SUBMITTED TO:
DR. DILSHAD SABIR

SUBMITTED BY:

DANISH ZULFIQAR

REGISTRATION NO.:

FA21-BEE-042

SEMESTER-SECTION:

5- B

COURSE:

Artificial Intelligence
EEE 462

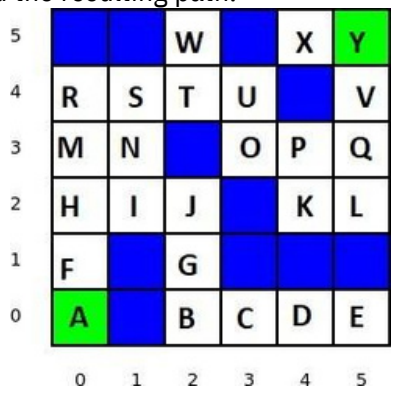
Lab Report 05

BS Electrical Engineering

Lab Activities:

Activity 1:

Consider a maze as shown below. Each empty tile represents a separate node in the graph, while the walls are represented by blue tiles. Your starting node is A and the goal is to reach Y. Implement an A* search to find the resulting path.



Solution 1:

```
import math

class Node:
    def __init__(self, state, parent, actions, totalCost, heuristic):
self.state = state
self.parent = parent
self.actions = actions
self.totalCost = totalCost
self.heuristic = heuristic


def findMin(frontier):
minV = math.inf
node = ''
for i in frontier:
if minV > frontier[i][1]:
minV = frontier[i][1]
node = i
return node


def actionSequence(graph, initialState, goalState):
solution = [goalState]
currentParent = graph[goalState].parent
while currentParent != None:
solution.append(currentParent)
currentParent = graph[currentParent].parent
solution.reverse()
return solution


def Astar():
initialState = 'A'
```

```
 goalState = 'Y'
graph = {
'A': Node('A', None, [('F', 1)], 0, (0, 0)),
'B': Node('B', None, [('G', 1), ('C', 1)], 0, (2, 0)),
'C': Node('C', None, [('B', 1), ('D', 1)], 0, (3, 0)),
'D': Node('D', None, [('C', 1), ('E', 1)], 0, (4, 0)),
'E': Node('E', None, [('D', 1)], 0, (5, 0)),
'F': Node('F', None, [('A', 1), ('H', 1)], 0, (0, 1)),
'G': Node('G', None, [('B', 1), ('J', 1)], 0, (2, 1)),
'H': Node('H', None, [('F', 1), ('I', 1), ('M', 1)], 0, (0, 2)),
'I': Node('I', None, [('H', 1), ('J', 1), ('N', 1)], 0, (1, 2)),
'J': Node('J', None, [('G', 1), ('I', 1)], 0, (2, 2)),
'K': Node('K', None, [('L', 1), ('P', 1)], 0, (4, 2)),
'L': Node('L', None, [('K', 1), ('Q', 1)], 0, (5, 2)),
'M': Node('M', None, [('H', 1), ('N', 1), ('R', 1)], 0, (0, 3)),
'N': Node('N', None, [('I', 1), ('M', 1), ('S', 1)], 0, (1, 3)),
'O': Node('O', None, [('P', 1), ('U', 1)], 0, (3, 3)),
'P': Node('P', None, [('O', 1), ('Q', 1)], 0, (4, 3)),
'Q': Node('Q', None, [('L', 1), ('P', 1), ('V', 1)], 0, (5, 3)),
'R': Node('R', None, [('M', 1), ('S', 1)], 0, (0, 4)),
'S': Node('S', None, [('N', 1), ('R', 1), ('T', 1)], 0, (1, 4)),
'T': Node('T', None, [('S', 1), ('U', 1), ('W', 1)], 0, (2, 4)),
'U': Node('U', None, [('O', 1), ('T', 1)], 0, (3, 4)),
'V': Node('V', None, [('Q', 1), ('Y', 1)], 0, (5, 4)),
'W': Node('W', None, [('T', 1)], 0, (2, 5)),
'X': Node('B', None, [('Y', 1)], 0, (4, 5)),
'Y': Node('Y', None, [('V', 1), ('X', 1)], 0, (5, 5)),
}
frontier = dict()
heuristicCost = math.sqrt(((graph[goalState].heuristic[0] -
graph[initialState].heuristic[0]) \
* 2) + ((graph[goalState].heuristic[1] -
graph[initialState].heuristic[1]) * 2))
frontier[initialState] = (None, heuristicCost)
explored = dict()
while len(frontier) != 0:
currentNode = findMin(frontier)
del frontier[currentNode]
if graph[currentNode].state == goalState:
return actionSequence(graph, initialState, goalState)

        heuristicCost = math.sqrt(((graph[goalState].heuristic[0] -
graph[currentNode].heuristic[0]) \
                                   * 2) + ((graph[goalState].heuristic[1] -
graph[currentNode].heuristic[1]) * 2))
currentCost = graph[currentNode].totalCost
        explored[currentNode] = (graph[currentNode].parent, heuristicCost +
currentCost)
for child in graph[currentNode].actions:
currentCost = child[1] + graph[currentNode].totalCost
heuristicCost = math.sqrt(((graph[goalState].heuristic[0] -
graph[child[0]].heuristic[0]) \
* 2) +
((graph[goalState].heuristic[1] - graph[child[0]].heuristic[1]) * 2))
if child[0] in explored:
if graph[child[0]].parent == currentNode or child[0] ==
initialState or \
explored[child[0]][1] <= currentCost +
heuristicCost:
continue
if child[0] not in frontier:
```

```
 graph[child[0]].parent = currentNode
graph[child[0]].totalCost = currentCost
                frontier[child[0]] = (graph[child[0]].parent, currentCost +
heuristicCost)
else:
if frontier[child[0]][1] < currentCost + heuristicCost:
graph[child[0]].parent = frontier[child[0]][0]
graph[child[0]].totalCost = frontier[child[0]][1] -
heuristicCost
else:
frontier[child[0]] = (currentNode, currentCost +
heuristicCost)
graph[child[0]].parent = frontier[child[0]][0]
graph[child[0]].totalCost = currentCost


solution = Astar()
print(solution)
```

Output:

```
"C:\Users\FaT\Desktop\New folder (2)\AI\pythonProject2\venv\Scripts\python.exe"
['A', 'F', 'H', 'M', 'R', 'S', 'T', 'U', 'O', 'P', 'Q', 'V', 'Y']

Process finished with exit code 0
```

Activity 2:

For the graph in previous activity, imagine node A as starting node and your goal is to reach Y. Apply hill climbing and see how closer you can get to your destination.

Solution:

```
import math
from mimetypes import init
from multiprocessing import current_process

class Node:
    def __init__(self, state, parent, actions, heuristic, totalCost):
self.state=state
self.parent=parent
self.actions=actions
self.totalCost=totalCost
self.heuristic=heuristic

def hillClimbing():
initialState = 'A'
goalState = 'Y'

graph = {
'A': Node('A', None, [('F', 1)], (0,0), 0),
        'B': Node('B', None, [('G', 1), ('C', 1)], (2,0), 0),
        'C': Node('C', None, [('B', 1), ('D', 1)], (3,0), 0),
```

```
 'D': Node('D', None, [('C', 1), ('E', 1)], (4,0), 0),
'E': Node('E', None, [('D', 1)], (5,0), 0),
'F': Node('F', None, [('A', 1), ('H', 1)], (0,1), 0),
'G': Node('G', None, [('B', 1), ('J', 1)], (2,1), 0),
'H': Node('H', None, [('F', 1), ('I', 1), ('M', 1)], (0,2), 0),
'I': Node('I', None, [('H', 1), ('J', 1), ('N', 1)], (1,2), 0),
'J': Node('J', None, [('G', 1), ('I', 1)], (2,2), 0),
'K': Node('K', None, [('L', 1), ('P', 1)], (4,2), 0),
'L': Node('L', None, [('K', 1), ('Q', 1)], (5,2), 0),
'M': Node('M', None, [('H', 1), ('N', 1), ('R', 1)], (0,3), 0),
'N': Node('N', None, [('I', 1), ('M', 1), ('S', 1)], (1,3), 0),
'O': Node('O', None, [('P', 1), ('U', 1)], (3,3), 0),
'P': Node('P', None, [('O', 1), ('Q', 1)], (4,3), 0),
'Q': Node('Q', None, [('L', 1), ('P', 1), ('V', 1)], (5,3), 0),
'R': Node('R', None, [('M', 1), ('S', 1)], (0,4), 0),
'S': Node('S', None, [('N', 1), ('R', 1), ('T', 1)], (1,4), 0),
'T': Node('T', None, [('S', 1), ('U', 1), ('W', 1)], (2,4), 0),
'U': Node('U', None, [('O', 1), ('T', 1)], (3,4), 0),
'V': Node('V', None, [('Q', 1), ('Y', 1)], (5,4), 0),
'W': Node('W', None, [('T', 1)], (2,5), 0),
'X': Node('X', None, [('Y', 1)], (4,5), 0),
'Y': Node('Y', None, [('V', 1), ('X', 1)], (5,5), 0)
}

parentNode=initialState
parentCost = math.sqrt(((graph[goalState].heuristic[0] -
graph[initialState].heuristic[0])**2) + ((graph[goalState].heuristic[1] -
graph[initialState].heuristic[1])**2))
explored=[]
solution=[]
minChildCost = parentCost - 1
while parentNode != goalState:
bestNode = parentNode
minChildCost = parentCost
explored.append(parentNode)
for child in graph[parentNode].actions:
if child[0] not in explored:
childCost = math.sqrt(((graph[goalState].heuristic[0] -
graph[child[0]].heuristic[0])**2) + ((graph[goalState].heuristic[1] -
graph[child[0]].heuristic[1])**2))
if childCost < minChildCost:
bestNode = child[0]
minChildCost = childCost
if bestNode == parentNode:
break
else:
parentNode = bestNode
parentCost = minChildCost
solution.append(parentNode)
return solution

solution = hillClimbing()

print(solution)
```

Output:

```
"C:\Users\FaT\Desktop\New folder (2)\AI\pythonProject2\venv\Scripts\python.exe"
['F', 'H', 'I', 'J']

Process finished with exit code 0
```

Home Activities:

1) Stage v (verify)

Activity 1:

Since hill climbing can get stuck as a local minima as we saw in the previous activity, apply random restart hill climbing to get to your goal Y. How many restarts do you need in this case?

Solution:

```
def print_maze(maze):
for row in maze:
print("".join(row))
def load_maze(filename):
file = open(filename, "r")
maze = []
start_pos = (0, 0)
end_pos = (0, 0)
row_num = 0
for line in file:
row = tuple(line.strip().split(","))
if 'S' in row:
start_col = row.index('S')
start_pos = (row_num, start_col)
if 'E' in row:
end_col = row.index('E')
end_pos = (row_num, end_col)
maze.append(row)
row_num += 1
file.close()
return tuple(maze), start_pos, end_pos

def solvable_maze(maze, path, end):
row, col = path[-1][0], path[-1][1]
# Check if the row is within the bounds of the maze
if row < 0 or row >= len(maze):
path.pop()
return False
# Check if the col is within the bounds of the maze
if col < 0 or col >= len(maze[0]):
path.pop()
return False
# if within bounds, extract the position
pos = maze[row][col]
# check if position is valid and not a wall X
if pos not in ["O", "S", "E"]:
path.pop()
return False
# check if position is at the exit
elif row == end[0] and col == end[1]:
return True
            # check if the position is already present in the path
elif (row, col) in path[0:-1]:
```

```
 path.pop()
return False
else:
path.append((row-1, col))
result = solvable_maze(maze, path, end)
if result:
return True
# move one step right
path.append((row, col+1))
result = solvable_maze(maze, path, end)
if result:
return True
# move one step down
path.append((row+1, col))
result = solvable_maze(maze, path, end)
if result:
return True
# move one step left
path.append((row, col-1))
result = solvable_maze(maze, path, end)
if result:
return True
        # if no path found then remove the current position from path
if not result:
path.pop()
return result


maze, start_pos, exit_pos = load_maze("solvable_maze.csv")
print("Start Coordinate:", start_pos)
print("Exit Coordinate:", exit_pos)
print_maze(maze)
path = [start_pos]
end = exit_pos
solvable = solvable_maze(maze, path, end)
print("Solution Found:", solvable)
print("Solution Path:", path)
```

Critical Analysis and Conclusion:

In this lab we learned Hill Climbing is a heuristic search used for mathematical optimization problems in the field of Artificial Intelligence.

Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may not be the global optimal maximum.

- In the above definition, mathematical optimization problems imply that hill-climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs. Example-Travelling salesman problem where we need to minimize the distance traveled by the salesman.
- 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in a reasonable time.
- A heuristic function is a function that will rank all the possible alternatives at any branching step in the search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.