CS 404- Assignment 1

Ozan Başaran

April 2023

Contents

Chapter 1

Color Maze Game

1.1 Introduction

Color-Maze puzzle is a single-agent grid-game played on a rectangular board that includes a maze. Initially, the agent is located on a single maze cell. The agent can move in four cardinal directions: up, down, right or left. Once a direction is chosen, the agent moves in that direction until it reaches a wall at once, and colors all the cells it travels over. Once a cell is colored, its color does not change. The goal is to color all the cells of the maze by moving the agent over them, while minimizing the total distance traveled by the agent.

1.2 Modelling the problem

1.2.1 The Initial State

The game board and how it is represented are as follows: Rectangular game board with a single agent is represented as a grid where each grid cell is uniquely marked with 0, X or S:

- 0 denotes the cells that are empty and that need to be colored,
- X denotes the cells occupied by the walls, and
- S denotes the cell occupied by the agent.

The initial state of the board has no 1's, there exists only 3 characters in the board, S,X and 0's.

This is an exemplary first state representation using matrices, where the agent is denoted by 'S'.

Please note that throughout this report the used Matrices deploy walls to surround the available game space. Please add a wall around the test cases for the algorithm to run without problems.

$$\text{M0}\ \begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & 0 & 0 & 0 & 0 & 0 & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & 0 & X & X & S & X & 0 & X \\
X & 0 & 0 & 0 & 0 & X & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

1.2.2 States

How our agent moves is constrained. When we choose a direction for the agent to move, it colorizes all the 0's until it hits a wall or an X, the colored tiles are represented by 1's.

We can model exemplary states for our initial state matrix:

As the agent can not move up, left or right, it only has down to go. Which makes the successor state of the game represented by a matrix as:

(M0 $\rightarrow Down \rightarrow M1$)

$$\text{M1}\ \begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & 0 & 0 & 0 & 0 & 0 & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & 0 & X & X & 1 & X & 0 & X \\
X & 0 & 0 & 0 & S & X & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

It is easily observable that the tile agent moved from changed to a 1, and that the Agent has moved down until the next tile down is a wall (X).

So now the agent again only has one direction to go, left.

After moving left this is the successor state:

(M1 $\rightarrow Left \rightarrow M2$)

$$\text{M2}\ \begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & 0 & 0 & 0 & 0 & 0 & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & 0 & X & X & 1 & X & 0 & X \\
X & S & 1 & 1 & 1 & X & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

We can clearly see that all the tiles the agent has moved through has been painted, or are now 1's.

Now we ran into a problem, the agent can move both up and down. If the agent chooses to move down, the resulting matrix will be:

(M2 $\rightarrow Down \rightarrow M3$)

$$\text{M3}\ \begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & 0 & 0 & 0 & 0 & 0 & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & 0 & X & X & 1 & X & 0 & X \\
X & 1 & 1 & 1 & 1 & X & 0 & X \\
X & S & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

If the agent chooses to move up the resulting state could be represented as

(M2 $\rightarrow Up \rightarrow M4$)

$$\text{M4}\ \begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & S & 0 & 0 & 0 & 0 & 0 & X \\
X & 1 & X & X & X & X & 0 & X \\
X & 1 & X & X & 1 & X & 0 & X \\
X & 1 & 1 & 1 & 1 & X & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

The agent has chosen to move down, after moving down, there is only one way the agent could move and it is up, After moving up the resultant matrix is:

(M3 $\rightarrow Up \rightarrow M5$)

$$\text{M5}\begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & S & 0 & 0 & 0 & 0 & 0 & X \\
X & 1 & X & X & X & X & 0 & X \\
X & 1 & X & X & 1 & X & 0 & X \\
X & 1 & 1 & 1 & 1 & X & 0 & X \\
X & 1 & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

The agent, now having again only one way to go chooses to go right, again until hitting a wall (M5 $\rightarrow Right \rightarrow M6$)

$$\text{M6}\begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & 1 & 1 & 1 & 1 & 1 & S & X \\
X & 1 & X & X & X & X & 0 & X \\
X & 1 & X & X & 1 & X & 0 & X \\
X & 1 & 1 & 1 & 1 & X & 0 & X \\
X & 1 & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

Only possible route is down this time, which the agent takes:

(M6 $\rightarrow Down \rightarrow M7$)

$$\text{M7}\begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & 1 & 1 & 1 & 1 & 1 & 1 & X \\
X & 1 & X & X & X & X & 1 & X \\
X & 1 & X & X & 1 & X & 1 & X \\
X & 1 & 1 & 1 & 1 & X & 1 & X \\
X & 1 & X & X & X & X & 1 & X \\
X & X & 0 & 0 & 0 & 0 & S & X \\
X & X & X & X & X & X & X & X
\end{array}$$

Having again an only option of going left gives us the:

(M7 $\rightarrow Left \rightarrow M8$)

1.2.3 Goal State and Test

The following matrix is the goal state of the color maze game discussed above.

$$\text{M8}\begin{array}{cccccccc} X & X & X & X & X & X & X & X \\ X & 1 & 1 & 1 & 1 & 1 & 1 & X \\ X & 1 & X & X & X & X & 1 & X \\ X & 1 & X & X & 1 & X & 1 & X \\ X & 1 & 1 & 1 & 1 & X & 1 & X \\ X & 1 & X & X & X & X & 1 & X \\ X & X & S & 1 & 1 & 1 & 1 & X \\ X & X & X & X & X & X & X & X \end{array}$$

The goal state is defined as follows: There exists no 0's on the board (matrix) and the only allowed characters are 1's, X's and S the agent.

A goal test is described as: a function that can be applied to a state and returns true if the state is satisfies the goal condition.

We can derive from our goal matrix that the goal test to be used is checking each cell in the matrix for 0's. If there are 0's the state at hand is not the goal state, if there are no 0's on the board left, than it is indeed the goal state.

The code used below uses string representations of both 0 and 1.

The Goal Test Function:

```
def goaltest(state):
  for row in range(state.shape[0]):
    for col in range(state.shape[1]):
      if state[row][col] == "0":
        return False
  return True
```

1.2.4 The Successor State Function

The list of possible successor states (possible moves) are obtained by checking the adjacent up, down, left and right tiles of the agent.

A function to find the position of the agent is utilized.

```
def find_agent_position(maze):
    for i in range(maze.shape[0]):
        for j in range(maze.shape[1]):
            if maze[i][j] == "S":
                return i, j
```

The following function is then used to return a list of possible moves the agent could make.

```
def get_valid_directions(maze,row,col,visited):
    valid_directions = []
    if row > 0 and maze[row-1][col] != 'X':
        valid_directions.append('up')
    if row < maze.shape[0]-1 and maze[row+1][col] != 'X':
        valid_directions.append('down')
    if col > 0 and maze[row][col-1] != 'X':
        valid_directions.append('left')
    if col < maze.shape[1]-1 and maze[row][col+1] != 'X':
        valid_directions.append('right')
    return valid_directions
```

1.2.5 The Step Cost Function

The Step Cost Function helps punish the AI model so that it does not take infinite amounts of steps to complete the problem, as our goal is to color the maze using the shortest path possible. We are to use a uniform cost c, which we will be deriving from the amount of tiles the agent has moved.

Each tile the agent has traversed increases the step cost by 1.

For example using our previous states, there was a point where our Agent had 2 possible routes to choose from:

$$\text{M2}\begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & 0 & 0 & 0 & 0 & 0 & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & 0 & X & X & 1 & X & 0 & X \\
X & S & 1 & 1 & 1 & X & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

(M2 $\rightarrow Down \rightarrow M3$)

$$\text{M3}\ \begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & 0 & 0 & 0 & 0 & 0 & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & 0 & X & X & 1 & X & 0 & X \\
X & 1 & 1 & 1 & 1 & X & 0 & X \\
X & S & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

(M2 $\rightarrow Up \rightarrow M4$)

$$\text{M4}\ \begin{array}{cccccccc}
X & X & X & X & X & X & X & X \\
X & S & 0 & 0 & 0 & 0 & 0 & X \\
X & 1 & X & X & X & X & 0 & X \\
X & 1 & X & X & 1 & X & 0 & X \\
X & 1 & 1 & 1 & 1 & X & 0 & X \\
X & 0 & X & X & X & X & 0 & X \\
X & X & 0 & 0 & 0 & 0 & 0 & X \\
X & X & X & X & X & X & X & X
\end{array}$$

The Step Cost for the agent taking route (M2 $\rightarrow Down \rightarrow M3$)

is 1 as the agent colors one square before hitting a wall

The Step Cost for the agent taking route (M2 $\rightarrow Up \rightarrow M4$)

is 3 as the agent colors four tiles before hitting a wall.

The Python code to find each route's step costs are as follows:

```python
def get_direction_cost(maze, routes):
    costs = {}
    if not routes:
      a="Pop"
      costs[a]=0
    for direction in routes:
        step_cost = 0
        agent_row, agent_col = find_agent_position(maze)
        if direction == "right":
            while maze[agent_row][agent_col+1] != "X":
                agent_col += 1
                step_cost += 1
            stack.append((agent_row, agent_col))
        elif direction == "left":
            while maze[agent_row][agent_col-1] != "X":
                agent_col -= 1
                step_cost += 1
            stack.append((agent_row, agent_col))
        elif direction == "up":
            while maze[agent_row-1][agent_col] != "X":
                agent_row -= 1
                step_cost += 1
            stack.append((agent_row, agent_col))
        elif direction == "down":
            while maze[agent_row+1][agent_col] != "X":
                agent_row += 1
                step_cost += 1
            stack.append((agent_row, agent_col))
        costs[direction] = step_cost

    return costs
```

When there exists more than one route the agent could take, we want the agent to move choose its next tile randomly. We also keep track of the previous route, the route agent took to end up at this tile. And we are forcing the agent to not take that route, as it would prove an endless loop.

```
import random

def get_least_cost_route(costs, prev_direction=None):
    # Check if there is only one valid direction
    if len(costs) == 1:
        return list(costs.keys())[0], list(costs.values())[0]

    # Choose the direction with the least cost
    min_key = min(costs, key=costs.get)

    # If there are multiple directions with the same cost, choose a random one
    min_keys = [k for k in costs if costs[k] == costs[min_key]]
    min_key = random.choice(min_keys)

    if prev_direction is not None and len(costs) > 1:
        # Check that the chosen direction is not the opposite of the previous di
        opposite_directions = {'up': 'down', 'down': 'up', 'left': 'right', 'rig
        if min_key == opposite_directions[prev_direction]:
            # If the chosen direction is opposite to the previous direction, rem
            del costs[min_key]
            return get_least_cost_route(costs, prev_direction)

    return min_key, costs[min_key]
```

1.3 UCS Algorithm

1.3.1 Introduction

In Uniform Cost Search (UCS), the algorithm keeps track of the cost of each path to a node, and always expands the least costly node in the frontier first. This allows the algorithm to explore the least expensive paths first, ensuring that it finds the optimal path to the goal node. In the context of the puzzle, the UCS algorithm is used to find the optimal path for the agent to reach the goal state while taking into account the cost of each move. If the agent finds no routes available, the stack pops, changing the agent's position to where it last was (the starting position before the move that lead to the agent having no routes).

The above discussed functions helps us find the possible routes the agent might take, find each route's cost and finally find the route with the least cost of move.

1.3.2 Implementation

The function provided below is the main code snippet where the UCS Algorithm runs. In this implementation, the algorithm keeps a stack of where the agent is at, finds available routes agent might take respective to that position, finds each route's cost and finally chooses the route with the least cost until the goal state is achieved.

If the agent runs into a problem where there exists no 0's at the adjacent tiles, the stack being kept comes into the picture. We pop the latest position of the agent from the stack until there exists a viable route our agent might take. The pop operations cost (the cost of traversing back to the tile where there exists are route with 0's to color) is added to the total cost.

The main function of the algorithm:

```
def move_agent(maze, visited, stack):
    total_cost = 0
    prev_route=None
    print("Starting Maze:")
    print(maze)
    while not goal_test(maze):
        agent_pos = find_agent_position(maze)
        row, col = agent_pos
        stack.append((row, col))
        print("\nCurrent Position: ({},{})".format(row, col))
        routes = get_valid_directions(maze, row, col, visited)
        print("Valid Directions: ", routes)
        if not routes:  # No valid directions available
          print("Error: Agent is trapped, cannot move to any direction!")
          break
        # Get the cost of each direction
        costs = get_direction_cost(maze, routes)
        print("Costs: ", costs)

        # Choose the cheapest route
        route, cost = get_least_cost_route(costs,prev_route)
        print("Cheap Route: ", route, "Route Costs: ", cost)
        prev_route=route
        if route == 'right':
          maze[row][col] = '1'
          for i in range(cost):
            maze[row][col+i+1] = '1'
            visited.append((row, col+i+1))
          col += cost
          maze[row][col] = 'S'
        elif route == 'left':
          maze[row][col] = '1'

          for i in range(cost):
            maze[row][col-i-1] = '1'
            visited.append((row, col-i-1))
          col -= cost
          maze[row][col] = 'S'

        elif route == 'down':
          maze[row][col] = '1'

          for i in range(cost):
            maze[row+i+1][col] = '1'
```

```
                visited.append((row+i+1, col))
            row += cost
            maze[row][col] = 'S'

        elif route == 'up':
            maze[row][col] = '1'

            for i in range(cost):
                maze[row-i-1][col] = '1'
                visited.append((row-i-1, col))
            row -= cost
            maze[row][col] = 'S'

        elif route == "Pop":
            a, b = row, col
            if len(stack) > 0:
                row, col = stack.pop()
            c, d = row, col
            total_cost += abs(c - a) + abs(d - b)
            maze[a][b] = "1"
            maze[row][col] = "S"

        # Increment the total cost
        total_cost += costs[route]
        print(maze)

    return maze, total_cost, visited, stack
```

1.4 Extending the Search Model for A* Search

1.4.1 Introduction

A* Search takes into account the cost of getting to the node as well as our estimate of the cost of getting to the goal from n: f(n) = g(n) + h(n) g(n) is the cost of the path to node n h(n) is the heuristic estimate of the cost of getting to a goal node from n. The f-value is an estimate of the cost of getting to the goal via this node (path). Always expand the node with the lowest f-value on the frontier.

1.4.2 The Heuristic Function

The color maze problem does not have a goal node, it doesn't need us to move the agent to a specific location to meet the task. Rather it has end states and these end states are achievable through many different combinations.

Agent's goal at each state or tile, is to color all the remaining 0's. Thus the agent's respective goals at each state is to get to the nearest uncolored tile until the end goal is satisfied.

As the goal is to color all of the "0"s into "1"s, keeping a Manhattan Distance heuristic made sense as at each node the relative distance of the successor empty node is also important. The goal is to traverse or color the maze in the shortest possible way

At every successor state, the algorithm checks that if it had gone there, at which distance the nearest uncolored tile is.

f(n)= g(n)+h(n)

The cost of the path to the node g(n), stays constant to what we have used in the UCS algorithm.

Heuristic estimate h(n) at each successor node, is the Manhattan Distance to the nearest uncolored node.

We don't need to add the heuristic element at each successor node though, if at a state, the agent has only one route to go, there exists no need to expand that node. Rather than that, the heuristic is to be used when the agent is at such a tile that it has more than one possible route to take.

1.4.3 The Admissibility

Consistency:

A heuristic function h(n) is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'.

In other words, h(n) $\leftarrow c(n, a, n') + h(n')$

for all nodes n and n' and all actions a that lead from n to n'

The Manhattan Distance to the nearest uncolored tile satisfies this property because it is always an underestimate of the true cost to the goal. This is because the Manhattan Distance is the shortest possible distance that can be travelled in a grid-like environment where only horizontal and vertical movements are allowed, and there are no obstacles. Since obstacles and diagonal movements are not considered in the calculation of the heuristic, the actual cost to the goal will always be greater than or equal to the Manhattan Distance.

Admissibility:

A heuristic function h(n) is admissible if it never overestimates the cost to reach the goal from n.

The Manhattan Distance to the nearest uncolored tile satisfies this property because, as mentioned earlier, it is always an underestimate of the true cost to the goal. This means that the heuristic will never overestimate the actual cost, and will always lead the agent to the optimal path.

Using the matrix below, let's observe how the program would behave when it comes across a decision node (a node with more than one possible route to take).

$$\text{Matrix-2} \begin{array}{cccccccc} X & X & X & X & X & X & X & X \\ X & 0 & 0 & 0 & 0 & 0 & 0 & X \\ X & 0 & X & X & X & X & 0 & X \\ X & 0 & X & X & S & X & 0 & X \\ X & 0 & 0 & 0 & 0 & X & 0 & X \\ X & 0 & X & X & X & X & 0 & X \\ X & X & 0 & 0 & 0 & 0 & 0 & X \\ X & X & X & X & X & X & X & X \end{array}$$

There exists only one decision node at this matrix and it's at this state below:

$$\text{Matrix-2} \begin{array}{cccccccc} X & X & X & X & X & X & X & X \\ X & 0 & 0 & 0 & 0 & 0 & 0 & X \\ X & 0 & X & X & X & X & 0 & X \\ X & 0 & X & X & 1 & X & 0 & X \\ X & S & 1 & 1 & 1 & X & 0 & X \\ X & 0 & X & X & X & X & 0 & X \\ X & X & 0 & 0 & 0 & 0 & 0 & X \\ X & X & X & X & X & X & X & X \end{array}$$

The agent now has 2 routes to choose from: Matrix-2 [1][1] Tile A or Matrix-2[5][1] Tile B or Up and Down respectively.

Let's compare both cases. Setting our next state to agent moving to tile B has a cost of 1. At the tile B, the nearest uncolored tiles are at Matrix-2[6][2] and Matrix-2[3][1] which both have Manhattan Distances of 2 to the tile B.

If our agent were to move to tile A, the cost of move would be 3. At the tile A, the nearest uncolored tile is at Matrix-2[1][2] which has a Manhattan Distance to tile A of 1.

As our final function was f(n) = g(n) + h(n)

Tile B= 1 (cost of moving to the tile) + 2 (Manhattan Distance of the closest uncolored tile) = 3

Tile A= 3(cost of moving to the tile) + 1 (Manhattan Distance of the closest uncolored tile) = 4

The algorithm selects to move to Tile B first. Which makes sense as moving down first then going up is a more feasible solution.

1.5 A* Algorithm Implementation

1.5.1 Helper Functions

We are to employ a helper function. This function is to go to where the agent would be if it took the selected route, and calculates where the nearest 0 is.

```
def find_nearest_zero(maze, row, col):
nearest_zero_dist = float('inf')
for i in range(len(maze)):
    for j in range(len(maze[0])):
        if maze[i][j] == '0':
            dist = abs(i - row) + abs(j - col)  # calculate Manhattan distan
            if dist < nearest_zero_dist:
                nearest_zero_dist = dist
return nearest_zero_dist
```

After this function we need a function to help evaluate our f(n) value. To do this the following function is employed

```
import copy

def get_direction_costs_a(amaze, routes):
    maze = copy.deepcopy(amaze)
    costs = {}
    total_costs = {}
    if not routes:
        a = "Pop"
        costs[a] = 0
    for direction in routes:
        step_cost = 0
        total = 0
        agent_row, agent_col = find_agent_position(maze)
        if direction == "right":
            while maze[agent_row][agent_col+1] != "X":
                agent_col += 1
                step_cost += 1
                maze[agent_row][agent_col] = "1"
                if goal_test(maze):
                    costs[direction]=step_cost
                    total_costs[direction]=total
                    return costs,total_costs

            cost_a = find_nearest_zero(maze, agent_row, agent_col)
            total = cost_a + step_cost
        elif direction == "left":
            while maze[agent_row][agent_col-1] != "X":
                agent_col -= 1
                step_cost += 1
                maze[agent_row][agent_col] = "1"
                if goal_test(maze):
                    costs[direction]=step_cost
                    total_costs[direction]=total
                    return costs, total_costs
            cost_a = find_nearest_zero(maze, agent_row, agent_col)
            total = cost_a + step_cost
        elif direction == "up":
            while maze[agent_row-1][agent_col] != "X":
                agent_row -= 1
                step_cost += 1
                maze[agent_row][agent_col] = "1"
                if goal_test(maze):
                    costs[direction]=step_cost
                    total_costs[direction]=total
                    return costs, total_costs
            cost_a = find_nearest_zero(maze, agent_row, agent_col)
            total = cost_a + step_cost
        elif direction == "down":
            while maze[agent_row+1][agent_col] != "X":
                agent_row += 1
                step_cost += 1
                maze[agent_row][agent_col] = "1"
                if goal_test(maze):
                    costs[direction]=step_cost
                    total_costs[direction]=total
                    return costs, total_costs
            cost_a = find_nearest_zero(maze, agent_row, agent_col)
            total = cost_a + step_cost
        total_costs[direction] = total
        costs[direction] = step_cost
    return costs, total_costs
```

The function above copies our maze, on that copy conducts the following operations: It expands the decision nodes, painting the tiles as if it had gone there, after that the algorithm tries finding if the current maze is firstly a complete solution, and then where the nearest zero is to that given tile.

The total costs dictionary is used to compute our f(n) value using the heuristics, while the costs dictionary holds the original path (step) cost.

The function below, chooses the route with the least evaluated f(n) value, and returns the route name and the total cost of taking that route.

```
    import random

def get_least_cost_route_a(costs, prev_direction=None):
    # Check if there is only one valid direction
    if len(costs) == 1:
        return list(costs.keys())[0], list(costs.values())[0]

    # Choose the direction with the least cost
    min_key = min(costs, key=costs.get)

    # If there are multiple directions with the same cost, choose a random one
    min_keys = [k for k in costs if costs[k] == costs[min_key]]
    min_key = random.choice(min_keys)

    if prev_direction is not None and len(costs) > 1:
        # Check that the chosen direction is not the opposite of the previous di
        opposite_directions = {'up': 'down', 'down': 'up', 'left': 'right', 'rig
        if min_key == opposite_directions[prev_direction]:
            # If the chosen direction is opposite to the previous direction, rem
            del costs[min_key]
            return get_least_cost_route(costs, prev_direction)

    return min_key, costs[min_key]
```

Finally comes the main function where it differs very little from the one used in the UCS Algorithm.

```
def move_agent_a(maze, visited, stack):
    total_cost = 0
    print("Starting␣Maze:")
    print(maze)
    prev_route=None
    while not goal_test(maze):
        agent_pos = find_agent_position(maze)
        row, col = agent_pos
        stack.append((row, col))
        print("\nCurrent␣Position:␣({},{})".format(row, col))
        routes = get_valid_directions(maze, row, col, visited)
        print("Valid␣Directions:␣", routes)
        if not routes:  # No valid directions available
          print("Error:␣Agent␣is␣trapped,␣cannot␣move␣to␣any␣direction!")
          break
        if len(routes) > 1:
            costs, heuristic_costs = get_direction_costs_a(maze, routes)
            print("Costs:␣", costs)
            print("Costs␣with␣Heuristics:␣", heuristic_costs)
            route,heuristic_cost= get_least_cost_route_a(heuristic_costs,prev_ro
            cost=costs[route]
            # If we are not at a decision node, continue like usual taking the o
        else:
            costs = get_direction_cost(maze, routes)
            print("Costs:␣", costs)
            route, cost = get_least_cost_route(costs)

        print("Cheap␣Route:␣", route, "Route␣Costs:␣", cost)
        prev_route=route
        if route == 'right':
            maze[row][col] = '1'
            for i in range(cost):
                maze[row][col+(i+1)] = '1'
                visited.append((row, col+(i+1)))
            stack.append((row, col+cost))
            col += cost
            maze[row][col] = 'S'
        elif route == 'left':
            maze[row][col] = '1'
            for i in range(cost):
                maze[row][col-(i+1)] = '1'
                visited.append((row, col-(i+1)))
            stack.append((row, col-cost))
```

```
                col -= cost
                maze[row][col] = 'S'
            elif route == 'down':
                maze[row][col] = '1'
                for i in range(cost):
                    maze[row+(i+1)][col] = '1'
                    visited.append((row+(i+1), col))
                stack.append((row+cost, col))
                row += cost
                maze[row][col] = 'S'
            elif route == 'up':
                maze[row][col] = '1'
                for i in range(cost):
                    maze[row-(i+1)][col] = '1'
                    visited.append((row-(i+1), col))
                stack.append((row-cost, col))
                row -= cost
                maze[row][col] = 'S'
            elif route == "Pop":
                maze[row][col] = "1"
                a, b = row, col
                if len(stack) > 0:
                    stack.pop()
                    e = stack.pop()
                    print()
                    maze[row][col] = "S"
                    c, d = row, col
                    total_cost += abs(c - a) + abs(d - b)
            total_cost += costs.get(route, 0)
            print(maze)
            print(stack)

        return maze, total_cost, visited, stack
```

1.6 Evaluation

1.6.1 Difficulty Levels

We are asked to provide 5 maze samples of 10x10, 11x11 for this reports case of 3 difficulty levels, namely Easy, Normal and Hard.

Easy levels for this report contain one or two decision nodes at max, we will start from the easiest of levels incrementing the difficulty at each time.

Normal levels have more than 2 decision nodes each, with more than 15 tiles to color.

Hard levels have more than 5 decision nodes each with more than 30 tiles to color for all.

To exemplify here are some of the levels.

Please note that LaTex uses a maximum of 10 items per matrix row, thus every even second row that contains 1 element is actually the last column of the previous row. Sorry for the inconvenience.

The Easy one:

$$\text{Matrix1}\begin{bmatrix}
X & X & X & X & X & X & X & X & X & X \\
X \\
X & X & X & X & X & X & X & X & X & X \\
X \\
X & X & X & X & X & X & X & X & X & X \\
X \\
X & X & X & X & X & X & X & X & X & X \\
X \\
X & X & X & X & X & X & X & X & X & X \\
X \\
X & X & X & X & X & X & X & X & X & X \\
X \\
X & X & X & X & X & X & X & X & X & X \\
X \\
X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 \\
X \\
X & X & X & X & X & X & X & X & & 0 \\
X \\
X & X & X & X & X & X & X & X & X & S \\
X \\
X & X & X & X & X & X & X & X & X & X \\
X
\end{bmatrix}$$

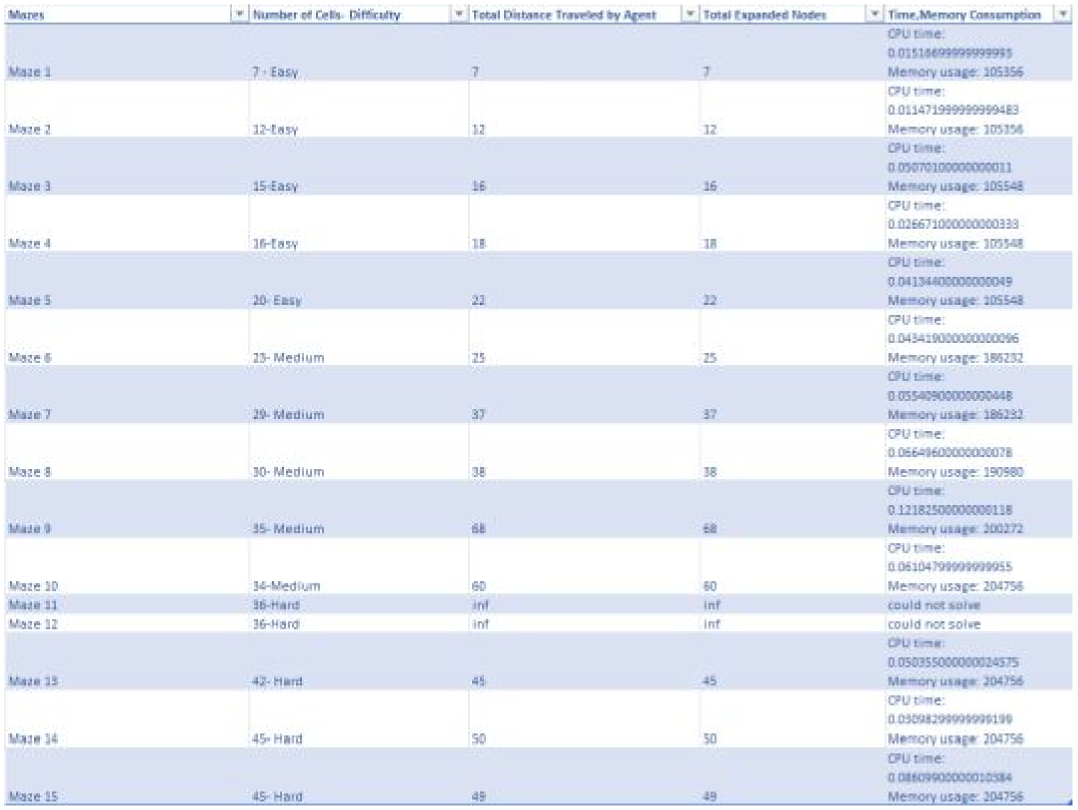
The Medium one:

$$\text{Matrix6}\begin{bmatrix} X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 \\ X \\ X & X & X & X & X & X & X & X & & 0 \\ X \\ X & X & X & X & X & X & X & X & X & S \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \end{bmatrix}$$

The Hard one:

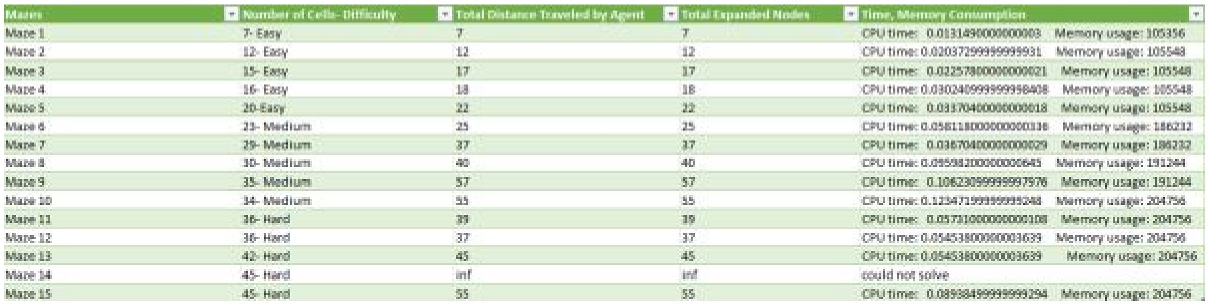
$$\text{Matrix11}\begin{bmatrix} X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \\ X & X & X & X & 0 & 0 & 0 & 0 & 0 & 0 \\ X \\ X & X & X & X & X & X & X & X & & 0 \\ X \\ X & X & X & X & X & X & X & X & X & S \\ X \\ X & X & X & X & X & X & X & X & X & X \\ X \end{bmatrix}$$

The Table below gauges the UCS Algorithm



Mazes	Number of Cells- Difficulty	Total Distance Traveled by Agent	Total Expanded Nodes	Time,Memory Consumption
Maze 1	7 - Easy	7	7	CPU time: 0.015186999999999993 Memory usage: 105356
Maze 2	12-Easy	12	12	CPU time: 0.011471999999999483 Memory usage: 105356
Maze 3	15-Easy	16	16	CPU time: 0.05070100000000011 Memory usage: 105548
Maze 4	16-Easy	18	18	CPU time: 0.026671000000000333 Memory usage: 105548
Maze 5	20- Easy	22	22	CPU time: 0.04134400000000049 Memory usage: 105548
Maze 6	23- Medium	25	25	CPU time: 0.043419000000000096 Memory usage: 186232
Maze 7	29- Medium	37	37	CPU time: 0.05540900000000448 Memory usage: 186232
Maze 8	30- Medium	38	38	CPU time: 0.06649600000000078 Memory usage: 190980
Maze 9	35- Medium	68	68	CPU time: 0.1218250000000118 Memory usage: 200272
Maze 10	34-Medium	60	60	CPU time: 0.06104799999999955 Memory usage: 204756
Maze 11	36-Hard	inf	inf	could not solve
Maze 12	36-Hard	inf	inf	could not solve
Maze 13	42- Hard	45	45	CPU time: 0.050355000000024575 Memory usage: 204756
Maze 14	45- Hard	50	50	CPU time: 0.03098299999999199 Memory usage: 204756
Maze 15	45- Hard	49	49	CPU time: 0.08609900000010584 Memory usage: 204756

The Table below gauges the A* Search Algorithm



Mazes	Number of Cells- Difficulty	Total Distance Traveled by Agent	Total Expanded Nodes	Time, Memory Consumption
Maze 1	7- Easy	7	7	CPU time: 0.0131490000000003 Memory usage: 105356
Maze 2	12- Easy	12	12	CPU time: 0.020372999999999931 Memory usage: 105548
Maze 3	15- Easy	17	17	CPU time: 0.022578000000000021 Memory usage: 105548
Maze 4	16- Easy	18	18	CPU time: 0.030240999999998408 Memory usage: 105548
Maze 5	20-Easy	22	22	CPU time: 0.03370400000000018 Memory usage: 105548
Maze 6	23- Medium	25	25	CPU time: 0.058118000000000336 Memory usage: 186232
Maze 7	29- Medium	37	37	CPU time: 0.036704000000000029 Memory usage: 186232
Maze 8	30- Medium	40	40	CPU time: 0.09598200000000645 Memory usage: 191244
Maze 9	35- Medium	57	57	CPU time: 0.10623099999997976 Memory usage: 191244
Maze 10	34- Medium	55	55	CPU time: 0.12347199999995248 Memory usage: 204756
Maze 11	36- Hard	39	39	CPU time: 0.05731000000000108 Memory usage: 204756
Maze 12	36- Hard	37	37	CPU time: 0.05453800000003639 Memory usage: 204756
Maze 13	42- Hard	45	45	CPU time: 0.05453800000003639 Memory usage: 204756
Maze 14	45- Hard	inf	inf	could not solve
Maze 15	45- Hard	55	55	CPU time: 0.08938499999999294 Memory usage: 204756