COEN 166 Artificial Intelligence
Lab 3: Pacman Search Algorithms
Name: Tristan Limawan and Collin Paiz **ID:** 1575563 and 1576109

Problem 1: Breadth-First Search

```
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 ***"
    frontier = util.Queue()
    visited = []

    #Get the starting state
    startState = problem.getStartState()
    current = Node(startState, None, None, 0)
    startNode = current
    frontier.push(current)
    visited.append(current)

    # Expand all possible unique nodes w/ BFS until we reach the goal node
    while (not frontier.isEmpty()):
        # Move to the next node in the queue
        current = frontier.pop()

        if (problem.goalTest(current.state)):
            # Generate the path
            path = []
            # Continue backwards up the path until we are at the starting node
            while (current is not startNode):
                path.append(current.action)
                current = current.parent
            # Reverse the path list to get order from start to beginning
            path.reverse()

            return path
        else:
            # For all the actions, generate the possible next nodes and add
them into the frontier queue if they aren't already in it
            for action in problem.getActions(current.state):
                nextNode = Node(problem.getResult(current.state, action),
current, action, problem.getCost(current.state, action))
                if nextNode not in visited:
                    frontier.push(nextNode)
                    visited.append(nextNode)
```

```
    return []
```

Explanation:

```
frontier = util.Queue()
visited = []

#Get the starting state
startState = problem.getStartState()
current = Node(startState, None, None, 0)
startNode = current
frontier.push(current)
visited.append(current)
```

Declare the frontier as a queue and make a list to store the visited nodes. The list for the visited nodes is used for graph search so that we remember nodes that we have already visited (to be elaborated on later). We want to get the starting state from the problem, set the current node to a new node with the starting state, and then set startNode to this node (so that we can later use it to backtrack). We then push the first (current) node to the frontier queue as well as the visited list.

```
# Expand all possible unique nodes w/ BFS until we reach the goal node
while (not frontier.isEmpty()):
    # Move to the next node in the queue
    current = frontier.pop()

    if (problem.goalTest(current.state)):
        # Generate the path
        path = []
        # Continue backwards up the path until we are at the starting node
        while (current is not startNode):
            path.append(current.action)
            current = current.parent
        # Reverse the path list to get order from start to beginning
        path.reverse()

        return path
```

While there are still nodes in the frontier queue, we are going to loop and get the next actions from the first node in the frontier queue. If we are at the goal state, then we are going to create an empty path list and work backwards up the nodes until we reach the start node. We are going to backtrack from the goal state which is the current node, and keep backtracking until we are back at the startNode. We will append this onto the path and continue going up the parent nodes until we are back at the start. Since the path is currently from goal to start, we reverse the list to get

the path from start to goal. Return this path.

```
    else:
        # For all the actions, generate the possible next nodes and add them into the frontier queue if they aren't already in it
        for action in problem.getActions(current.state):
            nextNode = Node(problem.getResult(current.state, action), current, action, problem.getCost(current.state, action))
            if nextNode not in visited:
                frontier.push(nextNode)
                visited.append(nextNode)
```

If we are not at the goal state then we will get all the possible actions, we are going to make a nextNode that is the result of that action. If this node is not already a node that is explored, we will push this node onto the frontier and append it onto visited. Appending th node to visited is how graph search is implemented; if the node is already explored (i.e. in the visited nodes list), then we can continue because we remember that node from before. Then we will continue the loop by updating current to the next node of the frontier queue by popping.

Problem 2: A* Search

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic
first."""
    "*** YOUR CODE HERE ***"
    frontier = util.PriorityQueue()
    startState = problem.getStartState()
    visited = []

    current = Node(startState, None, None, 0)
    frontier.push(current, heuristic(startState, problem))
    visited.append(current)

    while (not frontier.isEmpty()):
        current = frontier.pop()
        if (problem.goalTest(current.state)):
            path = []
            i = current
            while (i.parent):
                path.append(i.action)
                i = i.parent
            path.reverse()
            return path
        else:
           for action in problem.getActions(current.state):
               nextNode = Node(problem.getResult(current.state, action),
current, action, problem.getCost(current.state, action))
               if (nextNode not in visited):
                   visited.append(nextNode)
                   nextNode.action = action
                   nextNode.parent = current
                   nextNode.path_cost = current.path_cost + nextNode.path_cost
```

```
                    frontier.push(nextNode, nextNode.path_cost +
heuristic(nextNode.state, problem))

    return []
```

Explanation:

```
frontier = util.PriorityQueue()
startState = problem.getStartState()
visited = []

current = Node(startState, None, None, 0)
frontier.push(current, heuristic(startState, problem))
visited.append(current)
```

Overall, this function is similar to the previous one, except we are using a priority queue. Our frontier is a priority queue, and again we create a visited list to keep track of nodes we have already been to. We get the startState, set current to a node with this startState, and we push the current node onto the frontier with the priority, which is based on the path cost previous to the node + the Manhattan heuristic value. Since we are at the start node, the path cost right now is 0. Then we add this start node immediately to the visited list so we don't traverse the start node again.

```
while (not frontier.isEmpty()):
    current = frontier.pop()
    if (problem.goalTest(current.state)):
        path = []
        i = current
        while (i.parent):
            path.append(i.action)
            i = i.parent
        path.reverse()
        return path
```

While our frontier is not empty, we are going to set current to the first node in the frontier queue. If we are at the goal state, we do something similar to what we did to backtrack to the starting state. This time we are using the variables in the node to back track. We set i to the current node, which is the goal state, and while a parent exists for the node we are at, we will continue to go backwards and append the action onto the path list. When there are no more parents, we are at the start node. Again, our list is backwards from goal to start, so we reverse to get the path from start to goal.

```
else:
   for action in problem.getActions(current.state):
       nextNode = Node(problem.getResult(current.state, action), current, action, problem.getCost(current.state, action))
       if (nextNode not in visited):
           visited.append(nextNode)
           nextNode.action = action
           nextNode.parent = current
           nextNode.path_cost = current.path_cost + nextNode.path_cost
           frontier.push(nextNode, nextNode.path_cost + heuristic(nextNode.state, problem))
```

When we are not at the goal state, we want to expand the nodes. Similar to the previous function, we getall the actions, and for all actions we get the nextNodes. If the nextNode is not in visited, we will append it to the visited list to keep track of it. Then we set the nextNodes actions, parent, and path_cost. Path_cost is the cost of getting to that node from the start. TO get this, we add the path_cost of the parent (the start to the parent) with the cost of nextNode (the current node to the nextNode). This gives us the total path_cost from start to the nextNode. We finally push the node onto the frontier priority queue, adding the total path_cost with the Manhattan heuristic.