CS-361L Artificial Intelligence Lab 03

Type of Lab: Open Ended

Weightage: 5%

CLO 1: Apply Informed and Uninformed Search Techniques and build the ability to theoretical and practical understanding of Blind and Informed machine search and machine learning techniques.

Student Understand the search project of PacMan and where to write the code to navigate the pacman.	**Cognitive/Understanding**	CLO1	Rubric A
Demonstrate ethical and professional responsibilities involved in completion of Tasks	**Affective/Valuing**	(CLO6)	Rubric B

Reference: The Lab contents are extracted from the BerkeleyX/CS188

Rubric A: Cognitive Domain

Evaluation Method: GA shall evaluate the students for Question 1-3 according to following rubrics.

CLO	**0**	**1**	**2**	**3**	**4**	**5**
CLO1	Student was not able to complete challenge 01	Student was not able to understand challenge 02	Student was not able to complete challenge 02	Student was not able to understand challenge 03	Student was not able to partially complete challenge 03	The pacman was able to navigate itself in the maze through DFS
Roll Number						

Reference: The Lab contents are extracted from the BerkeleyX/CS188

Reference: The Lab contents are extracted from the BerkeleyX/CS188

Rubric B: Affective Domain: Lab Staff shall help GA in evaluation of the students for their CLO 6

CLO 6	**0**	**1**	**2**	**3**
Demonstrate ethical and professional responsibilities involved in completion of Tasks.	Student was not on time in the lab	Student showed some unethical behavior and was late in lab	Student was on time and showed some unethical behavior	Student was obedient and showed ethical behavior
Roll Number				

Reference: The Lab contents are extracted from the BerkeleyX/CS188

Search Problem: Moving Your Pac Man using First Child First (Depth First Search).

Note: All code should be written in the depthFirstSearch (problem) function of Search.py

Task A. Print the Start State of the Pacman.

1. **Add following code inside the depthFirstSearch function of search.py**

```
print "Start:", problem.getStartState()
return [];
```

2. Run following command

   Python pacman.py –l mediumClassic –p SearchAgent –a fn=depthFirstSearch

3. The output should be

```
C:\Python27\search>python pacman.py -l mediumClassic -p SearchAgent -a fn=depthF
irstSearch
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
Start: (9, 1)
Path found with total cost of 0 in 0.0 seconds
Search nodes expanded: 0
```

Task B: Write a program that list down all possible navigation states from the start state.

1. Add following code inside the depthFirstSearch function or search.py

```
currentState=problem.getStartState()
childrens=problem.getSuccessors(currentState)
print childrens
return [];
```

2. Run following command

   Python pacman.py –l mediumClassic –p SearchAgent –a fn=depthFirstSearch

3. The output shall show the all possible states that can be navigated from current state. The specific problem shall return two states (10,1) and (8,1) and also return the action that could take to that state. For example first record is ((10,1),'East',1) ; this first part show the grid position, second part represent the action that need to be taken to reach at **(10,1)**, and last part shows the cost of the node.

```
C:\Python27\search>python pacman.py -l mediumClassic -p SearchAgent -a fn=depthF
irstSearch
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
[((10, 1), 'East', 1), ((8, 1), 'West', 1)]
Path found with total cost of 0 in 0.0 seconds
```

Task C: Write a code that move pacman toward the first child of the currentState.

1. Replace following code inside the depthFirstSearch function or search.py

```
currentState=problem.getStartState()
childrens=problem.getSuccessors(currentState)
action = getActionFromTriplet(childrens[0])
return [action];
```

   Also, add following code for function getActionFromTriplet

```
def getActionFromTriplet(triple):
    return triple[1];
```

2. Run the code

Task D: write a code that move pacman from its current state to its first child and repeat the process for at least 10 iterations.

1. Write a code that move pacman toward the first child of the currentState.

```
currentState=problem.getStartState()
actions=[]
maxIteration =0
while(maxIteration <=10):

    children=problem.getSuccessors(currentState)
    actions.append(getActionFromTriplet(children[0]))
    "first index pick the record of firstChild"
    firstChild =children[0]
    "as record consist of triplet,state,action and cost"
    firstChildState = firstChild[0]
    currentState = firstChildState
    maxIteration=maxIteration+1
return actions;
```

2. Run the code

Challenge 01: Increase the maxIteration to 20,30 and 40 and see the behavior of pacman. If pacman do not die out, at some stage, it will be stuck in loop and cannot move forward. Write down the reason why he is getting into the loop and identify the part of the code which need to be modified.

Challenge 02: Change the above code so PACMAN never stuck in loop.

1. [This is lab task that students are supposed to solve.]
2. Command to run the solution

Reference: The Lab contents are extracted from the BerkeleyX/CS188

Lab Task:

Objective: Finding the Goal State for Mr. PacMan using DFS Search.

Note: All code should be written, until and unless not mentioned, in the `depthFirstSearch` (problem) function of `Search.py`

In searchAgents.py, you'll find a fully implemented SearchAgent, which plans out a path through Pacman's world and then executes that path step-by-step. **The search algorithms for formulating a plan are not implemented -- that's your job.**

First, test that the SearchAgent is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command above tells the SearchAgent to use tinyMazeSearch as its search algorithm, which is implemented in search.py. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! For your reference, the general search algorithm from lecture is shown in Figure 01:

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

Figure 1: Depth First Search (DFS) Algorithm

Important note: Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Reference: The Lab contents are extracted from the BerkeleyX/CS188

Important note: All of your search functions need to return a list of *actions* that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).

Important note: Make sure to **use** the Stack, Queue and PriorityQueue data structures provided to you in util.py! These data structure implementations have particular properties which are required for compatibility with the autograder.

Challenge 03: Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

Your code should quickly find a solution for:

```
python pacman.py -l tinyMaze -p SearchAgent –fn depthFirstSearch

python pacman.py -l mediumMaze -p SearchAgent –fn depthFirstSearch

python pacman.py -l bigMaze -z .5 -p SearchAgent –fn depthFirstSearch
```

Challenge 04:

If you use a `Stack` as your data structure, the solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, write down what depth-first search is doing wrong.