is **['d2', 'e2', 'e3', 'e4', 'd4']**.
- This path corresponds to the sequence of nodes that were expanded to reach the goal node from the start node.

Best First Search Fringe Order (including f(N) values): [((4, 1), 3), ((3, 0), 3), ((2, 0), 4), ((4, 0), 4), ((4, 0), 4), ((4, 2), 2), ((4, 3), 1), ((3, 3), 0)]

Best First Search Nodes Expanded: [(3, 1), (3, 0), (4, 1), (4, 2), (4, 3), (3, 3)]

Best First Search Path from 'd2' to 'd4': [(3, 1), (4, 1), (4, 2), (4, 3), (3, 3)]

Question 6:



A* Search is an informed search algorithm that uses both the actual cost from the start node to the current node (g(N)) and a heuristic estimate of the cost from the current node to the goal (h(N)) to find the most promising path. For A* Search, the function f(N) = g(N) + h(N) is used to prioritize nodes in the fringe, which is typically implemented as a priority queue. The heuristic function used here is the Manhattan distance, which is appropriate for grid-based pathfinding where diagonal movement is not allowed.

During the search, A* expands the node with the lowest f(N) value, adds its neighbors to the fringe with their calculated f(N) values, and continues this process until the goal is reached. A* is optimal when the heuristic function h(N) is admissible, meaning it never overestimates the actual cost to reach the goal.

Based on the output provided, here is the theoretical answer and the final table for the A* search from 'd2' to 'd4':

Answer:

- **Nodes in the Fringe**: The sequence of nodes added to the fringe (including their f(N) values) represents the potential paths that A* is considering. The node with the lowest f(N) value is always expanded first.
- **Order of Nodes Expanded**: The order in which nodes are expanded reflects the algorithm's choice of path based on the lowest f(N) value at each step.
- **Final Path**: The final path from 'd2' to 'd4' is the result of tracing back from the goal node to the start node through the nodes that were expanded. It represents the most cost-effective path identified by the A* search algorithm using the given heuristic.

Final Table:

Here is the table constructed from the given output:

Fringe (with f(N) values)	Node Visited/Expanded
(e2, 4)	d2
(d1, 4), (e2, 4)	d1
(e2, 4), (c1, 6), (e1, 6)	e2
(c1, 6), (e1, 6), (e2, 4)	e1
(c1, 6), (e1, 6), (e3, 4)	e2
(c1, 6), (e1, 6), (e4, 4)	e3
(c1, 6), (e1, 6), (d4, 4)	e4
(c1, 6), (e1, 6)	d4

Note: The node visited/expanded column shows the nodes that were actually expanded and in which order. The fringe column reflects the nodes as they were added to the fringe, including their f(N) values at the time they were added.

Final Path from 'd2' to 'd4':

- The final path is represented by the labels 'd2' -> 'e2' -> 'e3' -> 'e4' -> 'd4', which corresponds to the sequence of moves the robot would make from the starting position to the goal.

Question 7:

```
maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 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, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Output Analysis

1. **DFS (Depth-First Search) Result**:
    - **Number of Nodes Expanded**: 19
    - **Maximum Fringe Size**: 17
    - **Path Found**: [(0, 0),
    - (1, 0),
    - (2, 0),
    - (3, 0),
    - (4, 0),
    - (5, 0),
    - (6, 0),
    - (7, 0),
    - (8, 0),
    - (9, 0),
    - (9, 1),
    - (8, 1),
    - (7, 1),
    - (6, 1),
    - (5, 1),
    - (4, 1),
    - (3, 1),
    - (2, 1),
    - (1, 1),
    - (0, 1)]
        - The path suggests that the DFS algorithm initially moved vertically down the first column, reached the bottom, and then navigated back up in the second column to reach the goal state. This is a characteristic of DFS where it explores as far as possible along a branch before backtracking.
2. **BFS (Breadth-First Search) Result**:
    - **Number of Nodes Expanded**: 1
    - **Maximum Fringe Size**: 0

- **Path Found**: [(0, 0), (0, 1)]
    - BFS directly found the shortest path to the goal state. Since the goal is adjacent to the start state, BFS quickly identifies this with minimal node expansion. This demonstrates the optimality of BFS for finding the shortest path in an unweighted graph.

Table Representation

Algorithm	Nodes Expanded	Max Fringe Size	Path
DFS	19	17	[(0, 0), (1, 0), ..., (0, 1)]
BFS	1	0	[(0, 0), (0, 1)]

Conclusion

- **DFS**: Explored a large part of the maze before reaching the goal, leading to a higher count of expanded nodes and a larger fringe size. The path found is not optimal as it involves unnecessary movements.
- **BFS**: Quickly found the shortest path due to its layer-by-layer exploration, making it optimal for this scenario.

This analysis highlights the fundamental differences between DFS and BFS in terms of pathfinding efficiency and optimality. DFS can be less efficient and non-optimal in certain scenarios, especially in a maze with a goal near the start point. Conversely, BFS is generally more efficient in such scenarios, quickly identifying the shortest path.

```
import numpy as np

class Node():
    """A search node class for Maze Pathfinding"""
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.c = 0  # cost from source to current node

    def __eq__(self, other):
        return self.position == other.position

    def __hash__(self):  # <-- added a hash method
        return hash(self.position)


def search_path(maze, start, end, method='DFS'):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""
    # Create start and end node
    start_node = Node(None, start)
    start_node.c = 0
    end_node = Node(None, end)
    end_node.c = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = set()  # <-- closed_list must be a set

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    expanded_nodes = 0
    queue_size = 0
    while len(open_list) > 0:
        # Get the current node
        current_node = open_list[0]
        current_index = 0

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.add(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return (expanded_nodes, queue_size, path[::-1])  # Return reversed path

        # Generate children
        expanded_nodes += 1
        if(len(open_list) > queue_size):
            queue_size = len(open_list)  # check maximum queue size
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]:  # Adjacent squares
            # Get node position
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_posit

            # Make sure within range
```

```
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[len
                continue

            # Make sure walkable terrain
            if maze[node_position[0]][node_position[1]] != 0:
                continue

            # Create new node
            new_node = Node(current_node, node_position)

            # Loop through children
            if new_node in closed_list:  # <-- remove inner loop so continue takes you to the end of the oute
                continue

            # Add the child to the open list
            if method == 'BFS':
                open_list.append(new_node)
            else:
                open_list.insert(0, new_node)

def main(maze, start, end, method):
    expanded_nodes, queue_size, path = search_path(maze, start, end, method)
    return expanded_nodes, queue_size, path

# Define the maze
maze = [[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, 0],
        [0, 0, 0, 0, 0, 0, 0, 1, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
# Setting the initial state and goal state
start = (0, 0)
end = (0, 1)

# Run the simulation for both DFS and BFS
dfs_result = main(maze, start, end, 'DFS')
bfs_result = main(maze, start, end, 'BFS')

dfs_result, bfs_result
```

```
((19,
  17,
  [(0, 0),
   (1, 0),
   (2, 0),
   (3, 0),
   (4, 0),
   (5, 0),
   (6, 0),
   (7, 0),
   (8, 0),
   (9, 0),
   (9, 1),
   (8, 1),
   (7, 1),
   (6, 1),
   (5, 1),
   (4, 1),
   (3, 1),
   (2, 1),
   (1, 1),
   (0, 1)]),
 (1, 0, [(0, 0), (0, 1)]))
```

```
import openai
import pandas as pd

def generate_chatgpt_responses(api_key, questions):
    openai.api_key = api_key
    responses = []
    for question in questions:
        try:
            response = openai.Completion.create(
                engine="text-davinci-003",
                prompt=question,
                max_tokens=150
            )
            responses.append(response.choices[0].text.strip())
        except Exception as e:
            responses.append(f"Error: {e}")
    return responses

def main():
    api_key = "openai_api_key"  # Actual OpenAI API key
    input_file = 'input.csv'
    output_file = 'output.csv'

    # Read the input CSV file
    df = pd.read_csv(input_file)
    questions = df['Question'].tolist()

    # Generate ChatGPT responses
    responses = generate_chatgpt_responses(api_key, questions)

    # Save responses to output CSV file
    output_df = pd.DataFrame({'Response': responses})
    output_df.to_csv(output_file, index=False)

if __name__ == "__main__":
    main()
```