Department of Computer Science and Engineering
Islamic University of Technology (IUT)
A subsidiary organ of OIC

Laboratory Report: Lab 1

CSE 4712: Artificial Intelligence Lab

Name: Tasfia Tasneem Annesha
Student ID: 190041220
Section: 2B
Semester: 7th
Academic Year: 2021-2022

Date of Submission: 30/8/2023

1 Problem Statement

As part of the first task, we had to put a DFS search algorithm into action so that the Pacman agent could locate the food. The code looks like this:

Task 1:DFS search algorithm

This task was very easy. In order to complete the task, we must sum the values of the two arguments passed to the function and return the result.

Working code:

```
def depthFirstSearch(problem):

    "*** YOUR CODE HERE ***"

    # util.raiseNotDefined()

    fringe = util.Stack()
    visited = []
    fringe.push((problem.getStartState(), []))


    while True:
        if fringe.isEmpty():
            return None


        currState, currActionSequence = fringe.pop()


        if problem.isGoalState(currState):
            return currActionSequence

        visited.append(currState)


        for nextState, nextAction, nextCost in problem.getSuccessors(currState):

            if nextState not in visited:
                fringe.push((nextState, currActionSequence + [nextAction]))
```

Explanation:

First I declared a stack as a fringe. Given that this is a DFS graph search, I maintain a visited list. Starting node is pushed into the edge. The algorithm begins with a loop that iterates through until the fringe is empty. If the goal state is present, we pop each item from the fringe and determine whether it is; if not, we determine whether the node has already been visited. If it has, we return the list of steps taken to get there. If not, we expand this node, move its successors closer to the center, and add it to the list of nodes that have been visited.

Task 2: Breadth First Search

Working code:

```
def breadthFirstSearch(problem):

    "*** YOUR CODE HERE ***"
    #util.raiseNotDefined()

    fringe = util.Queue()
    visited = []
    fringe.push((problem.getStartState(), []))


    while True:
        if fringe.isEmpty():
            return None

        currState, currActionSequence = fringe.pop()


        if problem.isGoalState(currState):
            return currActionSequence


        if currState not in visited:

            for nextState, nextAction, nextCost in problem.getSuccessors(currState):
                fringe.push((nextState, currActionSequence + [nextAction]))

            visited.append(currState)
```

Analysis of the problem:

There is only a small difference between this task and task 1 overall. The fringe in BFS graph search is a queue.

Task 3: Uniform Cost Search

```
def uniformCostSearch(problem):
    "*** YOUR CODE HERE ***"
    #util.raiseNotDefined()
    fringe = util.PriorityQueue()
    visited = []
    fringe.push((problem.getStartState(),[],0),0)

    while True:
        if fringe.isEmpty():
            return None
        currState, currActionSequence ,currCost = fringe.pop()

        if problem.isGoalState(currState):
            return currActionSequence

        if currState not in visited:
            for nextState, nextAction, nextCost in problem.getSuccessors(currState):
                fringe.push((nextState, currActionSequence + [nextAction],currCost +
nextCost),currCost + nextCost)

            visited.append(currState)
```

Explanation of the Question:

The ways that DFS, BFS, and UCS are implemented vary very little from one another. Consequently, this task is clearly very similar to tasks 1 and 2. In order to complete this task, we use nearly the same code. However, the following distinctions make this an uninformed graph search algorithm:

1. Here the fringe is priority queue
2. The fringe for DFS and BFS contained tuples that were made up of a state and an action list. However, the UCS algorithm's tuples also include an additional item—the total cost incurred to get there. Priorities are determined based on cost.
3. Cost of reaching start state is 0
4. The cumulative cost is calculated and pushed along with the successor nodes by adding the current cumulative cost and the cost of moving to the next state.