ARTIFICIAL INTELLIGENCE ASSIGNMENT #1

Due Date: 2020/4/14(Tuesday) 11:59

Python Version: 3.6.

All of the source code is from CS188 Berkeley.
(https://inst.eecs.berkeley.edu/~cs188/sp20/projects/)

You only have to solve Question 1~4 in https://inst.eecs.berkeley.edu/~cs188/sp20/project1/. You can read through this PDF file or directly go to the link to solve the questions.

Submission policy is at the end of this PDF.

Tutorial

In cs188 assignments, you can evaluate your assignment grade by yourself using autograder. If you want to get familiar with the autograder, go through this tutorial.

First, download all of the files associated the autograder tutorial as a zip archive: (tutorial.zip). Unzip the file and you can see its contents:

```
cose361@one:~$ cd tutorial
cose361@one:~/tutorial$ ls
addition.py       shopAroundTown.py   textDisplay.py
autograder.py     shopSmart.py        town.py
buyLotsOfFruit.py submission_autograder.py  tutorialTestClasses.py
grading.py        testClasses.py      util.py
projectParams.py  testParser.py
shop.py           test_cases
```

The files that you'll edit or run:

- `addition.py`: source file for question 1
- `buyLotsOfFruit.py`: source file for question 2
- `shop.py`: source file for question 3
- `shopSmart.py`: source file for question 3
- `autograder.py`: autograding script (see below)

Try to run `python autograder.py` to grade your solutions for all the three problems and you will get the output as below (omitting the results for questions 2 and 3). Since you haven't solve the problems, all scores will be zero.

```
cose361@one:~/tutorial$ python autograder.py
Starting on 3-30 at 17:08:55

Question q1
==========

*** FAIL: test_cases/q1/addition1.test
***   add(a,b) must return the sum of a and b
***   student result: "0"
***   correct result: "2"
*** FAIL: test_cases/q1/addition2.test
***   add(a,b) must return the sum of a and b
***   student result: "0"
***   correct result: "5"
*** FAIL: test_cases/q1/addition3.test
***   add(a,b) must return the sum of a and b
***   student result: "0"
***   correct result: "7.9"
*** Tests failed.

### Question q1: 0/1 ###

...

Finished at 17:08:55

Provisional grades
==================
Question q1: 0/1
Question q2: 0/1
Question q3: 0/1
------------------
Total: 0/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Let's try to solve question 1 and run `python autograder.py` again. Open `addition.py` and modify as follows:

```
def add(a, b):
"Return the sum of a and b"
"*** YOUR CODE HERE ***"
    return a + b
```

Now rerun `python autograder.py` (omitting the results for questions 2 and 3). You can see that you got one point for question 1.

```
cose361@one:~/tutorial$ python autograder.py
Starting on 3-30 at 17:33:11

Question q1
===========

*** PASS: test_cases/q1/addition1.test
***   add(a,b) returns the sum of a and b
*** PASS: test_cases/q1/addition2.test
***   add(a,b) returns the sum of a and b
*** PASS: test_cases/q1/addition3.test
***   add(a,b) returns the sum of a and b

### Question q1: 1/1 ###

...

Finished at 17:33:11

Provisional grades
==================
Question q1: 1/1
Question q2: 0/1
Question q3: 0/1
------------------
Total: 1/3

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

This is all for the autograder tutorial!

Note that you ***do not*** submit the tutorial codes!

For more details, click *link*.

Assignment #1

In this first assignment, you will implement **DFS, BFS, and A* search** for a Pacman agent to find paths through his maze world, both to reach a particular location and to collect food efficiently.

Like the tutorial, there is an autograder for you to grade your assignment by yourself. You can run the code as below.

```
cose361@one:~/search$ python autograder.py
```

Download the code from blackboard or by link(search.zip). You can ignore other files that are not listed below.

▼ **Files you'll edit:**

- `search.py`: Where all of your search algorithms will reside.

▼ **Files you might want to look at:**

- `pacman.py`: The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this project.
- `game.py`: The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
- `util.py`: Useful data structures for implementing search algorithms.
- `searchAgents.py`: Where all of your search-based agents will reside.

Welcome to Pacman

After downloading the code, unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

You can change the layout and the Pacman Agent by using `-l` and `-p`.

```
python pacman.py -l testMaze -p GoWestAgent
```

For more options, type the command below in search folder.

```
python pacman.py -h
```

Question 1 (3 points): Finding a Fixed Food Dot using Depth First Search

In `searchAgent.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.

For question 1, implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py` (image below).

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

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.

If you implemented correctly, you will quickly find solutions for:

```
python pacman.py -l tinyMaze -p SearchAgent
```

```
python pacman.py -l mediumMaze -p SearchAgent
```

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

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: 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, think about what depth-first search is doing wrong.

Question 2 (3 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py` (image below). Again, write a graph search algorithm that avoids expanding any already visited states.

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

If you implemented correctly, you will quickly find solutions for:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

If BFS does not find a least cost solution, check your implementation.

Question 3 (3 points): Varying the Cost Function

Implement the uniform-cost graph search algorithm in the uniformCostSearch function in `search.py`.

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

We encourage you to look through `util.py` for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

```
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
```

```
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Note: You should get very low and very high path costs for the StayEastSearchAgent and StayWestSearchAgent respectively, due to their exponential cost functions (see `searchAgents.py` for details).

Question 4 (3 points): A* search

Implement A* graph search in the empty function `aStarSearch` in `search.py`.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The `nullHeuristic` heuristic function in `search.py` is a trivial example.

You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as `manhattanHeuristic` in `searchAgents.py`).

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on `openMaze` for the various search strategies?

For more details, click *link*.

Submission

You only need to solve Question 1~4. To get full credit, run `autograder.py` and it should look like below.

```
cose361@one:~/search$ python autograder.py

...

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 12/25
```

▼ There are two files you need to submit.

1. Submit `search.py` on Blackboard(2 points).
2. Submit a pdf file containing(3points):
    1. screenshot the result of `autograder.py` in terminal

```
Finished at 14:43:41

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 12/25
```

    2. Three discussions on three different algorithms(DFS, BFS, A*) when playing Pacman
        - Discuss why the DFS can have lower score than the BFS. (When playing mediumMaze, DFS has lower score than BFS). You can attach the screenshot of Pacman play. Then, why would someone prefer DFS over BFS even though it can have lower score?

- Discuss the difference between A* and BFS algorithm. Which algorithm do you think is better than the other? why?
- Ask yourself one question and answer.

To check your understanding, please submit explanation of your code with pdf file or comment in the python files.

About Plagrism

We know that it is easy to find source code for this assignment. However, if we find out that you just copied from one of the source code, grade for the assignment will be zero. You can refer to those source codes, but do not just copy and paste them.

Q & A

If you have any questions, feel free to ask TAs by e-mail.

- Youngjin Oh: *dign50501@korea.ac.kr*
- Hyeonjin Park: *hyeonjin961030@gmail.com*
- SeungDong Yoa: *ysd1029@korea.ac.kr*