CSE366: Artificial Intelligence
[Spring 2024]

Report on Assignment 01
Enhanced Dynamic Robot Movement Simulation

Submitted by

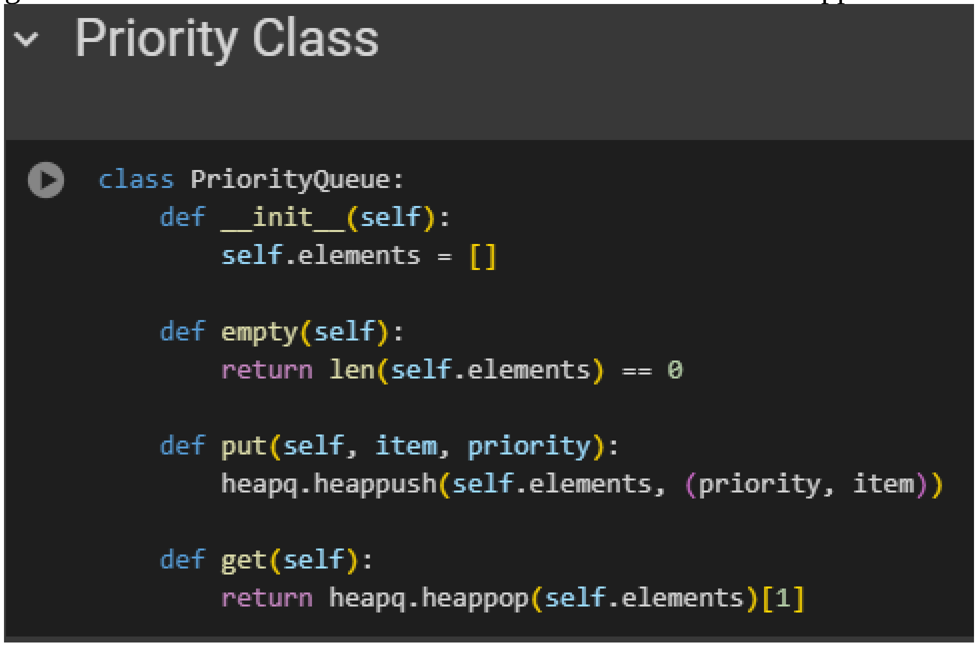
Student ID: 2020-3-60-073
Student Name: Md. Iftekhar Hossain Khan

Introduction:

This assignment is about Enhance Dynamic Robot Movement Simulation. Here we've simulated the robot movement using two of the pathfinding algorithms, UCS and A*.

Priority Class:

The 'Priority Class' was used in both algorithms. This class is mainly implemented for maintaining the nodes at each movement the robot makes and chooses the prioritized node each time after determining the obstacles and then moves to that node. Below is a code snippet of 'Priority Class'

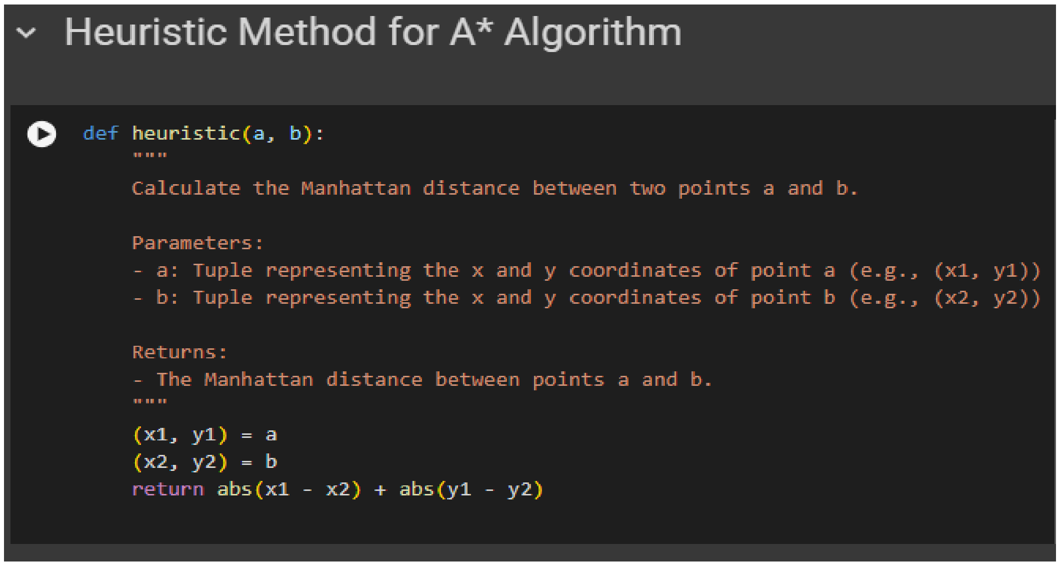


Priority Class

```
class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]
```

Heuristic Method:

This method is mainly for the A* algorithm. It is used for heuristically calculate the selection of the next node where the robot will perform its next movement. Below is the code snippet of the method:

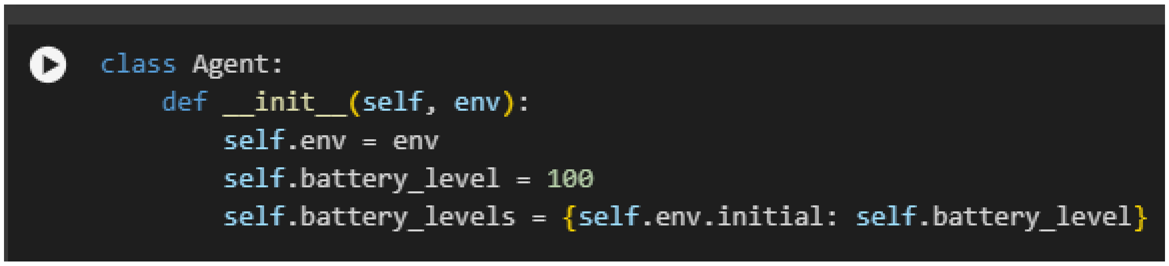


Heuristic Method for A* Algorithm

```
def heuristic(a, b):
    """
    Calculate the Manhattan distance between two points a and b.

    Parameters:
    - a: Tuple representing the x and y coordinates of point a (e.g., (x1, y1))
    - b: Tuple representing the x and y coordinates of point b (e.g., (x2, y2))

    Returns:
    - The Manhattan distance between points a and b.
    """
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)
```

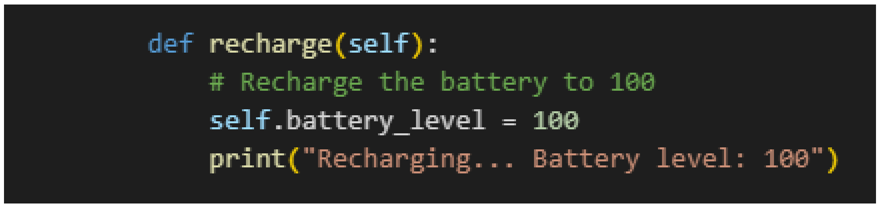
Agent Class:
In our Agent Class, we added the algorithms of UCS and A* methods to perform the robot simulation. Our additional task was to add the battery level at each movement the robot takes for which we added a variable called 'battery_level' and a dictonary called 'battery_levels' to store the information of the battery level in each movement. Also, we additionally added a method called 'recharge()' where the initial value of 'battery_level' sets to 100 when the battery level reaches to 0.

The battery level data is built when the path is successfully found and reconstructing in the reconstruct_path() method. Here are the code snippets below for the additional codes:



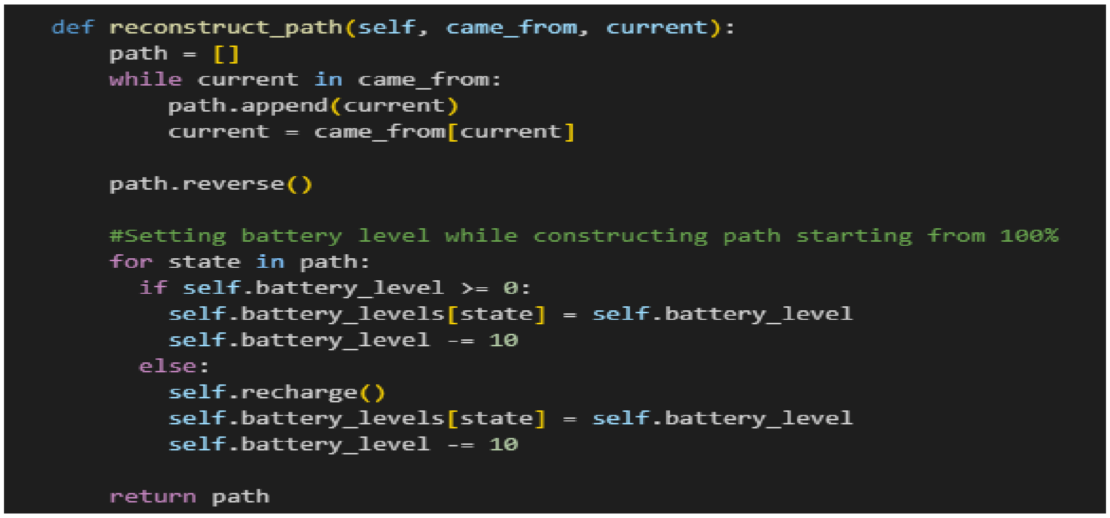
```
class Agent:
    def __init__(self, env):
        self.env = env
        self.battery_level = 100
        self.battery_levels = {self.env.initial: self.battery_level}
```

Figure 1: variables and dictionary of battery_levels added



```
    def recharge(self):
        # Recharge the battery to 100
        self.battery_level = 100
        print("Recharging... Battery level: 100")
```

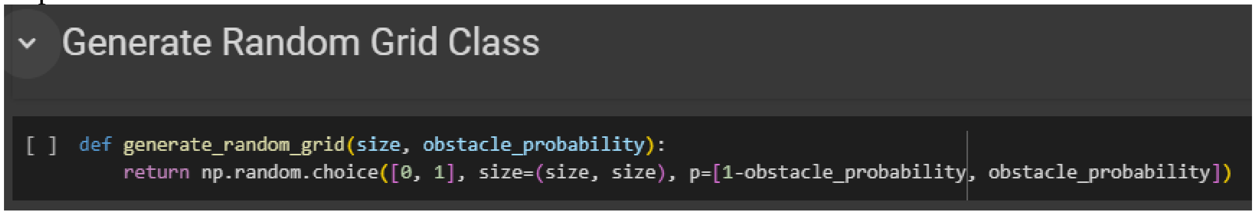
Figure 2: recharge() method for setting the initial value of batter_level to 100 once reaches to 0



```
def reconstruct_path(self, came_from, current):
    path = []
    while current in came_from:
        path.append(current)
        current = came_from[current]

    path.reverse()

    #Setting battery level while constructing path starting from 100%
    for state in path:
      if self.battery_level >= 0:
        self.battery_levels[state] = self.battery_level
        self.battery_level -= 10
      else:
        self.recharge()
        self.battery_levels[state] = self.battery_level
        self.battery_level -= 10

    return path
```

Figure 3: battery levels are built while the path gets reconstructed

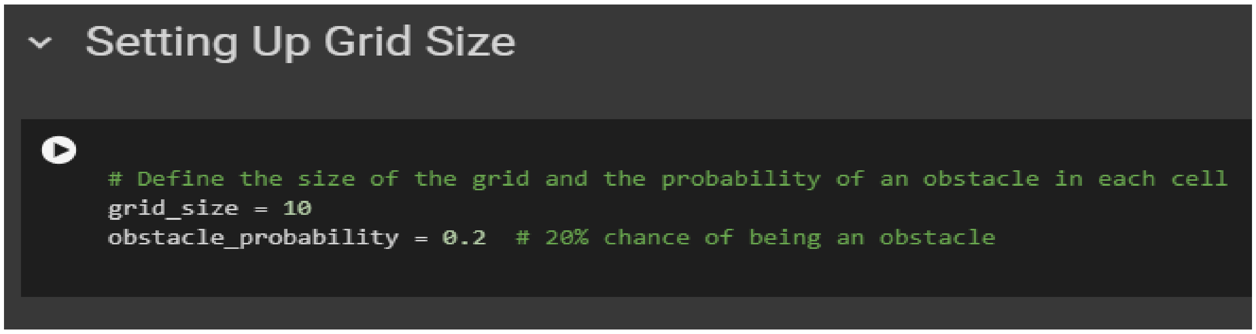
Setting up the Randomly Generated Grid Method:
For dynamically generating random grid, we have used numpy library, the code below was implemented:



Generate Random Grid Class

```
def generate_random_grid(size, obstacle_probability):
    return np.random.choice([0, 1], size=(size, size), p=[1-obstacle_probability, obstacle_probability])
```

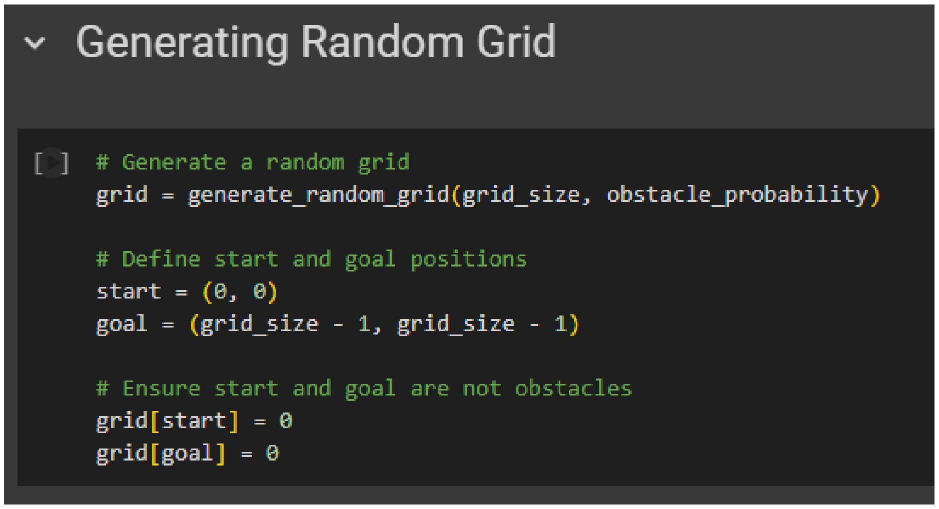
Figure 4: Generate random grid class



Setting Up Grid Size

```
# Define the size of the grid and the probability of an obstacle in each cell
grid_size = 10
obstacle_probability = 0.2  # 20% chance of being an obstacle
```

Figure 5: Setting up the grid size

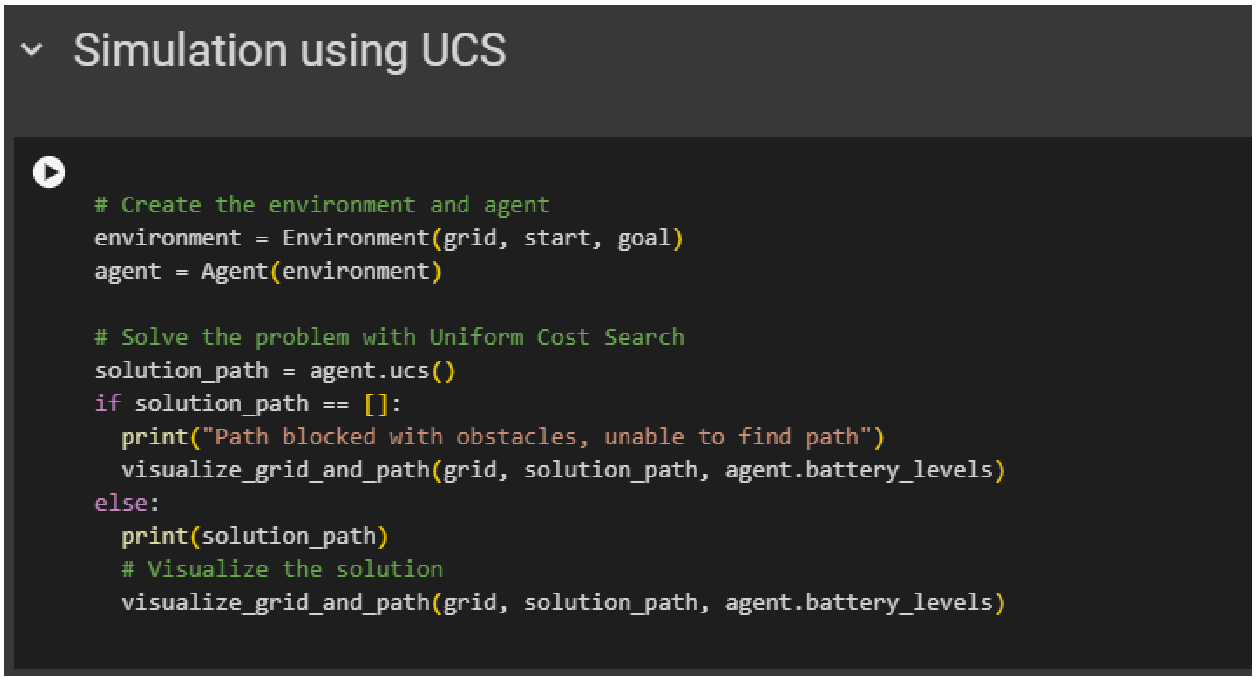


Generating Random Grid

```
# Generate a random grid
grid = generate_random_grid(grid_size, obstacle_probability)

# Define start and goal positions
start = (0, 0)
goal = (grid_size - 1, grid_size - 1)

# Ensure start and goal are not obstacles
grid[start] = 0
grid[goal] = 0
```

Figure 6: Using the generating random grid method to generate the grid class

Simulation Result:



Simulation using UCS

```
# Create the environment and agent
environment = Environment(grid, start, goal)
agent = Agent(environment)

# Solve the problem with Uniform Cost Search
solution_path = agent.ucs()
if solution_path == []:
  print("Path blocked with obstacles, unable to find path")
  visualize_grid_and_path(grid, solution_path, agent.battery_levels)
else:
  print(solution_path)
  # Visualize the solution
  visualize_grid_and_path(grid, solution_path, agent.battery_levels)
```

Figure 7: Setting up simulation for UCS algorithm

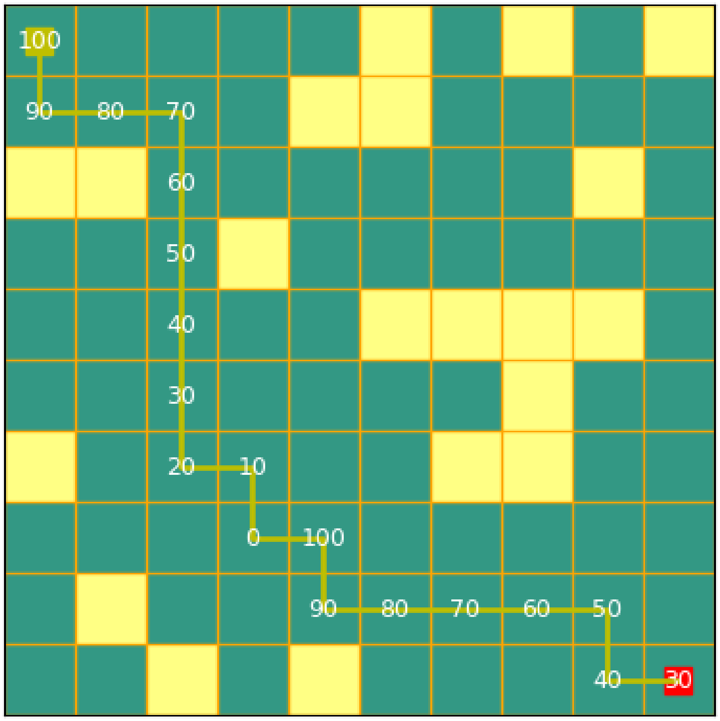
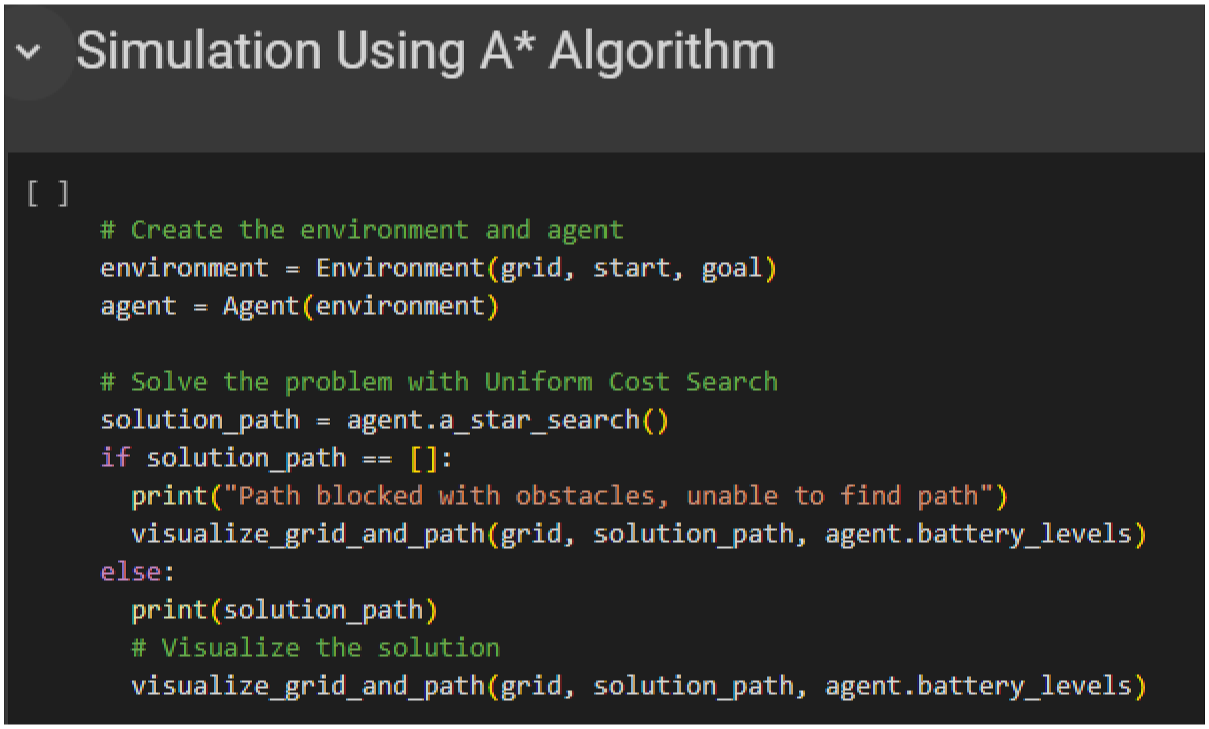


Figure 8: Simulation result for the UCS algorithm after generating a random 10x10 grid



Simulation Using A* Algorithm

```
[ ]
    # Create the environment and agent
    environment = Environment(grid, start, goal)
    agent = Agent(environment)

    # Solve the problem with Uniform Cost Search
    solution_path = agent.a_star_search()
    if solution_path == []:
      print("Path blocked with obstacles, unable to find path")
      visualize_grid_and_path(grid, solution_path, agent.battery_levels)
    else:
      print(solution_path)
      # Visualize the solution
      visualize_grid_and_path(grid, solution_path, agent.battery_levels)
```

Figure 9: Setting up simulation for A* algorithm

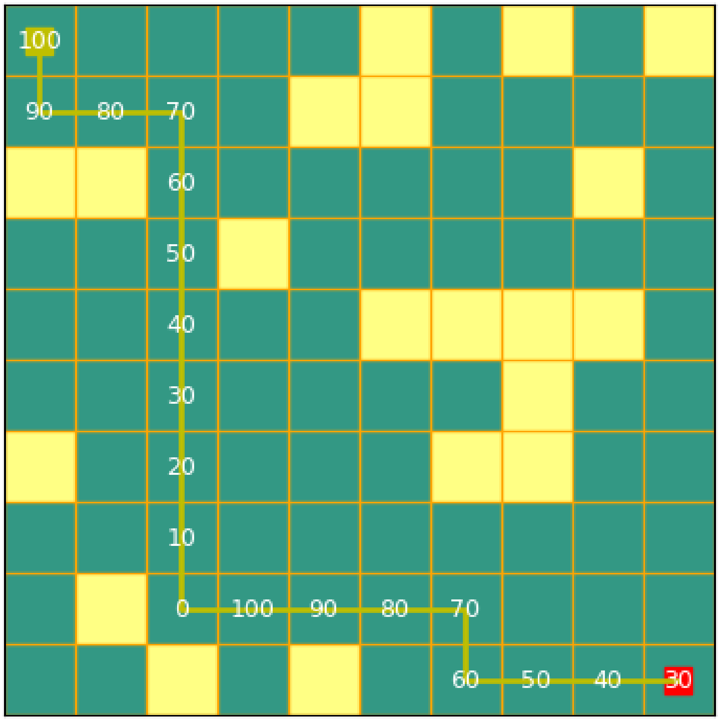


Figure 10: Simulation result for the A* algorithm on the same grid which was used in UCS method.