LAB # 5

Statement Purpose:

This lab will introduce students to heuristic and local search methods. In particular we will implement a particular variant of heuristic search known as A* search. Students will also be introduced to local search, a particular instance of local search method known as hill climbing will also be implemented.

Activity Outcomes:

This lab teaches you the following topics:

- How to represent problems in terms of state space graph
- How to find a solution of those problems using A* search.
- How to find a solution of those problems using hill climbing.

Instructor Note:

As pre-lab activity, read Chapters 3 and 4 from the book (Artificial Intelligence, A Modern Approach by Peter Norvig, 3rd edition) to know the basics of search algorithms.

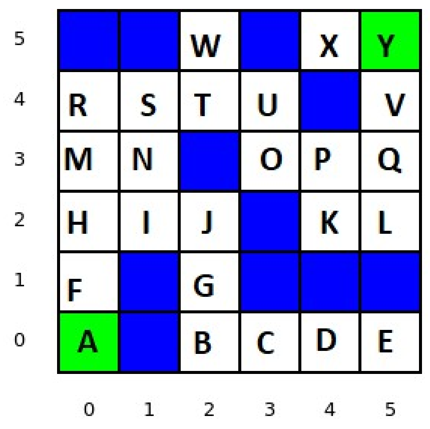
1) Stage J (Journey) Introduction:

So fat we have studied uninformed search strategies. There can also be occasions where we are given some extra information related to a particular goal in the form of heuristics. We can use such heuristics in our search strategy. A particular form of this strategy known as A* search uses the total cost of a solution via a particular node as that node's evaluation criteria. We will see its implementation in detail. We will also see hill climbing which is an instance of local search strategy.

2) Stage a1 (apply) 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:

Since A* search needs a heuristic function that specifies the distance of each node with the goal, you can use Euclidean distance between a particular node and the goal node as your heuristic function of that particular node.

But first we need to modify the structure of Node class that will also include the heuristic distance of that node.

```
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
```

Next we modify the structure of dictionary that will save our graph. Here at each node, we also specify the coordinate location of each node. For implementation we will follow uniform cost implementation except for the fact that the total cost will include both the previous cost of reaching that node and the distance of the goal with that node.

```
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)], (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)
             }

    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)
        print(currentNode)
        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:
                [illegible]
                else:
                    frontier[child[0]]=(currentNode, currentCost + heuristicCost)
                    graph[child[0]].parent=frontier[child[0]][0]
                    graph[child[0]].totalCost=currentCost


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

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:

Instead of maintaining a fringe or a frontier to save the nodes that are to be explored, hill climbing just explores the best child of a given node, then explores the best grandchild of a particular child and so on and so forth.

```
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 hillClimbing():

    graph = {
            'A' : Node('A', None, [('F',1)], 0, (0,0)),
            'B' : Node('B', None, [('C',1), ('G',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, [('K',1), ('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), ('W',1), ('U',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('X', None, [('Y',1)], 0,(4,5)),
            'Y' : Node('Y', None, [('X',1), ('Y',1)], 0,(5,5))}

    initialState = 'A'
    goalState = 'Y'

    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)
```

This will give ['F', 'H', 'I', 'J'] as the solution, which means that the algorithm gets stuck at J and doesn't go further towards Y since the distance of G is greater than J. Remember that the path is not important in local search. The solution should be your last node (currently that is J which happens to be local maxima).