COEN 166 Artificial Intelligence

Lab 3: Sample Submission

Name: Carlo Bilbao ID: 1608742

Problem 1 Breadth-First Search

Function 1

```
# paste your code:
def breadthFirstSearch(problem):
    #import time

    """
    Search the shallowest nodes in the search tree first.

    You are not required to implement this, but you may find it useful for Q5.
    """
    "*** YOUR CODE HERE ***"
    #Parameter 'problem' gives you all the information of the simulation,
    #taken from searchAgents.py as PositionSearchProblem
    #Queue taken from util.py

    #Defines all directions
    from game import Directions
    n = Directions.NORTH
    s = Directions.SOUTH
    e = Directions.EAST
    w = Directions.WEST

    #Setup, take information from parameter 'problem'
    #Node(state, parent, action, path_cost)
    #S = problem.getStartState()
    S = Node(problem.getStartState(), None, None, 0)
    n_curr = S #Initialize the current node to Start node
    #A1 = Node("A", S, "Up", 4)
    #B1 = Node("B", S, "Down", 3)
    #B2 = Node("B", A1, "Left", 6)
    #B1 == B2
    #Create a Queue
    from util import Queue
    queue = Queue()
    #Create an set of visited nodes
    visitedNodes = set()

    #Create an array of moves, which makes up the solution
```

```
solution = []

#NOTE: MAKE SURE TO PUSH STARTING NODE INTO QUEUE
queue.push(S)

#We have visited the starting node, so mark it as visited
#visitedNodes.append(S)

#Boolean to store if we visited this current node, initialized to false
#isVisted = False

#print ('Start State', n_curr)
print ('Start State', n_curr.state)
#print ('Start Actions', problem.getActions(n_curr))
print ('Start Actions', problem.getActions(n_curr.state))
print ('Queue Start', queue.list)

#Search the shallowest nodes in the search tree first
#We will run the simulation until the queue is fully emptied out (means we visited all)
while (queue.isEmpty() == False):
    #time.sleep(0.05)
    print ('Top Queue', queue.list)
    print ('Top Visited Nodes', visitedNodes)

    isVisited = False
    #TODO
    #Set the current node to the item first in line in the queue
    print ('Size of queue', len(queue.list))
    n_curr = queue.list.__getitem__(len(queue.list) - 1)
    #print('Current', n_curr)
    print('Current', n_curr.state)

    #Check if the current location has been visited using the visitedNodes array
    for x in visitedNodes:
        if (x == n_curr):
            isVisited = True
            queue.pop()
            #print('We have visited', n_curr)
            print('We have visited', n_curr.state)

    #Not visited, this means we expand the node
    #n_curr.state
    if (isVisited == False):
        visitedNodes.add(n_curr)
```

```
        #Mark the current node as visited by removing it from the queue
        queue.pop()

        #Expand the node
        #Run goal_test, if goal node, return solution
        #if (problem.goalTest(n_curr) == True):
        if (problem.goalTest(n_curr.state) == True):
            sol = []

            print ('Success')
            #TO DO
            #After finding the solution, reconstruct the path
            #Do this until we reach back to the starting state
            while (n_curr != S):
                sol.append(n_curr.state)

                #Add the current node to the solution set
                solution.append(n_curr)
                #Backtrack, go to the parent node
                print ('Parent', n_curr.state)
                n_curr = n_curr.parent
                print ('Solution', solution)
                print ('Sol', sol)
                print ('Queue length', len(sol))
            #Reverse the order of the array
            #solution = solution.reverse()
            solution.reverse()
            #sol = sol.reverse()
            #Add the starting element, for the backtracking calculations
            print ('S start state', S.state)
            sol.append(S.state)
            sol.reverse()
            num = len(sol)
            print ('Reverse Sol', sol)

            print ('VISITED NODES LENGTH: ', len(visitedNodes))

            #[s, s, w, s, w, w, s, w]
            #This array actually holds the actions for the simulation to work
            finalSolution = []

            #Find the action set based on the information
            for i in range(0, num - 1):
```

```
                #The vector will tell us the direction the agent moved
                #temp = sol[i + 1] - sol[i]
                temp = tuple(map(lambda i, j: i - j, sol[i + 1], sol[i]))
                print ('Temp calculation', temp)
                if (temp == (0, 1)):
                    finalSolution.append(n)
                elif (temp == (1, 0)):
                    finalSolution.append(e)
                elif (temp == (0, -1)):
                    finalSolution.append(s)
                elif (temp == (-1, 0)):
                    finalSolution.append(w)

            print('Final Solution', finalSolution)
            return finalSolution

        #Generate the node's successors (the adjacent ones), put into Frontier Queue

        #Get the list of actions
        #print ('Actions', problem.getActions(n_curr))
        #print ('Actions', problem.getActions(n_curr.state))
        #actions = problem.getActions(n_curr)
        actions = problem.getActions(n_curr.state)
        print ('Queue', queue.list)
        print ('Visited Nodes', visitedNodes)

        #For every action, we create a new node, add to Frontier Queue
        #TO DO
        for act in actions:
            #Create new node information
            #TO DO ------- n_new = (n_curr.getResult(STATE, ACTION), n_curr, act,
getCost(STATE, ACTION))
            #n_new = problem.getResult(n_curr, act)
            n_new = Node(problem.getResult(n_curr.state, act), n_curr, act,
problem.getCost(n_curr.state, act))
            print ('Act', act)
            print ('n_new', n_new)
            #n_new = Node(n_curr.getResult(n_curr, act), n_curr, act, n_curr.getCost(n_curr,
act))
            #queue.push(act)
            #ONLY ADD IF IT IS NOT CURRENTLY IN THE QUEUE
            #TODO CURRENTLY RUNNING INTO AN INFINITE LOOP, WEED OUT WHAT
YOU WANT TO ADD
            #SUGGESTION: FLIP THE LOGIC HERE..?
```

```
            inQueue = False
            print('Queue list', queue.list)
            for x in queue.list:
                print ('Item', x)
                if (n_new == x):
                    inQueue = True
            if (inQueue == False):
                queue.push(n_new)
        print('')



    util.raiseNotDefined()
```

Comment: explain how you modified/developed the above function/code…
We will run the whole simulation until the frontier queue is empty. At the beginning of the simulation, we have added the starting state, using the information from the problem. First, at the beginning of each loop iteration, we will take the first item in the queue and determine whether we should expand or not. I do this with a visitedNodes set (no duplicates). We will check if the current node is in the visitedNodes set, and if it is then we don't expand and take it out of our queue. If it is not visited, then expand.
We should add this current node to the visitedNodes set, and then mark the current node as visited by removing it from the queue.
If the current node is not the goal node, then we should expand it by generating the node's successors. We do this by getting the possible actions using problem.getActions and then storing that in an actions array variable. For each action, we find the new resulting node (after taking that action from the current node) and then append that to the queue (we will examine these nodes later). We have to make sure that these new resultant nodes are not currently in the queue, so we will check before adding them.
However, if the current node is indeed the goal node, we should display the solution. We will have to reconstruct the path, so starting at the current state (assumed to be the goal state at first), we will back-track, taking the node's parent information using n_curr.parent. We will add each and every node to a solution array, but since we are adding through backtracking, we will have to reverse the order to get the real solution.
Now, we need to determine the actual actions between each node in the solution array. To do this, I will compare each adjacent item-pair in the solution array. They are currently represented as positions (in a tuple), so if we subtract those positions, we will have a direction vector, which will tell us the action that was taken to get from the first position to the next. I used the lambda function in order to subtract a tuple from another tuple, which will get me the direction information as a unit vector. We compare the result, and determine which action was taken. Then we will add that action to the finalSolution array, which is a set of actions.

Problem 2 A* Search

Function 1

```
# paste your code:
def aStarSearch(problem, heuristic=nullHeuristic):
    #import time

    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    #Parameter 'problem' gives you all the information of the simulation,
    #taken from searchAgents.py as PositionSearchProblem
    #Queue taken from util.py

    #Defines all directions
    from game import Directions
    n = Directions.NORTH
    s = Directions.SOUTH
    e = Directions.EAST
    w = Directions.WEST

    #Setup, take information from parameter 'problem'
    #Node(state, parent, action, path_cost)
    #S = problem.getStartState()
    S = Node(problem.getStartState(), None, None, 0)
    n_curr = S #Initialize the current node to Start node
    #A1 = Node("A", S, "Up", 4)
    #B1 = Node("B", S, "Down", 3)
    #B2 = Node("B", A1, "Left", 6)
    #B1 == B2
    #Create a Priority Queue
    from util import PriorityQueue
    priorityqueue = PriorityQueue()
    #Create an set of visited nodes
    visitedNodes = set()

    #Used to call the manhattanHeuristic function
    import searchAgents
    searchAg = searchAgents

    #Create an array of moves, which makes up the solution
    solution = []

    #NOTE: MAKE SURE TO PUSH STARTING NODE INTO QUEUE
    #startPriorityCost = problem.getCost(S.state, None) + searchAg.manhattanHeuristic(S.state,
problem)
    startPriorityCost = 0 + searchAg.manhattanHeuristic(S.state, problem)
```

```
    priorityqueue.push(S, startPriorityCost)

    #We have visited the starting node, so mark it as visited
    #visitedNodes.append(S)

    #Boolean to store if we visited this current node, initialized to false
    #isVisted = False

    #print ('Start State', n_curr)
    print ('Start State', n_curr.state)
    #print ('Start Actions', problem.getActions(n_curr))
    print ('Start Actions', problem.getActions(n_curr.state))
    print ('Queue Start', priorityqueue.heap)

    #Search the shallowest nodes in the search tree first
    #We will run the simulation until the queue is fully emptied out (means we visited all)
    while (priorityqueue.isEmpty() == False):
        #time.sleep(0.05)
        print ('Top Queue', priorityqueue.heap)
        print ('Top Visited Nodes', visitedNodes)

        isVisited = False
        #TODO
        #Set the current node to the item first in line in the queue
        print ('Size of queue', len(priorityqueue.heap))
        #NOTE: If we are to retrieve a node from the priority queue, we must take the third
        #item in the tuple; goes like this: (g(n), pathCost, state)
        n_curr = priorityqueue.heap.__getitem__(0)[2]
        #print('Current', n_curr)
        print('Current', n_curr.state)

        #Check if the current location has been visited using the visitedNodes array
        for x in visitedNodes:
            if (x == n_curr):
                isVisited = True
                priorityqueue.pop()
                #print('We have visited', n_curr)
                print('We have visited', n_curr.state)

        #Not visited, this means we expand the node
        #n_curr.state
        if (isVisited == False):
            visitedNodes.add(n_curr)
```

```
        #Mark the current node as visited by removing it from the queue
        priorityqueue.pop()

        #Expand the node
        #Run goal_test, if goal node, return solution
        #if (problem.goalTest(n_curr) == True):
        if (problem.goalTest(n_curr.state) == True):
            sol = []

            print ('Success')
            #TO DO
            #After finding the solution, reconstruct the path
            #Do this until we reach back to the starting state
            while (n_curr != S):
                sol.append(n_curr.state)

                #Add the current node to the solution set
                solution.append(n_curr)
                #Backtrack, go to the parent node
                print ('Parent', n_curr.state)
                n_curr = n_curr.parent
                print ('Solution', solution)
                print ('Sol', sol)
                print ('Queue length', len(sol))
            #Reverse the order of the array
            #solution = solution.reverse()
            solution.reverse()
            #sol = sol.reverse()
            #Add the starting element, for the backtracking calculations
            print ('S start state', S.state)
            sol.append(S.state)
            sol.reverse()
            num = len(sol)
            print ('Reverse Sol', sol)

            #[s, s, w, s, w, w, s, w]
            #This array actually holds the actions for the simulation to work
            finalSolution = []

            #Find the action set based on the information
            for i in range(0, num - 1):
                #The vector will tell us the direction the agent moved
                #temp = sol[i + 1] - sol[i]
                temp = tuple(map(lambda i, j: i - j, sol[i + 1], sol[i]))
```

```
                print ('Temp calculation', temp)
                if (temp == (0, 1)):
                    finalSolution.append(n)
                elif (temp == (1, 0)):
                    finalSolution.append(e)
                elif (temp == (0, -1)):
                    finalSolution.append(s)
                elif (temp == (-1, 0)):
                    finalSolution.append(w)

            print('Final Solution', finalSolution)
            return finalSolution

        #Generate the node's successors (the adjacent ones), put into Frontier Queue

        #Get the list of actions
        #print ('Actions', problem.getActions(n_curr))
        #print ('Actions', problem.getActions(n_curr.state))
        #actions = problem.getActions(n_curr)
        actions = problem.getActions(n_curr.state)
        print ('Queue', priorityqueue.heap)
        print ('Visited Nodes', visitedNodes)

        #For every action, we create a new node, add to Frontier Queue
        #TO DO
        for act in actions:
            #Create new node information
            #TO DO ------- n_new = (n_curr.getResult(STATE, ACTION), n_curr, act,
getCost(STATE, ACTION))
            #n_new = problem.getResult(n_curr, act)
            #TODO!!
            #MAKE SURE TO DO CUMULATIVE COST HERE g(n)



            #To find actions, reconstruct the path
            solu = []
            #Create a temporary node to traverse the tree
            temp_curr = n_curr
            #Do this until we reach back to the starting state
            while (temp_curr != S):
                solu.append(temp_curr.state)

                #Add the current node to the solution set
```

```
                solution.append(temp_curr)
                #Backtrack, go to the parent node
                print ('Parent', temp_curr.state)
                temp_curr = temp_curr.parent
                print ('Solution', solution)
                print ('Sol', solu)
                print ('Queue length', len(solu))
            #Reverse the order of the array
            #solution = solution.reverse()
            solution.reverse()
            #sol = sol.reverse()
            #Add the starting element, for the backtracking calculations
            print ('S start state', S.state)
            solu.append(S.state)
            solu.reverse()
            num = len(solu)
            print ('Reverse Sol', solu)

            #[s, s, w, s, w, w, s, w]
            #This array actually holds the actions for the simulation to work
            finalActions = []

            #Find the action set based on the information
            for i in range(0, num - 1):
                #The vector will tell us the direction the agent moved
                #temp = sol[i + 1] - sol[i]
                temp = tuple(map(lambda i, j: i - j, solu[i + 1], solu[i]))
                print ('Temp calculation', temp)
                if (temp == (0, 1)):
                    finalActions.append(n)
                elif (temp == (1, 0)):
                    finalActions.append(e)
                elif (temp == (0, -1)):
                    finalActions.append(s)
                elif (temp == (-1, 0)):
                    finalActions.append(w)



            n_new = Node(problem.getResult(n_curr.state, act), n_curr, act,
problem.getCostOfActions(finalActions))
            print ('Act', act)
            print ('n_new', n_new)
```

```
            #n_new = Node(n_curr.getResult(n_curr, act), n_curr, act, n_curr.getCost(n_curr,
act))
            #queue.push(act)
            #ONLY ADD IF IT IS NOT CURRENTLY IN THE QUEUE
            #TODO CURRENTLY RUNNING INTO AN INFINITE LOOP, WEED OUT WHAT
YOU WANT TO ADD
            #SUGGESTION: FLIP THE LOGIC HERE..?
            inQueue = False
            print('Queue list', priorityqueue.heap)
            for x in priorityqueue.heap:
                print ('Item', x)
                #Access the third item in the tuple of the priorityqueue, which is a state
                if (n_new == x[2]):
                    inQueue = True
            if (inQueue == False):
                #The priority value is the path cost + g(n)
                #Priority cost
                #g(n) is cumulative cost
                #TODO!!
                #MAKE SURE TO DO CUMULATIVE COST g(n) + STEP COST f(n) + h(n)
                priorityCost = problem.getCostOfActions(finalActions) +
problem.getCost(n_new.state, act) + searchAg.manhattanHeuristic(n_new.state, problem)
                priorityqueue.push(n_new, priorityCost)
        print('')

    util.raiseNotDefined()
```

Comment: explain how you modified/developed the above function/code…

We will run the whole simulation until the frontier queue is empty. At the beginning of the simulation, we have added the starting state, using the information from the problem. First, at the beginning of each loop iteration, we will take the first item in the queue with the highest priority and determine whether we should expand or not. Each item in the queue is not just a single node anymore (it is a tuple). So, we will have to access the third item in the tuple, which is the node that we wanted (n_curr = priorityqueue.heap.getitem_(0)[2]). I do this with a visitedNodes set (no duplicates). We will check if the current node is in the visitedNodes set, and if it is then we don't expand and take it out of our queue. If it is not visited, then expand. We should add this current node to the visitedNodes set, and then mark the current node as visited by removing it from the queue.

If the current node is not the goal node, then we should expand it by generating the node's successors. We do this by getting the possible actions using problem.getActions and then storing that in an actions array variable. For each action, we find the new resulting node (after taking that action from the current node) and then append that to the queue (we will examine these nodes later). We have to make sure that these new resultant nodes are not currently in the queue, so we will check before adding them. Since each item in the queue for this problem

is not just a single node (we have costs now associated with each node), we will access the state location information by referencing the third item of each tuple, as you can see in n_new == x[2]. We are comparing here the current node and seeing if it is currently in the queue. If it is not in the queue, then add it. The important thing to note is that we now have to calculate the total cost of each node for this problem (cumulative + step + heuristic), which will determine its priority in the queue. In order to find the cumulative cost, we need to find the set of actions that. Similar to finding the solution array as a set of actions by backtracking from the current node to the start node, we will need to backtrack to find the actions for the cumulative cost of the current node.

However, if the current node is indeed the goal node, we should display the solution. We will have to reconstruct the path, so starting at the current state (assumed to be the goal state at first), we will back-track, taking the node's parent information using n_curr.parent. We will add each and every node to a solution array, but since we are adding through backtracking, we will have to reverse the order to get the real solution.

Now, we need to determine the actual actions between each node in the solution array. To do this, I will compare each adjacent item-pair in the solution array. They are currently represented as positions (in a tuple), so if we subtract those positions, we will have a direction vector, which will tell us the action that was taken to get from the first position to the next. I used the lambda function in order to subtract a tuple from another tuple, which will get me the direction information as a unit vector. We compare the result, and determine which action was taken. Then we will add that action to the finalSolution array, which is a set of actions