Artificial Intelligence

(SSZC444)

Assignment-5

Name: Prajwal S Telkar

BITS ID: 2023MT12205

Birla Institute of Technology and Science

MTech Software Systems

Submitted to,

Professor: Brahmanaidu Kakarla

Introduction

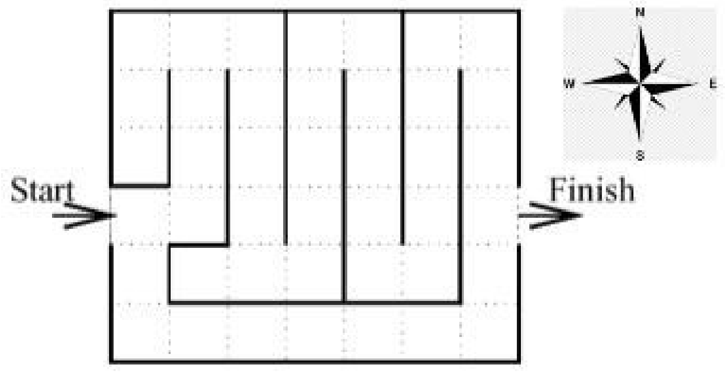
Artificial Intelligence (AI) is a field that encompasses the development of intelligent systems capable of performing tasks that typically require human-like cognitive abilities. At the heart of AI lies algorithms, which are sets of instructions designed to solve specific problems or accomplish tasks. These algorithms power the decision-making processes of AI systems, enabling them to analyze data, learn from it, and make predictions or recommendations.

Algorithms in AI come in various forms, each tailored to address different challenges. From classic algorithms like depth-first search and breadth-first search to more advanced techniques such as genetic algorithms and neural networks, the diversity of algorithms in AI reflects the breadth of problems they can tackle. These algorithms leverage mathematical principles, logical reasoning, and computational techniques to navigate complex problems and find optimal solutions.

I am using **Python** as the scripting language. I have utilized the existing python library **pyamaze** while importing modules like maze, agent, COLOR, to create the maze and to showcase the animation of robot going through the maze from start state to goal state in the best path.

Problem statement

Given the below maze configuration, the task of the robot is to navigate in the maze and find the optimal path to reach the finish position. It can move to the north, south, west and east direction. While navigating through the environment it has obstacles like walls. For each transition, a path cost of +3 is added in search. Assume that the robot's vision sensors are sensitive to the exposure to the sunlight and whenever it tries to move towards the east direction resulting in incurring an additional penalty of +5 cost. Use Euclidean distance as a heuristic wherever necessary.



Use the following algorithms to find the optimal path.

A* Algorithm

Hill Climbing Algorithm

PEAS (Performance measure, Environment, Actuator, Sensor)

1) **Performance Measure:** This defines how the success of an agent in performing a task is measured. It could be a simple binary measure or a more complex measure such as a numeric value representing the efficiency or effectiveness of the agent's behavior.
   Here for maze navigation robot, I have considered the time taken to reach the goal, the number of steps taken, or the total cost incurred during navigation.
2) **Environment:** This refers to the external context or surroundings in which the agent operates. It includes everything the agent interacts with or perceives while performing its task. Here the agent (robot) is in the maze environment where it includes walls (obstacles), open spaces, and the start and goal positions.
3) **Actuators:** Actuators are the mechanisms through which the agent can affect its environment. They are responsible for carrying out the actions decided upon by the agent based on its perceptions and reasoning. Here for our agent (robot), the actuators could be the motors or controls that allow it to move in different directions (north, south, east, west).
4) **Sensors:** Sensors are the means through which the agent perceives its environment. They provide the agent with information about the state of the environment, which the agent then uses to make decisions and take actions. Here for our agent (robot), the sensors could be proximity sensors to detect walls or obstacles, or sensors to detect the goal position.

A* Algorithm:

The A* algorithm is a widely used search algorithm renowned for its effectiveness in finding the shortest path in graphs or networks. It combines the benefits of both Dijkstra's algorithm and greedy best-first search, considering both the cost incurred to reach a node and an estimate of the cost required to reach the destination. This estimate, often referred to as a heuristic, guides the search process towards the most promising paths.

In the context of the maze navigation problem, A* can be utilized to guide the robot through the maze while minimizing the path cost. By incorporating a suitable heuristic, such as the

Euclidean distance to the destination, the algorithm can intelligently prioritize paths that are more likely to lead to the goal. Additionally, A* allows for the incorporation of penalties associated with specific actions, such as the additional cost incurred when moving towards the east direction due to sensitivity to sunlight.

By leveraging the A* algorithm, the robot can navigate the maze efficiently, avoiding obstacles and optimizing its path to reach the finish position while considering the path costs and penalties along the way.

Hill Climbing Algorithm:

Hill Climbing is a local search algorithm that iteratively explores neighboring solutions with the aim of finding the optimal solution. It starts with an initial solution and iteratively moves to the neighboring solutions that lead to improvements in the objective function until a local optimum is reached.

By incorporating the Hill Climbing algorithm into the navigation strategy, the robot can refine its path through the maze, potentially discovering alternative routes or optimizations that were not initially apparent. This iterative refinement process enhances the robot's ability to navigate complex environments effectively, ultimately leading to improved performance in reaching the finish position while minimizing path costs and penalties.

Difference between A* and Hill Climbing Algorithm

	A* Algorithm	Hill Climbing Algorithm
Objective	A* aims to find the shortest path from a start node to a goal node by considering both the cost to reach the current node and the estimated cost to reach the goal node from the current node.	Hill Climbing aims to find the best solution by iteratively moving towards the direction that improves the current solution, without considering the global optimal solution.
Backtracking	A* maintains a record of the path taken to reach each node and can backtrack if a better path is found later.	Hill Climbing does not maintain a record of the path taken and does not backtrack. It only moves to a neighboring state if it improves the current state.

Completeness	A* is complete, meaning it is guaranteed to find a solution if one exists, given a consistent heuristic.	Hill Climbing is not complete. It can get stuck in local optima and may not find the global optimal solution.
Optimality	A* is optimal if the heuristic function is admissible, meaning it never overestimates the cost to reach the goal. In such cases, A* is guaranteed to find the shortest path.	Hill Climbing is not optimal. It can find a local optimum but may not find the global
Memory Usage	A* may use more memory than Hill Climbing because it needs to store information about the nodes visited and the paths taken.	Hill Climbing typically uses less memory than A* because it only needs to remember the current state and the best neighboring state.

Implementation

In the program, I have created two utility functions:

- A heuristic function which calculates the Euclidian distance between the neighbor and goal state.

```
def heuristic(cell1, cell2):
    x1, y1 = cell1
    x2, y2 = cell2
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
```

- I have created a function getNeighbor(cell,direction) which takes the input as the cell position and based on the direction returns the position of cell where agent can make its next move.

```
def getNeighbor(cell, direction):
    x, y = cell
    if direction == 'E':
        return (x, y + 1)
    elif direction == 'W':
        return (x, y - 1)
    elif direction == 'N':
        return (x - 1, y)
    elif direction == 'S':
        return (x + 1, y)
    else:
        raise ValueError("Invalid direction")
```

A* algorithm

Initialization:

- The algorithm starts at cell (4, 1) and the goal is to reach cell (4, 7).
- The priority queue open is initialized with the start cell (4, 1) and its cost (0).
- nodes_expanded to keep track nodes expanded by the algorithm to get the optimal path.

```
def aStar(m):
    start = (4, 1)
    goal = (4, 7)
    open = PriorityQueue()
    open.put((0, start))
    came_from = {}
    cost_so_far = {start: 0}
    nodes_expanded = 0

    start_time = time.time()
```

Exploration:

```
while not open.empty():
    nodes_expanded += 1
    current_cost, current = open.get()
    print("Current cell:", current)
    if current == goal:
        break

    for d in 'ESNW':
        if m.maze_map[current][d]:
            if d == 'E':
                new_cost = cost_so_far[current] + 3 + 5  # Cost for moving
east
            else:
                new_cost = cost_so_far[current] + 3

            neighbor = getNeighbor(current, d)
            if neighbor not in cost_so_far or new_cost <
cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + heuristic(neighbor, goal)
                open.put((priority, neighbor))
                came_from[neighbor] = current
                print("    ->", d, "Neighbor:", neighbor, "New cost:",
new_cost, "Priority:", priority)

path_cost = cost_so_far[goal]  # Total cost of the path
print("\nTotal cost of the path:", path_cost)

path = []
current = goal
while current != start:
    path.append(current)
    current = came_from[current]
path.append(start)
path.reverse()
end_time = time.time()
time_taken = end_time - start_time
return path, path_cost, nodes_expanded, time_taken
```

- The algorithm explores the neighboring cells of the current cell, prioritizing those with lower costs.
- For each neighboring cell, it calculates the cost to reach that cell and the total estimated cost (priority) to reach the goal from that cell.
- The algorithm prints information about the current cell, the direction of movement, the neighbor cell, the new cost, and the priority.
- Path Cost: Once the goal cell (4, 7) is reached, the algorithm calculates the total cost of the optimal path from the start to the goal cell.

Driver script

```
if __name__ == '__main__':
    m = maze(6, 7)
    m.CreateMaze(loadMaze='Assignment_5.csv', x=4, y=7, theme=COLOR.light)
    apath, apath_cost, a_star_nodes_expanded, a_star_time_taken = aStar(m)

    a1 = agent(m, 4, 1, footprints=True, color=COLOR.blue, filled=True,
shape='arrow')
    m.tracePath({a1: apath})

    print("\nA* Algorithm:")
    print("Path:", apath)
    print("Total cost of the path:", apath_cost)
    print("Number of Nodes Expanded:", a_star_nodes_expanded)
    print("Time Taken:", a_star_time_taken)

    label = textLabel(m, 'A Star Algorithm: Path Cost', apath_cost)

    m.run()
```

- Sets up the maze environment such that a maze object is created with dimensions 6 rows by 7 columns.

- generates the maze using data loaded from a CSV file named 'Assignment_5.csv'. Goal state is set at position (4, 7) in the maze, and the theme of the maze is set to light colors.
- Apply the A* algorithm to find an optimal path in the maze.
- Creates an agent to navigate through the maze based on the path found. An agent object is created and placed at position (4, 1) in the maze. It leaves footprints as it moves, represented by blue arrows.
- Prints out information about the path found by the A* algorithm.
- Displays the maze with the optimal path traced.

Output:

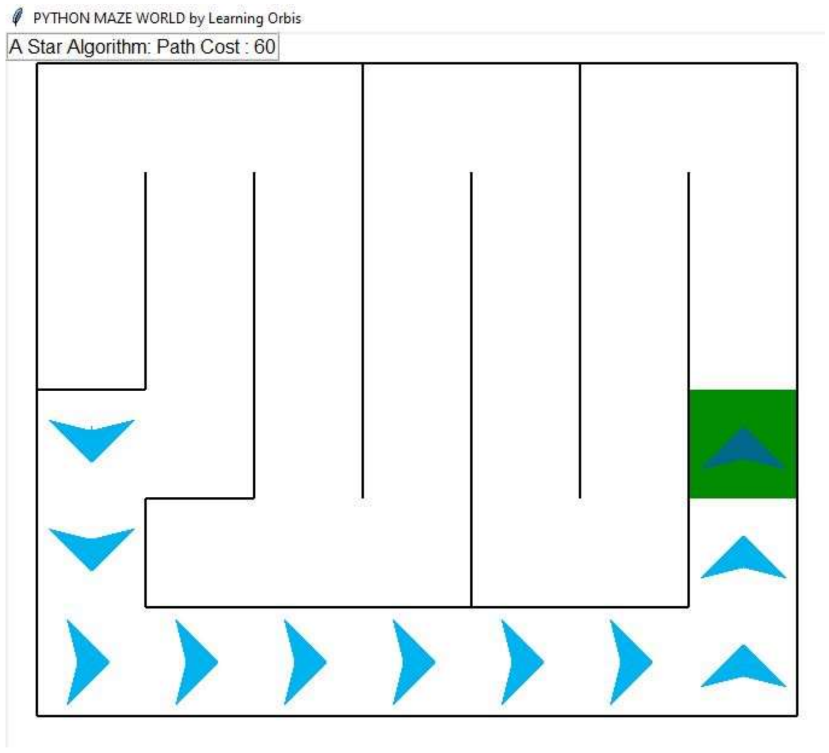
```
Current cell: (4, 1)

  -> E Neighbor: (4, 2) New cost: 8 Priority: 13.0

  -> S Neighbor: (5, 1) New cost: 3 Priority: 9.082762530298218

Current cell: (5, 1)

  -> S Neighbor: (6, 1) New cost: 6 Priority: 12.32455532033676

Current cell: (6, 1)

  -> E Neighbor: (6, 2) New cost: 14 Priority: 19.385164807134505

Current cell: (4, 2)

  -> N Neighbor: (3, 2) New cost: 11 Priority: 16.099019513592786

Current cell: (3, 2)

  -> N Neighbor: (2, 2) New cost: 14 Priority: 19.385164807134505

Current cell: (2, 2)

  -> N Neighbor: (1, 2) New cost: 17 Priority: 22.8309518948453

Current cell: (6, 2)

  -> E Neighbor: (6, 3) New cost: 22 Priority: 26.47213595499958

Current cell: (1, 2)

  -> E Neighbor: (1, 3) New cost: 25 Priority: 30.0
  -> W Neighbor: (1, 1) New cost: 20 Priority: 26.70820393249937
Current cell: (6, 3)
  -> E Neighbor: (6, 4) New cost: 30 Priority: 33.60555127546399
Current cell: (1, 1)
  -> S Neighbor: (2, 1) New cost: 23 Priority: 29.32455532033676
Current cell: (2, 1)
  -> S Neighbor: (3, 1) New cost: 26 Priority: 32.08276253029822
Current cell: (1, 3)
  -> S Neighbor: (2, 3) New cost: 28 Priority: 32.47213595499958
Current cell: (3, 1)
Current cell: (2, 3)
  -> S Neighbor: (3, 3) New cost: 31 Priority: 35.12310562561766
Current cell: (6, 4)
  -> E Neighbor: (6, 5) New cost: 38 Priority: 40.82842712474619
Current cell: (3, 3)
  -> S Neighbor: (4, 3) New cost: 34 Priority: 38.0
Current cell: (4, 3)
  -> S Neighbor: (5, 3) New cost: 37 Priority: 41.12310562561766
Current cell: (6, 5)
  -> E Neighbor: (6, 6) New cost: 46 Priority: 48.236067977499786
Current cell: (5, 3)
  -> E Neighbor: (5, 4) New cost: 45 Priority: 48.16227766016838
  -> W Neighbor: (5, 2) New cost: 40 Priority: 45.099019513592786
Current cell: (5, 2)
Current cell: (5, 4)

   -> N Neighbor: (4, 4) New cost: 48 Priority: 51.0

Current cell: (6, 6)

   -> E Neighbor: (6, 7) New cost: 54 Priority: 56.0

Current cell: (4, 4)

   -> N Neighbor: (3, 4) New cost: 51 Priority: 54.16227766016838

Current cell: (3, 4)

   -> N Neighbor: (2, 4) New cost: 54 Priority: 57.60555127546399

Current cell: (6, 7)

   -> N Neighbor: (5, 7) New cost: 57 Priority: 58.0

Current cell: (2, 4)

   -> N Neighbor: (1, 4) New cost: 57 Priority: 61.242640687119284

Current cell: (5, 7)

   -> N Neighbor: (4, 7) New cost: 60 Priority: 60.0

Current cell: (4, 7)


A* Algorithm:

Path: [(5, 1), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (4, 7)]

Path Cost (Excluding Heuristics): 60

Number of Nodes Expanded: 28

Time Taken: 0.0020444393157958984
```

Output Explanation

- Each line of the output corresponds to the algorithm's exploration of a cell and its neighbors.
- A priority queue ensures that the node with the lowest total cost/priority (path cost from start to the node + heuristic cost to the goal) is selected for expansion. This allows A* to explore the most promising nodes first, potentially leading to a more efficient search.
- Current cell: (4, 1)
    - ➔ E Neighbor: (4, 2) New cost: 8 Priority: 13.0
    - ➔ S Neighbor: (5, 1) New cost: 3 Priority: 9.082762530298218
- For example, the line Current cell: (4, 1) -> E Neighbor: (4, 2) New cost: 8 Priority: 13.0 indicates that moving east from cell (4, 1) to cell (4, 2) has a new cost of 8 and a priority of 13.0.
- Similarly (4,1) -> S Neighbor: (5, 1) New cost: 3 Priority: 9.082762530298218 indicates that moving south from cell (4, 1) to cell (5,1) has a new cost of 3 and a priority of 9.082762530298218.
- Since Priority for (5,1) is less, it is enqueued in the priority queue. The algorithm continues exploring and updating costs and priorities until it reaches the goal cell. As nodes are added to the priority queue, they are inserted in such a way that the node with the lowest total cost is always at the front of the queue. This ensures that nodes are expanded in the order that is most likely to lead to the goal. If the cost to reach a node is updated during the search (e.g., if a better path to a node is discovered), the priority queue can efficiently reorder the nodes based on the updated costs, ensuring that the search remains optimal. Using a priority queue allows A* to explore the search space in a memory-efficient manner, as it only needs to store a subset of nodes at any given time (those that are candidates for expansion).

The algorithm successfully finds the optimal path to the goal cell (4, 7) with a total cost of 60, as indicated by the output Total cost of the path: 60.

Path Sequence: (4,1) -> (5,1) -> (6,1) -> (6,2) -> (6,3) -> (6,4) -> (6,5) -> (6,6) -> (6,7) -> (5,7) -> (4,7)


PYTHON MAZE WORLD by Learning Orbis
A Star Algorithm: Path Cost : 60

Hill Climbing Algorithm

Initialization:

```
def hillClimbing(m):
    start = (4, 1)
    goal = (4, 7)
    current = start
    path = [current]
    total_path_cost = 0
    visited = set()
    i = 0
    nodes_expanded = 0

    start_time = time.time()
```

- The algorithm starts at cell (4, 1) and the goal is to reach cell (4, 7).
- current is initialized to the start cell, and path contains the current path.
- visited is a set to keep track of visited cells.
- nodes_expanded to keep track nodes expanded by the algorithm to get the optimal path.

Exploration:

```python
print("Start State:", current)
while current != goal:
    nodes_expanded += 1
    print(f"Iteration: ", i)
    visited.add(current)
    neighbors = []
    for d in 'ESNW':
        if m.maze_map[current][d]:
            neighbor = getNeighbor(current, d)
            cost = 3  # Base transition cost
            if d == 'E':
                cost += 5  # Additional cost for moving east (sunlight
penalty)
            total_cost = cost + heuristic(neighbor, goal)  # Total cost of
reaching the neighbor
            neighbors.append((neighbor, total_cost, cost, d))

    print("Current:", current, "Neighbors:", neighbors)
    neighbors.sort(key=lambda x: x[1])  # Sort neighbors based on total cost
    print("Sorted Neighbors:", neighbors)
    for neighbor in neighbors:
        if neighbor[0] in visited:
            neighbors.remove(neighbor)
            break  # Exit the loop after removing the neighbor
    print("Sorted Neighbors after removing duplicates since backtracking not
allowed:", neighbors)

    if len(neighbors) == 0:
        print("No valid neighbors. Stuck at:", current)
        break

    next_neighbor, total_cost, path_cost, direction = neighbors[0]

    current = next_neighbor
    path.append(current)
    total_path_cost += path_cost  # Update path cost with the correct cost
value, including the penalty for moving east
    i = i+1
    print("Next State:", current, "Path Cost:", total_path_cost, "Total
Cost:", total_cost, "Checked Neighbor:",
          direction)
    print("\n")

end_time = time.time()
time_taken = end_time - start_time

return path, total_path_cost, nodes_expanded, time_taken
```

The algorithm explores the neighboring cells of the current cell, considering only valid moves.

- For each neighboring cell, it calculates the total cost to reach that cell and adds it to the neighbors list.

- The algorithm prints information about the current cell, its neighbors, and the selected neighbor.
- **Sorting Neighbors:** The algorithm sorts the neighbors based on their total cost in ascending order.
- **Backtracking Prevention:** To prevent backtracking, the algorithm removes any neighbors that have already been visited from the neighbors list.
- If all neighbors have been visited or no valid neighbors are found, the algorithm prints a message and breaks the loop.
- **Selecting Next State:** The algorithm selects the neighbor with the lowest total cost as the next state to move to.
- It updates the current cell, adds it to the path, and updates the total_path_cost with the cost of the selected neighbor.
- **Iteration and Termination:** The algorithm continues iterating until it reaches the goal cell or gets stuck (no valid neighbors).
- Each iteration is labeled with a number for reference.

Driver script

```
if __name__ == '__main__':
    m = maze(6, 7)
    m.CreateMaze(loadMaze='Assignment_5.csv', x=4, y=7, theme=COLOR.light)
    hpath, htotal_path_cost, hill_climbing_nodes_expanded,
hill_climbing_time_taken = hillClimbing(m)

    a1 = agent(m, 4, 1, footprints=True, color=COLOR.blue, filled=True,
shape='arrow')
    m.tracePath({a1: hpath})

    print("\nHill Climbing Algorithm:")
    print("Path:", hpath)
    print("Path Cost:", htotal_path_cost)
    print("Number of Nodes Expanded:", hill_climbing_nodes_expanded)
    print("Time Taken:", hill_climbing_time_taken)

    l = textLabel(m, 'Hill Climbing Total Path Cost', htotal_path_cost)

    m.run()
```

- Sets up the maze environment such that a maze object is created with dimensions 6 rows by 7 columns.
- generates the maze using data loaded from a CSV file named 'Assignment_5.csv'. Goal state is set at position (4, 7) in the maze, and the theme of the maze is set to light colors.
- Apply the Hill Climbing algorithm to find an optimal path in the maze.

- Creates an agent to navigate through the maze based on the path found. An agent object is created and placed at position (4, 1) in the maze. It leaves footprints as it moves, represented by blue arrows.
- Prints out information about the path found by the Hill Climbing algorithm.
- Displays the maze with the optimal path traced.

Output:

Start State: (4, 1)

Iteration: 0

Current: (4, 1) Neighbors: [((4, 2), 13.0, 8, 'E'), ((5, 1), 9.082762530298218, 3, 'S')]

Sorted Neighbors: [((5, 1), 9.082762530298218, 3, 'S'), ((4, 2), 13.0, 8, 'E')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((5, 1), 9.082762530298218, 3, 'S'), ((4, 2), 13.0, 8, 'E')]

Next State: (5, 1) Path Cost: 3 Total Cost: 9.082762530298218 Checked Neighbor: S

Iteration: 1

Current: (5, 1) Neighbors: [((6, 1), 9.32455532033676, 3, 'S'), ((4, 1), 9.0, 3, 'N')]

Sorted Neighbors: [((4, 1), 9.0, 3, 'N'), ((6, 1), 9.32455532033676, 3, 'S')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((6, 1), 9.32455532033676, 3, 'S')]

Next State: (6, 1) Path Cost: 6 Total Cost: 9.32455532033676 Checked Neighbor: S

Iteration: 2

Current: (6, 1) Neighbors: [((6, 2), 13.385164807134505, 8, 'E'), ((5, 1), 9.082762530298218, 3, 'N')]

Sorted Neighbors: [((5, 1), 9.082762530298218, 3, 'N'), ((6, 2), 13.385164807134505, 8, 'E')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((6, 2), 13.385164807134505, 8, 'E')]

Next State: (6, 2) Path Cost: 14 Total Cost: 13.385164807134505 Checked Neighbor: E

Iteration: 3

Current: (6, 2) Neighbors: [((6, 3), 12.47213595499958, 8, 'E'), ((6, 1), 9.32455532033676, 3, 'W')]

Sorted Neighbors: [((6, 1), 9.32455532033676, 3, 'W'), ((6, 3), 12.47213595499958, 8, 'E')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((6, 3), 12.47213595499958, 8, 'E')]

Next State: (6, 3) Path Cost: 22 Total Cost: 12.47213595499958 Checked Neighbor: E

Iteration: 4

Current: (6, 3) Neighbors: [((6, 4), 11.60555127546399, 8, 'E'), ((6, 2), 8.385164807134505, 3, 'W')]

Sorted Neighbors: [((6, 2), 8.385164807134505, 3, 'W'), ((6, 4), 11.60555127546399, 8, 'E')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((6, 4), 11.60555127546399, 8, 'E')]

Next State: (6, 4) Path Cost: 30 Total Cost: 11.60555127546399 Checked Neighbor: E

Iteration: 5

Current: (6, 4) Neighbors: [((6, 5), 10.82842712474619, 8, 'E'), ((6, 3), 7.47213595499958, 3, 'W')]

Sorted Neighbors: [((6, 3), 7.47213595499958, 3, 'W'), ((6, 5), 10.82842712474619, 8, 'E')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((6, 5), 10.82842712474619, 8, 'E')]

Next State: (6, 5) Path Cost: 38 Total Cost: 10.82842712474619 Checked Neighbor: E

Iteration: 6

Current: (6, 5) Neighbors: [((6, 6), 10.23606797749979, 8, 'E'), ((6, 4), 6.60555127546399, 3, 'W')]

Sorted Neighbors: [((6, 4), 6.60555127546399, 3, 'W'), ((6, 6), 10.23606797749979, 8, 'E')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((6, 6), 10.23606797749979, 8, 'E')]

Next State: (6, 6) Path Cost: 46 Total Cost: 10.23606797749979 Checked Neighbor: E

Iteration: 7

```
Current: (6, 6) Neighbors: [((6, 7), 10.0, 8, 'E'), ((6, 5), 5.82842712474619, 3, 'W')]

Sorted Neighbors: [((6, 5), 5.82842712474619, 3, 'W'), ((6, 7), 10.0, 8, 'E')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((6, 7), 10.0, 8, 'E')]

Next State: (6, 7) Path Cost: 54 Total Cost: 10.0 Checked Neighbor: E
```

Iteration: 8

```
Current: (6, 7) Neighbors: [((5, 7), 4.0, 3, 'N'), ((6, 6), 5.23606797749979, 3, 'W')]

Sorted Neighbors: [((5, 7), 4.0, 3, 'N'), ((6, 6), 5.23606797749979, 3, 'W')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((5, 7), 4.0, 3, 'N')]

Next State: (5, 7) Path Cost: 57 Total Cost: 4.0 Checked Neighbor: N
```

Iteration: 9

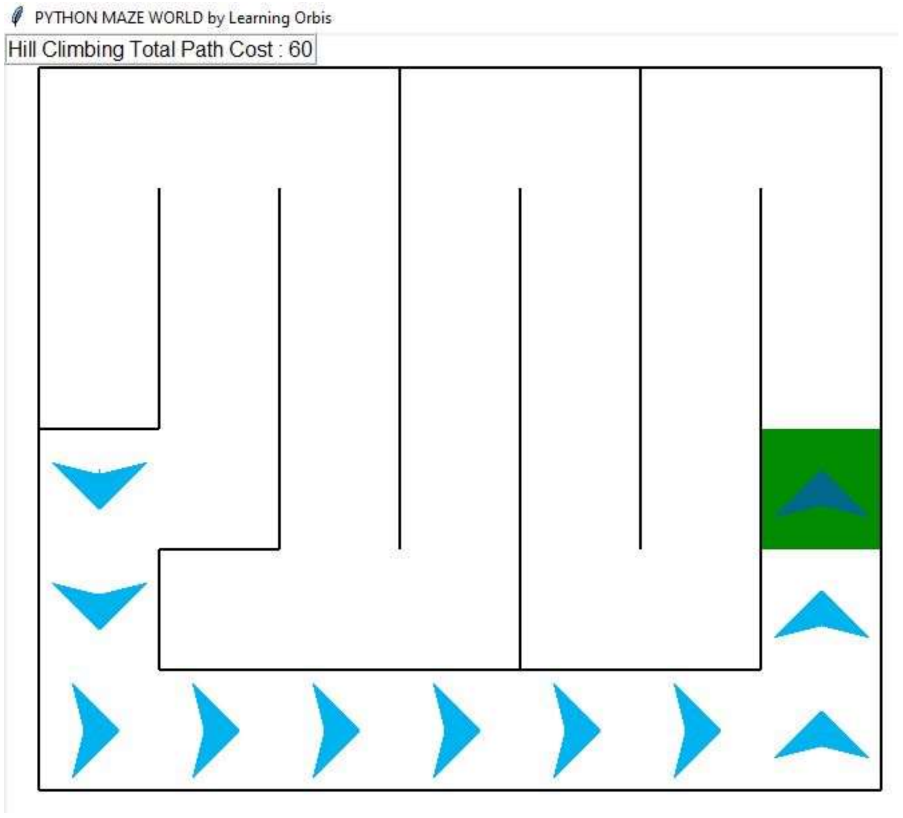
```
Current: (5, 7) Neighbors: [((6, 7), 5.0, 3, 'S'), ((4, 7), 3.0, 3, 'N')]

Sorted Neighbors: [((4, 7), 3.0, 3, 'N'), ((6, 7), 5.0, 3, 'S')]

Sorted Neighbors after removing duplicates since backtracking not allowed: [((4, 7), 3.0, 3, 'N')]

Next State: (4, 7) Path Cost: 60 Total Cost: 3.0 Checked Neighbor: N


Hill Climbing Algorithm:

Path: [(5, 1), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (4, 7)]

Path Cost (Excluding Heuristics): 60

Number of Nodes Expanded: 10

Time Taken: 0.0009975433349609375
```

Output Explanation

- Each iteration shows the current cell, its neighbors, the sorted neighbors, and the selected neighbor for movement.
- For example, Current: (4, 1) Neighbors: [((4, 2), 13.0, 8, 'E'), ((5, 1), 9.082762530298218, 3, 'S')] indicates that the current cell is (4, 1), and it has two neighbors: (4, 2) to the east with a total cost of 13.0 and (5, 1) to the south with a total cost of 9.082762530298218. Selects the least of total cost and explores the neighbor nodes. Since (5,1) has less cost, it explores (5,1)
  Current: (5, 1) Neighbors: [((6, 1), 9.32455532033676, 3, 'S'), ((4, 1), 9.0, 3, 'N')]
  Sorted Neighbors: [((4, 1), 9.0, 3, 'N'), ((6, 1), 9.32455532033676, 3, 'S')]
  Sorted Neighbors after removing duplicates since backtracking not allowed: [((6, 1), 9.32455532033676, 3, 'S')]
  Next State: (6, 1) Path Cost: 6 Total Cost: 9.32455532033676 Checked Neighbor: S

The algorithm successfully finds the optimal path to the goal cell (4, 7) with a total cost of 60, as indicated by the output Next State: (4, 7) Path Cost: 60 Total Cost: 3.0 Checked Neighbor: N.

Sequence: (4,1) -> (5,1) -> (6,1) -> (6,2) -> (6,3) -> (6,4) -> (6,5) -> (6,6) -> (6,7) -> (5,7) -> (4,7)


PYTHON MAZE WORLD by Learning Orbis
Hill Climbing Total Path Cost : 60

Conclusion

1. **A* Algorithm**:

   Path: [(5, 1), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (4, 7)]

   Path cost (excluding heuristic): 60

   Number of Nodes Expanded: 28

   Time Taken: 0.0020444393157958984

2. **Hill Climbing Algorithm**:

   Path: [(5, 1), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (4, 7)]

   Path Cost (excluding heuristic): 60

   Number of Nodes Expanded: 10

   Time Taken: 0.0009975433349609375

Considering the algorithms to evaluate the best path to reach the goal state for a system it is observed as follows:

- **Path**: Both algorithms found the same path from the start state to the goal state, which is [(5, 1), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (4, 7)]. This indicates that both algorithms were able to navigate through the maze successfully.
- **Total Cost of the Path**: The total cost of the path found by both algorithms is 60. This means that both algorithms were able to find the same optimal path and calculate the assigned cost to the path, considering the movement costs and any additional penalties for certain actions.
- **Number of Nodes Expanded**: The A* algorithm expanded **28** nodes, while the Hill Climbing algorithm expanded **10** nodes. This shows that the Hill Climbing algorithm explored fewer nodes in its search for the goal compared to the A* algorithm.
- **Time Taken**: The A* algorithm took **0.0020444393157958984** seconds, while the Hill Climbing algorithm took **0.0009975433349609375** seconds. The Hill Climbing algorithm was faster in this case, requiring less time to find the optimal path compared to the A* algorithm. Since A* algorithm needs to expand more nodes, more time is consumed.

Code:

I have added excel sheet which contains the maze configuration and the implementation python code for A* and Hill climbing algorithms are in the following drive link. Additionally, I have added the working video/demo of the code.

A* Algorithm Code:

https://drive.google.com/file/d/1aUxy7i2RtgM2GaSK0GSJycy-I1_4KIOJ/view?usp=sharing

Hill Climbing Algorithm Code:
https://drive.google.com/file/d/1oiCgG9oicxLRrdq5iKmAbtofOcCqR8uJ/view?usp=sharing

Excel Sheet:

https://drive.google.com/file/d/1PBZTbyOh7zCPGFbSmF-Ug873RQnNONI5/view?usp=sharing

Folder Link:
https://drive.google.com/drive/folders/1IZe_LfK0tJaYIFeoWIl1uRokjlHrOAZS?usp=drive_link

Note: I would please request you to go through a demo. Thanks!

Video Link:
https://drive.google.com/file/d/1z5CXzOHgSWtes0B42kDrQny6iLlEtZ_p/view?usp=sharing

References

- https://www.javatpoint.com/ai-informed-search-algorithms
- https://www.geeksforgeeks.org/search-algorithms-in-ai/
- https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/
- https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_hill_climbing.htm
- Stuart Russell and Peter Norvig, “Artificial Intelligence – A Modern Approach”, Pearson Education, Third Edition.
- Elaine Rich and Kevin Knight, “Artificial Intelligence”, Tata McGraw Hill Publishing Company, New Delhi, 2003