COEN 166 Artificial Intelligence

Lab Assignment #3 - Search

Name: Eva Stenberg **ID: 00001576722**

Problem 1 Breadth-First Search
Function 1 #
paste your code:

```
def breadthFirstSearch(problem):

    frontierList = util.Queue()                          #list that has been visited
    exploredList = []
    start = problem.getStartState()                      #get list of possible actions from current state
    root = (start, [], 0)
    frontierList.push(root)
    while (not frontierList.isEmpty()):                  #while the list is not empty
        current_state, actions, current_cost = frontierList.pop()
        if (current_state not in exploredList): #if current cell has not already been visited, add to list
            exploredList.append(current_state)
            if(problem.goalTest(current_state)):         #if current cell is goal state
                return actions
            else:
                for i in problem.getActions(current_state):      #traverses to next cell
                    new_actions = actions + [i]
                    new_cost = current_cost + problem.getCostOfActions(new_actions)
                    new_state = problem.getResult(current_state, i)
                    new_node = (new_state, new_actions, new_cost)
                    frontierList.push(new_node)
    return actions                                       #returns solution path
    util.raiseNotDefined()
```

Comment: I started by initializing frontier as a queue which is the list of nodes that have been visited. Next I created a list of possible actions from the current state and I implemented a while loop. While the loop is not empty, it checks if the cell has already been visited, and if not, adds it to the visited list. It then checks if it's the goal state, and if it is, it returns the actions taken to get to the goal state. If not, it creates a child node and goes back through the loop.

Function #2

```
def depthFirstSearch(problem):

    dfsStack = util.Stack()
    visited = []

    startState = problem.getStartState()
    S = Node(startState, None, None, 0)
    currentNode = S

    dfsStack.push(currentNode)
    actionList = dfsHelper(problem, visited, dfsStack)
    return actionList
    util.raiseNotDefined()

def dfsHelper(problem, visited, stack):
    currentNode = stack.pop()
    visited.insert(0, currentNode.state)

    if problem.goalTest(currentNode.state):                #found goal state
        actionList = []
        while currentNode.state != problem.getStartState():
            actionList.insert(0, currentNode.action)
            currentNode = currentNode.parent
        return actionList                                  #create a list of actions to get there and return it
    else:
        actions = problem.getActions(currentNode.state)     #find actions at current state
        for action in actions:                              #for each possible action
            nextState = problem.getResult(currentNode.state, action)
            if not nextState in visited:    #if it is not already visited, add and recursively call this
function
                nextCost = problem.getCost(currentNode.state, action)
                nextNode = Node(nextState, currentNode, action, nextCost)
                stack.push(nextNode)        #stack -- LIFO
                result = dfsHelper(problem, visited, stack)
                if not(result=="done"):     #branch without goal node
                    return result
        return "done"
```

Comment: The algorithm starts at the root node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it.

Problem 2 A* Search
Function 1 #
paste your code:

```
def aStarSearch(problem, heuristic): …
 node = problem.getStartState()                    #get list of possible actions from current state
   border = util.PriorityQueue()
   border.push(node, 0)
   visited = {}                                              #list of nodes that have been visited
   visited[node] = 0
   paths = {}
   paths[node] = []
   while not border.isEmpty():
      node = border.pop()
      if problem.goalTest(node):                  #checks if node is goal node and returns actions
         return paths[node]
      for action in problem.getActions(node):
         child = problem.getResult(node, action)          #creates child node and adds to actions
         cost_child = problem.getCostOfActions(paths[node] + [action]) + heuristic(child,
problem)  #find g + h
         if child not in visited or visited[child] > cost_child:      #repeats steps for the child node
            paths[child] = paths[node] + [action]
            visited[child] = cost_child
            border.push(child, cost_child)
   util.raiseNotDefined()
```

Comment: I defined a node to get the list of possible actions from the current state. aStarSearch does the same as breadth first search but it uses border and PriorityQueue to perform the actions of checking for goal state and traversing along until goal state is reached.

Function 2

```
def UniformCostSearch(problem):

   startingNode = problem.getStartState()
   if problem.goalTest(startingNode):
      return []
```

```
visited = []

pQueue = util.PriorityQueue()
#((coordinate/node , action to current node , cost to current node),priority)
pQueue.push((startingNode, [], 0), 0)

while not pQueue.isEmpty():

    currentNode, actions, prevCost = pQueue.pop()
    if currentNode not in visited:
        visited.append(currentNode)

        if problem.goalState(currentNode):
            return actions

        for nextNode, action, cost in problem.getActions(currentNode):
            newAction = actions + [action]
            priority = prevCost + cost
            pQueue.push((nextNode, newAction, priority),priority)
util.raiseNotDefined()
```

Comment: This function inserts the root note into the priority queue, and while the queue is not empty, it removes the element with the highest priority. It then checks if it's the goal node and if so, it prints the total cost. If not, it enqueues all the children of the current node to the priority queue.