COEN 166 Artificial Intelligence

Lab Assignment #3: Sample Submission

Name: Tanner Kaczmarek ID: 00001443463

Problem 1 Breadth-First search

Function 1:

```
def breadthFirstSearch(problem):
    """
    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 ***"

    from game import Directions
    from collections import deque

    visited = []
    deq = deque()

    if(problem.goalTest(problem.getStartState())):
        return []

    node = Node(problem.getStartState(), None, None, 0)
    deq.appendleft(node)
    found = False

    while(found != True):
        node = deq.pop()
        if(problem.goalTest(node.state) == True):
            found == True
            break
        acts = problem.getActions(node.state)
        visited.insert(0, node.state)
        for currentAction in acts:
            child_state = problem.getResult(node.state, currentAction)
            child_cost = problem.getCost(node.state, currentAction)
            child_Node = Node(child_state, node, currentAction, child_cost)
            if(child_state in visited):
                continue
            else:
                deq.appendleft(child_Node)
```

```
    finalActions = []
    while not node.state == problem.getStartState():
        finalActions.insert(0, node.action)
        node = node.parent

    return finalActions
```

Comment: The solution for Breadth First Function has only one function. There are many different parts of this solution though. The first part is essentially creating variables that I will use in my while loop to find my goal node. I chose to use a deque but I essentially use it as a queue as I insert from one side and pop from the other side.

In my while loop I iterate until the current node's state I am working from off of the deque is equal to the goalState of the function. If that's not the case then I will go through the actions of the current node and create a child_Node for the action and append it to the other side of the deque from which I pop off of to get the node I am working with. The deque is only used as a FIFO data structure.

The last part of the function gets all the actions of the function into a list called finalActions. It starts with the goalNode from the while loop and iterates through its parent until the current node's state is equal to the startState. While it goes through, it adds the action of the node to the finalActions list.

Problem 2 Depth-First Search

Function 1:

```
def depthFirstSearch(problem):

    from game import Directions

    node = Node(problem.getStartState(), None, None, 0)
    frontier = util.Stack()
    frontier.push(node)

    visited = []

    def iterate():
        tempNode = frontier.pop()
        if(problem.goalTest(tempNode.state) == True):
            return tempNode
        visited.insert(0, tempNode.state)
        acts = problem.getActions(tempNode.state)
        solution = "failed"
        for x in acts:
            child_state = problem.getResult(tempNode.state, x)
```

```
            child_cost = problem.getCost(tempNode.state, x)
            child_Node = Node(child_state, tempNode, x, child_cost)
            if(child_state in visited):
                continue
            frontier.push(child_Node)
            solution = iterate()
            if (str(solution) != "failed"):
                    return solution
        return solution

    goalNode = iterate()
    actions = []
    while not goalNode.state == problem.getStartState():
        actions.insert(0, goalNode.action)
        goalNode = goalNode.parent

    return actions
```

Comment: In my implementation of DepthFirstSearch I used two functions (depthFirstSearch() & iterate()). One inside the other. To keep it simple I am just going to explain them together.

The first part of Depth First Search, I am creating variables that I will use later in the my iterate function to find the goalNode. I decided to use a stack that is a LIFO data structure.

The next part of my DepthFirstSearch is the function iterate(). Iterate is called below as goalNode = iterate(). The first thing I do with iterate is I pop off the top node and check if this node's state is equal to the goal State. If it is not then I add it's state to a visited list. The next thing I do is I iterate through the actions of the current node. I add the current action's resultant node to frontier if it has not been visited before. I then do recursion by calling the iterate function again. I do this to search for the deepest possible option before looking at other options.

The last part of my depth first search gets all the actions of the function into a list called actions. It starts with the goalNode from the while loop and iterates through its parent until the current node's state is equal to the startState. While it goes through, it adds the action of the node to the actions list.