CS7050 Artificial Intelligence

Submitted to: Prof Vassil Vassilev

Submitted by
Muhammad Sajjad Hussain-24014153
Muhammad Usama Fiaz-23029706
Muhammad Umer-23035882
Usama Hussain-23030627

Problem Solving using State Space Search *and Knowledge-based Inference*

December 2, 2024

London Metropolitan University ,
London

Contents

Maze Solving by State Space Search and Knowledge-based Inference

1 Maze Description

- **Maze Dimensions**: 5 rows and 6 columns, so it's a grid of size 5x6.
- **Start Node**: (0, 0)
- **End Node**: (4, 5)
- **Blocked Nodes**: Represented as obstacles (red nodes), which prevent movement.
- **Permitted Moves**: Up, Down, Left, Right.

2 Maze Validation

We use BFS for solving this problem .First we check the maze is validation for searching the goal start and end both points are in the range of maze or not .Approach using BFS, a breadth-first search algorithm that explores all possible paths level by level to ensure finding the shortest path in an unweighted graph like this Maze.BFS is a blind search method that explores all possible moves in a layer-wise fashion.

```python
import matplotlib.pyplot as plt
import numpy as np
from collections import deque
import time

# Define the maze as a grid
maze = [
    [0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0, 0],
    [0, 1, 0, 0, 1, 0],
    [0, 0, 0, 0, 1, 0]
]  # 1 means obstacles, and 0 means paths in the maze

# Define possible permitted movements: up, down, left, right
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Define the start and goal positions
start = (0, 0)
goal = (4, 5)


def validate_maze(maze, start, goal):
    # Check if maze is a rectangle
    row_length = len(maze[0])
    for row in maze:
        if len(row) != row_length:
            print("Invalid maze: All rows must have the same length.")
            return False

    # Check if start and goal are within the maze boundaries
    rows, cols = len(maze), len(maze[0])
    if not (0 <= start[0] < rows and 0 <= start[1] < cols):
        print("Invalid start position: Out of maze bounds.")
        return False
```

```
    if not (0 <= goal[0] < rows and 0 <= goal[1] < cols):
        print("Invalid goal position: Out of maze bounds.")
        return False

    # Check if start and goal are on valid paths (0, not 1)
    if maze[start[0]][start[1]] == 1:
        print("Invalid start position: Start is on an obstacle.")
        return False
    if maze[goal[0]][goal[1]] == 1:
        print("Invalid goal position: Goal is on an obstacle.")
        return False

    # Check for valid maze contents (only 0 for paths and 1 for obstacles)
    for row in maze:
        for cell in row:
            if cell not in [0, 1]:
                print("Invalid maze content: Maze should contain only 0 (paths)
 and 1 (obstacles).")
                return False

    return True

# Example usage:
if validate_maze(maze, start, goal):
    print("Maze validation successful.")
else:
    print("Maze validation failed.")
```

Listing 1: Maze Validation Code

3 Breadth First Search

Breadth-First Search (BFS) algorithm for finding all possible paths and their associated costs from a start point to a goal in a maze represented as a grid.

```
import matplotlib.pyplot as plt
import numpy as np
from collections import deque
import time  # Import the time module

# Define the maze as a grid
maze = [
    [0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0, 0],
    [0, 1, 0, 0, 1, 0],
    [0, 0, 0, 0, 1, 0]
]  # 1 means obstacles, and 0 means paths

# Define start and goal positions
start = (0, 0)
goal = (4, 5)

# Directions for moving: up, down, left, right
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Function to check if a position is valid within the maze
def is_valid(x, y, maze):
    rows, cols = len(maze), len(maze[0])
    return 0 <= x < rows and 0 <= y < cols and maze[x][y] == 0  # 0 represents
    valid paths

```

```python
# Breadth-First Search algorithm to find all paths and their costs
def bfs_all_paths_with_cost(maze, start, goal):
    queue = deque([([start], start)])  # Queue stores (path, current position)
    all_paths = []  # Store all paths found to the goal with costs

    while queue:
        path, (x, y) = queue.popleft()

        # Check if we have reached the goal
        if (x, y) == goal:
            # Calculate the cost of this path (number of steps)
            cost = len(path) - 1  # Exclude the start point from the cost
            all_paths.append((path, cost))  # Save this path with its cost
            continue  # Continue to explore other paths

        # Explore all possible directions (up, down, left, right)
        for dx, dy in directions:
            newx, newy = x + dx, y + dy
            if is_valid(newx, newy, maze) and (newx, newy) not in path:  # Ensure we don’t revisit nodes in this path
                new_path = path + [(newx, newy)]
                queue.append((new_path, (newx, newy)))

    return all_paths

# Function to measure the time of the given algorithm
def measure_time(algorithm, *args):
    start_time = time.time()  # Record the start time
    result = algorithm(*args)  # Run the algorithm with the given arguments
    end_time = time.time()  # Record the end time
    elapsed_time = end_time - start_time  # Calculate the elapsed time
    print(f"Execution time: {elapsed_time:.4f} seconds")
    return result

# Function to visualize a single path
def visualize_path(maze, path, title=""):
    # Create a color map for visualization
    rows, cols = len(maze), len(maze[0])
    grid = np.zeros((rows, cols, 3))  # RGB grid initialized to black

    # Set background to light green
    background_color = [0.9, 1, 0.9]  # Light green
    grid[:, :] = background_color

    # Set obstacles to light gray
    obstacle_color = [0.8, 0.8, 0.8]  # Light gray for obstacles
    for i in range(rows):
        for j in range(cols):
            if maze[i][j] == 1:  # 1 now represents obstacles
                grid[i, j] = obstacle_color

    # Set path to light blue
    path_color = [0.6, 0.8, 1]  # Light blue for path
    for (x, y) in path:
        grid[x, y] = path_color  # Assign light blue color to the path cells

    # Plot the grid
    plt.figure(figsize=(6, 6))
    plt.imshow(grid)
    plt.title(title)
    plt.axis("off")
    plt.show()

```

```python
# Run BFS to find all paths and their costs, and measure execution time
all_paths_with_cost = measure_time(bfs_all_paths_with_cost, maze, start, goal)

# Output and visualize all paths
if all_paths_with_cost:
    print("All paths found with costs:")
    for i, (path, cost) in enumerate(all_paths_with_cost, 1):
        print(f"Path {i}: {path}, Cost: {cost}")
        visualize_path(maze, path, title=f"Path {i} - Cost: {cost}")

    # Visualize the final optimal (shortest) path
    optimal_path, optimal_cost = all_paths_with_cost[0]  # BFS guarantees the
    first path is the shortest
    print(f"Optimal Path: {optimal_path}, Cost: {optimal_cost}")
    visualize_path(maze, optimal_path, title=f"Optimal Path (Cost: {
    optimal_cost})")
else:
    print("No path found.")
```

Listing 2: BFS Blind Search Code

3.1 Maze Layout and Basic Setup

The maze is represented as a grid of cells, where each cell can either be open (1) or blocked by an obstacle (0). The starting position of our path is set at the top-left corner of the grid, (0, 0), and the target is in the bottom-right at (4, 5). It allows for movement up, down, left, or right (but not diagonally). Each direction is represented by a change in the x or y coordinates.

3.2 Valid Moves Checker

A function, **is_valid**, ensures that any move stays within the grid boundaries and doesn't try to go through an obstacle that comes in the path . For example, if the algorithm wants to move up but that position is either outside the maze or blocked, **is_valid** function will return False, and that direction will be ignored and try to move in the other direction.

3.3 Finding Paths by using Breadth First Search

3.4 Breadth-First Search (BFS) in Maze Solving

Purpose of BFS:

- BFS is a graph traversal algorithm well-suited for finding the shortest path in an unweighted graph.
- In this maze-solving problem, BFS is chosen because it systematically explores all paths layer by layer, guaranteeing that the shortest path to the goal is found first.

How the Algorithm Works:

- The algorithm uses a queue to manage paths being explored.
- The path taken so far (stored as a list of coordinates).
- The current position in the maze.

Initialization:

- Start from the given start point and add it as the first entry in the queue with an initial path `[start]`.

- If the goal is reached, the path is saved in a list (`all_paths`) along with its cost (number of steps in the path, excluding the starting point).
- The algorithm considers moving up, down, left, and right.
- The move must stay within the maze boundaries.
- The cell must be a walkable path (not an obstacle).
- If valid, extend the path to include this new position and add the updated path back into the queue for further exploration.
- The algorithm continues until all paths have been explored, and every valid path to the goal is saved.

3.5 Timing the Algorithm

A function called measure_time is used to record the execution time. By wrapping the algorithm call in this function, it measures how long the BFS takes to complete. This can be useful to see how efficiently the algorithm performs on different maze sizes or configurations. But most of the time it value changes when the size of maze changes but in our case the maze size is same and so its value changes slowly due to serve and machine speed.

3.6 Visualizing Paths in the Maze

A visualize_path function that uses matplotlib to create a visual map of each path on the maze grid.

- The visualization uses color coding.
- Walkable areas are shown in light green.
- Obstacles are in light gray.
- The specific path being shown is in light blue.

3.7 Generating Output and the Optimal Path

We use BFS Algorithm we find total 42 paths and the cost of ech path is different because BFS try every move in the maze in every direction and find the goal .But most of the maze has same cost .Basically Cost is the steps from Starting point to Goal point in the maze. And in the last we print the optimal path of the maze .Because the optimal path is the path with least cost and the least cost in the maze is 9 . So , In the last the with the minimum cost path print in the last among all of them.

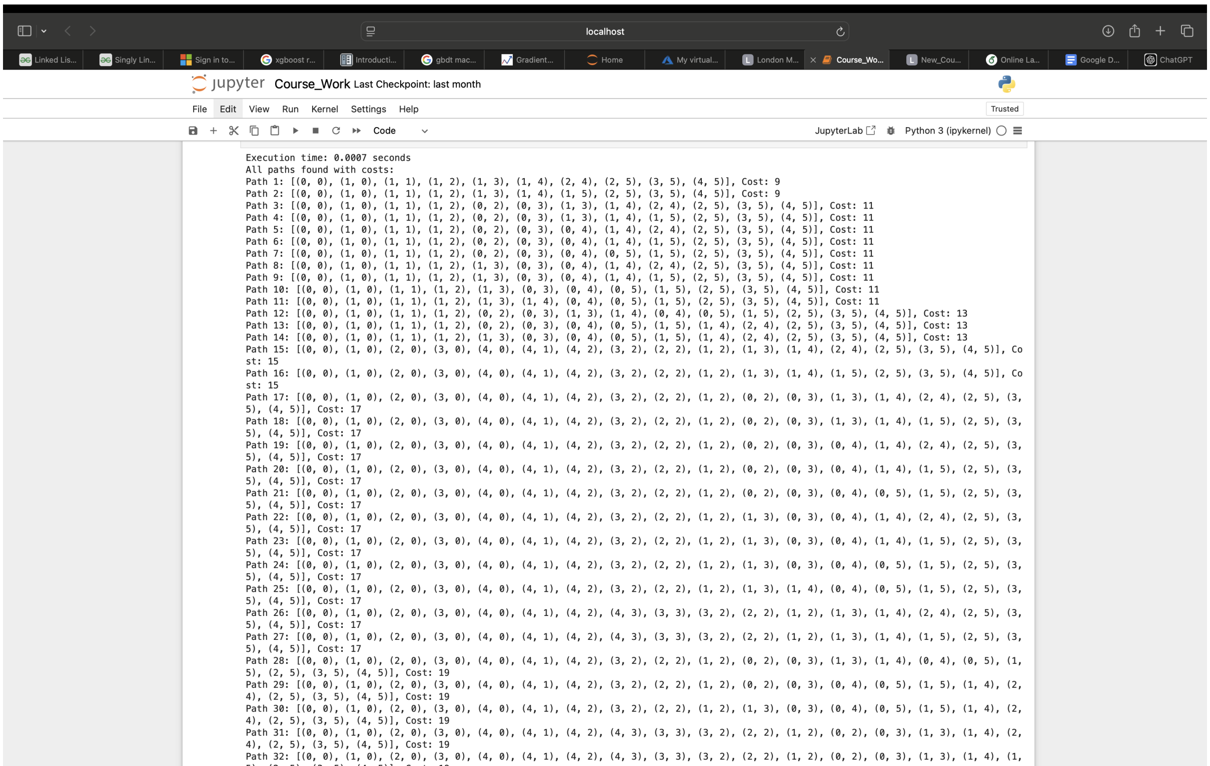


Figure 1: Juypter Notebook Output.

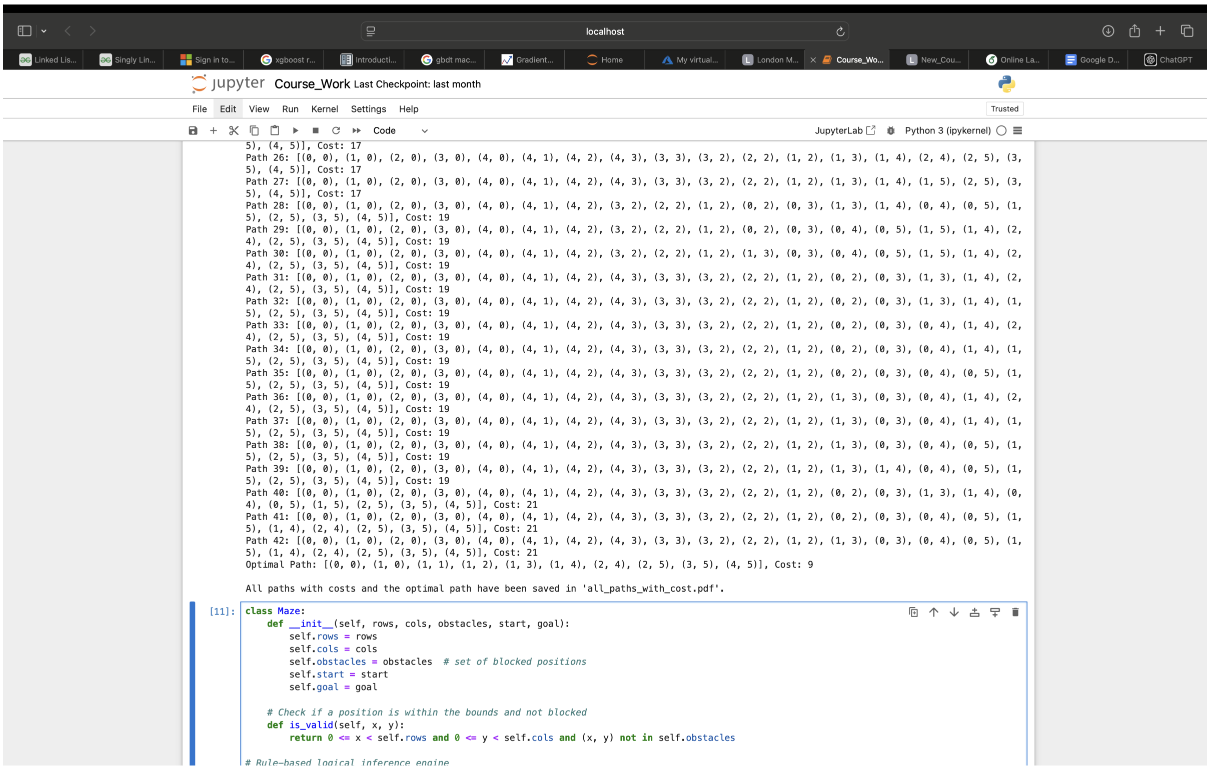
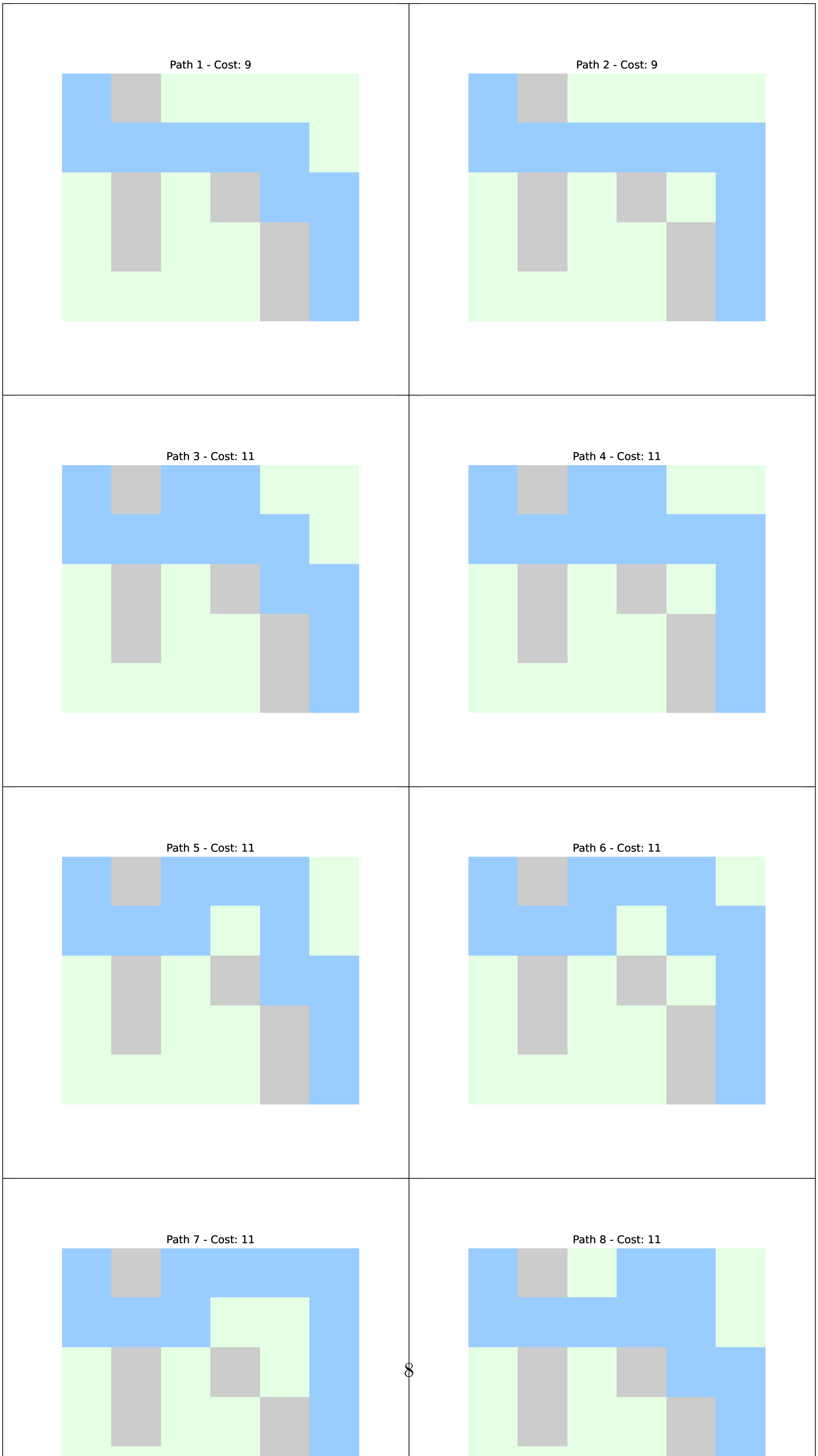
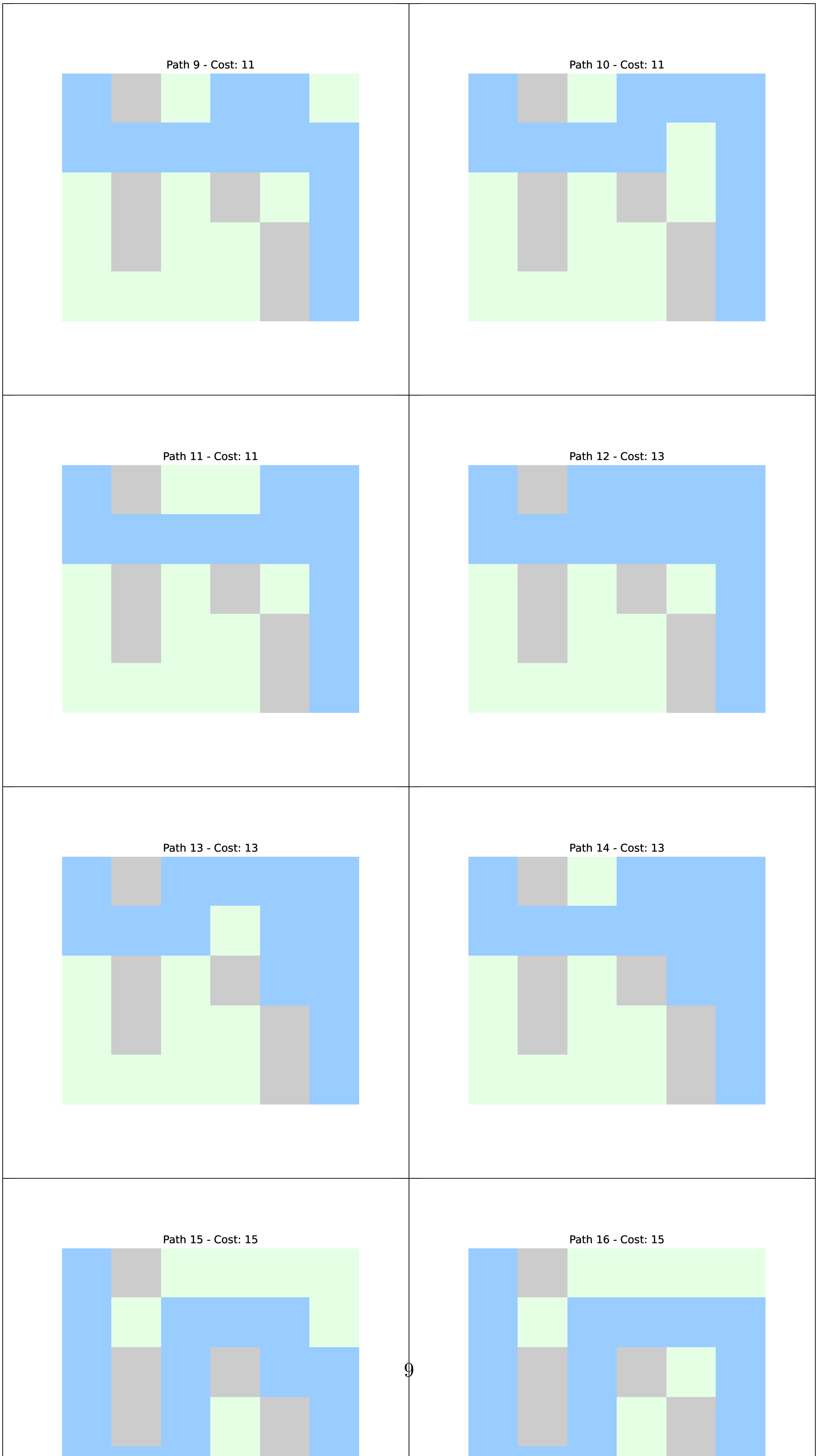
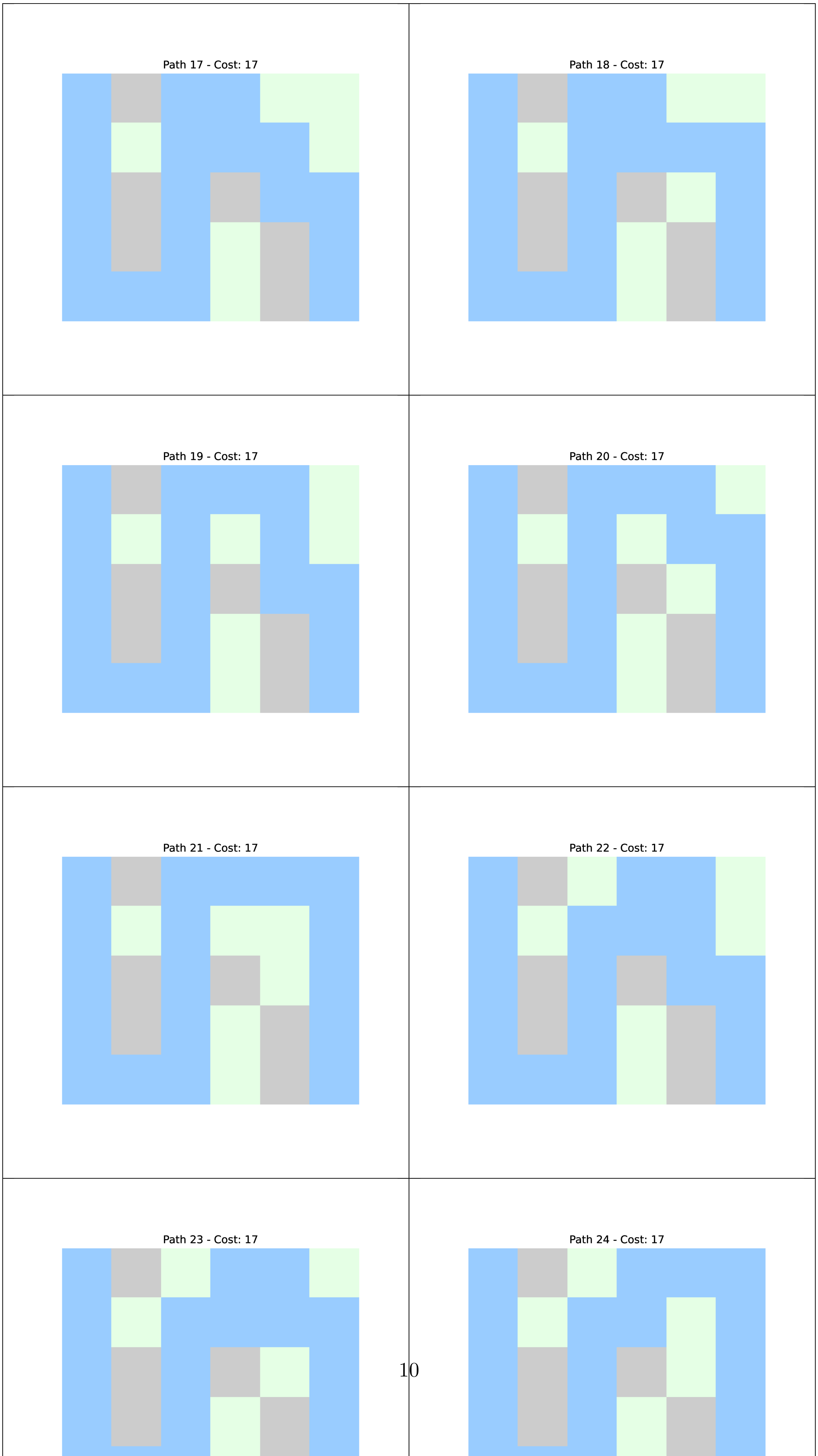


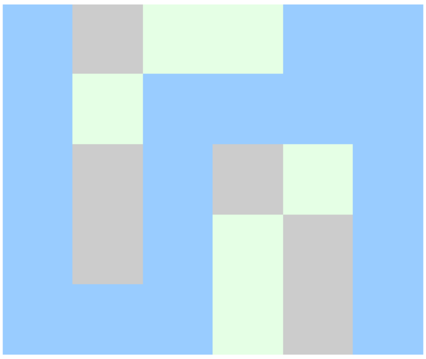
Figure 2: Juypter Notebook Output.


Path 1 - Cost: 9
Path 2 - Cost: 9
Path 3 - Cost: 11
Path 4 - Cost: 11
Path 5 - Cost: 11
Path 6 - Cost: 11
Path 7 - Cost: 11
Path 8 - Cost: 11

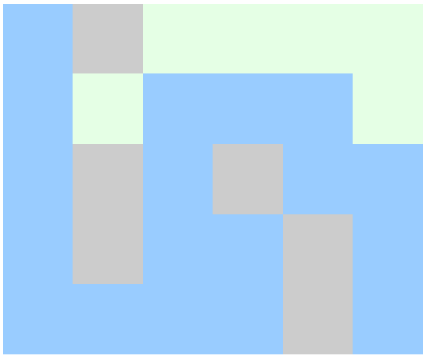

Path 9 - Cost: 11
Path 10 - Cost: 11
Path 11 - Cost: 11
Path 12 - Cost: 13
Path 13 - Cost: 13
Path 14 - Cost: 13
Path 15 - Cost: 15
Path 16 - Cost: 15


Path 17 - Cost: 17
Path 18 - Cost: 17
Path 19 - Cost: 17
Path 20 - Cost: 17
Path 21 - Cost: 17
Path 22 - Cost: 17
Path 23 - Cost: 17
Path 24 - Cost: 17

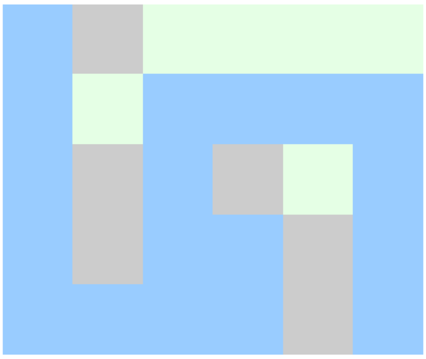
Path 25 - Cost: 17



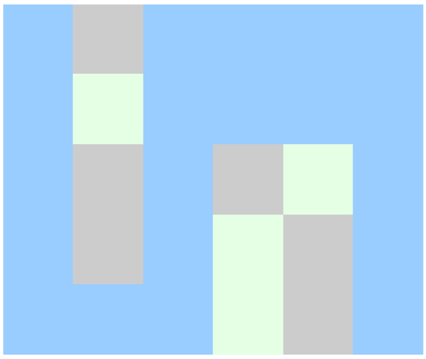
Path 26 - Cost: 17



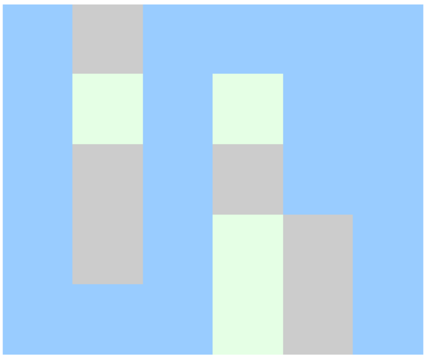
Path 27 - Cost: 17



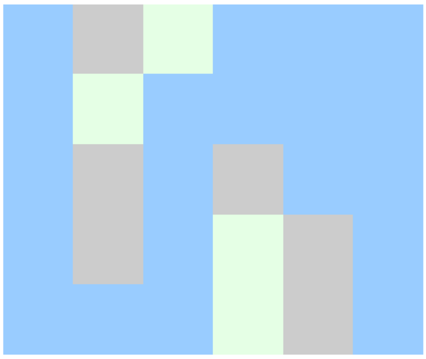
Path 28 - Cost: 19



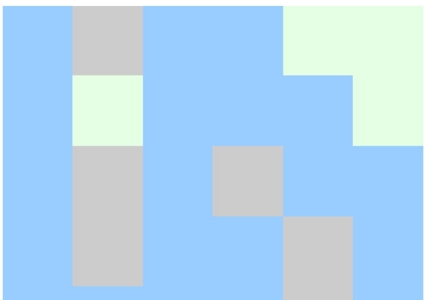
Path 29 - Cost: 19



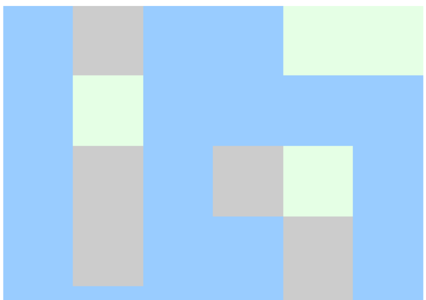
Path 30 - Cost: 19

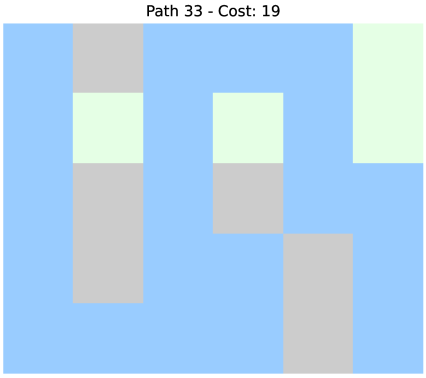


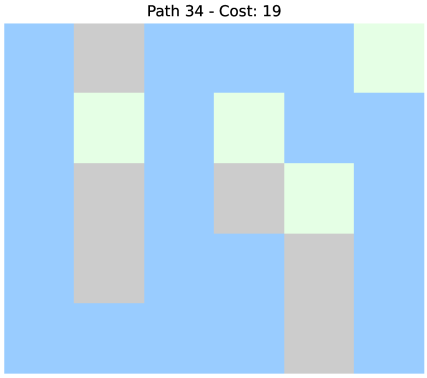
Path 31 - Cost: 19

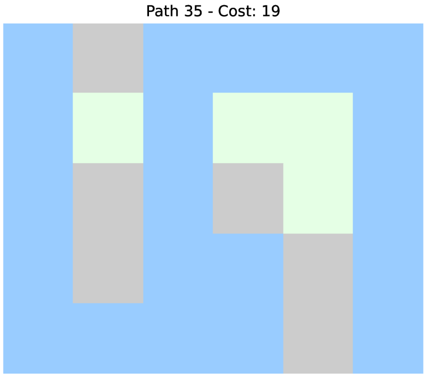


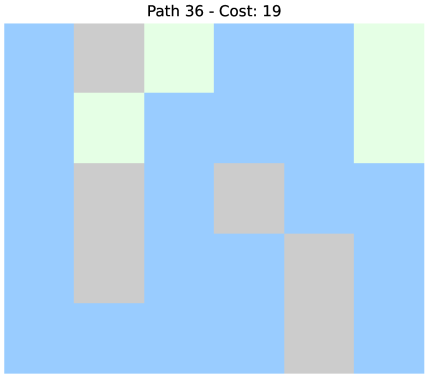
Path 32 - Cost: 19

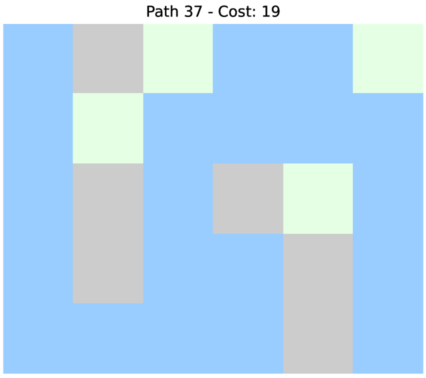


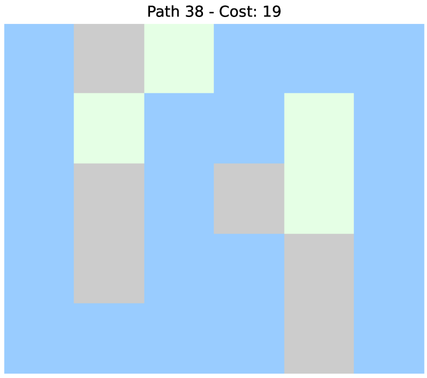

Path 33 - Cost: 19

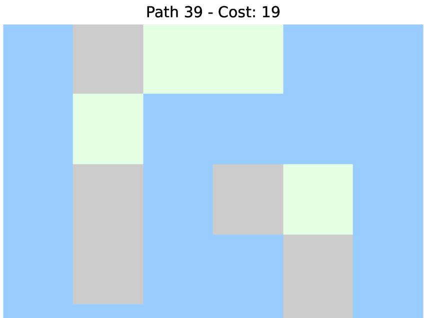

Path 34 - Cost: 19

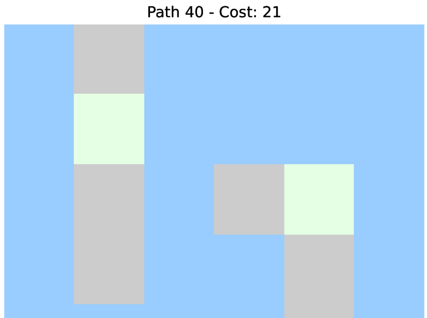

Path 35 - Cost: 19

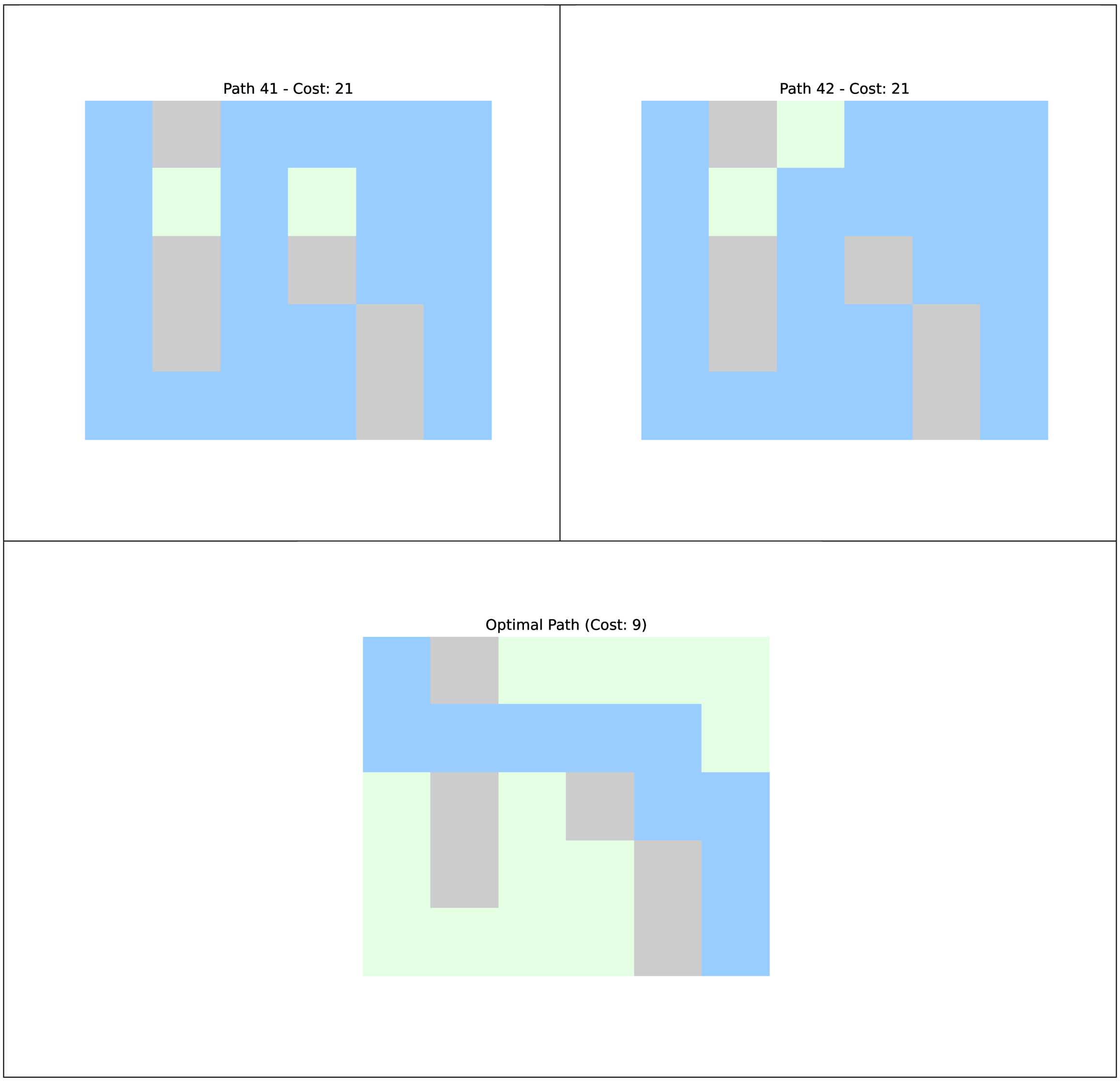

Path 36 - Cost: 19


Path 37 - Cost: 19


Path 38 - Cost: 19


Path 39 - Cost: 19


Path 40 - Cost: 21


Path 41 - Cost: 21
Path 42 - Cost: 21
Optimal Path (Cost: 9)

4 Logical Theory

4.1 Maze Class

- Sets up the maze dimensions and obstacles.
- Checks if a position is valid (within bounds and not blocked).

4.2 LogicalInference Class

- Uses logical rules to determine possible moves (*up, down, left, right*) from any point.
- Includes a `find_path` method that performs a BFS search to explore paths from start to goal.

4.3 Pathfinding

- Starts at the start node and applies move rules to find reachable cells.
- Stops once the goal is reached, returning the path taken.
- If no path is found, it returns `None`.

4.4 Knowledge Representation

- The maze is represented as a set of facts:
  - Allowed moves.
  - Obstacles.
  - Walls.
  - Boundaries.
- Each node in the maze can have predicates indicating:
  - Its position.
  - Whether it is blocked.
  - The available moves.

4.5 Logical Rules

- Rules are defined for moving *up, down, left, or right*.
- These rules check if a move is allowed based on the current state of the agent.
- Includes a rule for reaching the goal (success condition).

4.6 Inference Process

- The inference engine applies rules iteratively.
- Determines a sequence of moves from the start position to the goal.

4.7 Logical Theory

- Basic predicates are defined as follows:
  - `at(X, Y)`: The agent is at position (X, Y).
  - `blocked(X, Y)`: The position (X, Y) is blocked.
  - `move(Direction, (X1, Y1), (X2, Y2))`: A move from $(X1, Y1)$ to $(X2, Y2)$ in the specified direction.
  - `goal(X, Y)`: The position (X, Y) is the goal.

4.8 Rules

- **Move Left:**

```
move(left, (X, Y), (X-1, Y)) :- not blocked(X-1, Y), X > 0
```

- **Move Right:**

```
move(right, (X, Y), (X+1, Y)) :- not blocked(X+1, Y), X < MaxCols
```

- **Move Up:**

```
move(up, (X, Y), (X, Y-1)) :- not blocked(X, Y-1), Y > 0
```

- **Move Down:**

```
move(down, (X, Y), (X, Y+1)) :- not blocked(X, Y+1), Y < MaxRows
```

- **Reach Goal:**

```
goal(X, Y) :- at(X, Y), X == GoalX, Y == GoalY
```

```python
class Maze:
    def __init__(self, rows, cols, obstacles, start, goal):
        self.rows = rows
        self.cols = cols
        self.obstacles = obstacles  # Set of blocked positions
        self.start = start
        self.goal = goal

    # Check if a position is within bounds and not blocked
    def is_valid(self, x, y):
        return 0 <= x < self.rows and 0 <= y < self.cols and (x, y) not in self
    .obstacles

class StateSpaceSearch:
    def __init__(self, maze):
        self.maze = maze

    def explore_state_space(self):
        # Represent the state space as a queue for exploration
```

```
        from collections import deque
        queue = deque([(self.maze.start, [self.maze.start])])  # (current state, path)
        visited = set()  # Keep track of visited states

        while queue:
            current_state, path = queue.popleft()
            x, y = current_state

            # Check if the goal state is reached
            if current_state == self.maze.goal:
                return path

            # Mark the current state as visited
            visited.add(current_state)

            # Generate possible next states
            for new_state in self.get_neighbors(x, y):
                if new_state not in visited:
                    queue.append((new_state, path + [new_state]))

        return None  # Return None if no path is found

    def get_neighbors(self, x, y):
        # Define possible moves and filter valid ones
        directions = [(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)]  # Left, Right, Up, Down
        return [(nx, ny) for nx, ny in directions if self.maze.is_valid(nx, ny)]

# Example Usage
if __name__ == "__main__":
    # Define the maze
    obstacles = {(0, 1), (2, 1), (3, 1), (2, 3), (3, 4), (4, 4)}
    start_node = (0, 0)
    goal_node = (4, 5)
    maze = Maze(5, 6, obstacles, start_node, goal_node)

    # Perform state space search
    search = StateSpaceSearch(maze)
    path = search.explore_state_space()

    # Print the result
    if path:
        print("Path found:")
        for step in path:
            print(step)
    else:
        print("No path found.")
```

Listing 3: Logic Inference

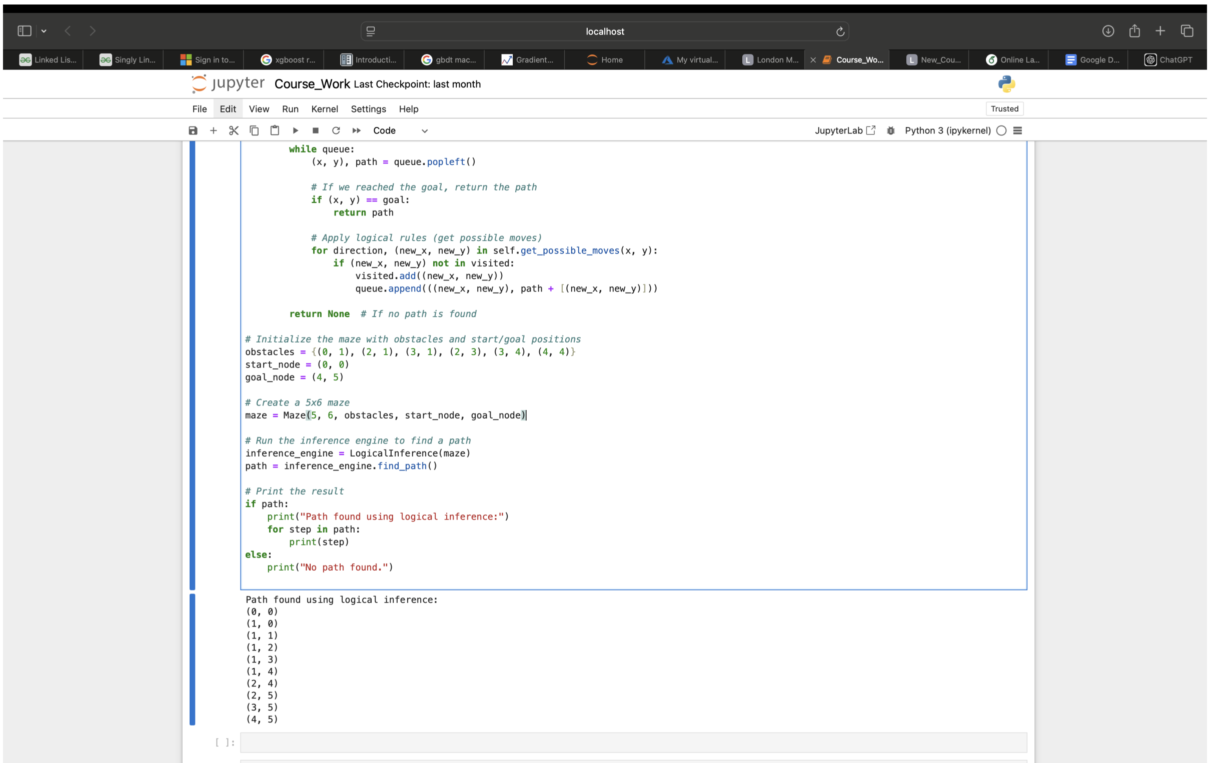


Figure 3: Juypter Notebook Output.