Assignment I: Search

This assignment involves writing search implementations for a virtual agent in the Pacman environment. The goal is to enable the Pacman agent to plan paths through the maze world, both to reach a particular location and to collect food efficiently. This assignment will use the Pacman project developed by Dan Klein for AI educational purposes.

1. Getting Started

- Download the starter code for the assignment from this repository. The package contains code for running the Pacman environment. You will be editing the **search.py** and **searchAgents.py** files in this exercise.
- This assignment will use Python programming language. The recommended Python version is 3.6.x.
- We recommend that you create a separate conda environment for the AI assignments so that you don't interfere with the system python or any other python environment that you may already have. If you don't have conda already installed, please see this for the latest version. After installing conda, create an environment that uses python 3.6.

```
conda create --name col333 python=3.6
```

- You can test the setup with the following commands. The steps should show the Pacman GUI and you should be able to control the agent in the grid.

```
conda activate col333
git clone https://github.com/reail-iitd/COL333-671-A1
cd COL333-671-A1
python pacman.py
conda deactivate col333
```

- We have tested the above setup with Ubuntu 16.04/Python3 and Windows 10/Python 3. We have tried on several machines. Students are expected to take initiative in setting up the environment on their individual machines.

- Running *python pacman.py* should display the Pacman agent navigating in a virtual maze environment. You can spawn the default reflex agent that always goes in a single (West) direction in different mazes.

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

- Pressing CTRL-c in the terminal will exit the environment. The options (and their default values) for *pacman.py* can be listed with the command below. E.g., if the Pacman moves slowly then the option –frameTime 0 can be used.

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

- There are some sample test cases that we provide for each question which can be exercised using the command below. These can be used while developing your solution. Once your code is submitted, your code will be evaluated on a range of new test cases.

```
python autograder.py # to test all test cases for each question
python autograder.py --q <question_name> # to only check a particular question
```

2. Search Algorithm Implementation [25 points]

- This section involves implementing search algorithms for planning paths in the maze for the Pacman agent. The objective is to get Pacman to navigate to a fixed food dot in the maze.
- Test the SearchAgent with a default search algorithm (tinyMazeSearch). The Pacman should navigate the maze successfully.

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

- Next, implement the search algorithms listed below. While implementing the search algorithms remember the following:
    - All of the search functions must return a list of actions that will lead the agent from the start to the goal. All actions must be legal moves and valid (e.g., cannot move into walls).
    - Please **only** use the **Stack, Queue and PriorityQueue** data structures provided in **util.py**. These data structures are necessary for the auto grading functionality.

2.1. Depth First Search (DFS)

- Implement the graph search version of DFS. Fill the **depthFirstSearch** function in **search.py**.
- You can test in various conditions with commands as follows. The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means the state is explored earlier).

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

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

python pacman.py -l bigMaze -z 0.5 -p SearchAgent
```

- Run the command below to check if your implementation passes the default test cases in autograder.py

```
python autograder.py -q test_dfs
```

2.2. Breadth First Search (BFS)

- Implement the graph-search version of BFS algorithm. Fill the **breadthFirstSearch** function in **search.py**. As previously, you can test the implementation with the following commands.

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

python pacman.py -l bigMaze -z 0.5 -p SearchAgent -a fn=bfs
```

- Run the command below to check if your implementation passes the default test cases in autograder.py

```
python autograder.py -q test_bfs
```

2.3. Uniform Cost Search (UCS)

- Incorporate a notion of cost in the search. Fill the **uniformCostSearch** function in **search.py**. Implement the graph-search version of UCS. Use the data structures provided in **util.py**.
- Test your implementations with worlds with different costs.

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

python pacman.py -l bigMaze -z 0.5 -p SearchAgent -a fn=ucs
```

- Run the command below to check if your implementation passes the default test cases in autograder.py

```
python autograder.py -q test_ucs
```

2.4. A* Search

- Implement A* graph search by providing your code in the empty function **aStarSearch** in the file **search.py**. The A* function takes a heuristic function as an argument. A heuristic can take two arguments: a state in the search problem and the problem itself. The ***nullHeuristic*** in **search.py** is a trivial example of a heuristic function.

- Test your implementation with the **manhattanHeuristic** in searchAgents.py as follows:

```
python pacman.py -l bigMaze -z 0.5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
```

- Run the command below to check if your implementation passes the default test cases in autograder.py

```
python autograder.py -q test_astar
```

3. Corners Problem [40 points]

- This part will involve using A* search (graph version) for the Corners Problem. The first part involves formulating the search problem and the second part involves designing appropriate heuristics.
- The agent lives in a *corner maze*, where there are four dots, one in each corner. The agent's goal is to find the shortest path through the maze such that it visits all the corners, irrespective of whether there is food present at that maze location or not.
- Implement the ***CornersProblem*** search problem in **searchAgents.py**. The implementation can be tested as follows:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

- Note that the state representation used should be minimal. In particular, please do not include the full GameState in the state during planning. **(Note: Do not touch the already defined function increment_expanded(). Also make sure it gets called at the end of getSuccessors() - already done for you. Not adhering to this will straightaway lead to 0 marks.)**
- Run the command below to check if your implementation passes the default test cases in autograder.py

```
python autograder.py -q test_corners_state
```

- Implement a heuristic function for this problem. Note that your heuristic must be consistent and non-negative (positive values everywhere). Further, the heuristic should not be null (zero everywhere) and should not be computing the actual cost to the goal.
- Place your heuristic implementation for the **CornersProblem** in **cornersHeuristic**. The search can be run as:

```
python pacman.py -l mediumCorners -p SearchAgent -a
fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

- Run the command below to check if your implementation passes the default test cases in autograder.py

```
python autograder.py -q test_corners_heuristic
```

- The evaluation for this part will take into account the number of nodes expanded and the time you take to generate the solution. There will be a reasonably long timeout - crossing this time limit will immediately terminate your solution. Your solution must use a non-trivial, non-negative and consistent heuristic as specified previously.

4. Food Clearing Problem [35 points]

- This part involves solving the food clearing problem. The goal is to search for the shortest path for the Pacman to collect all the food available in the maze. Note that the solution should depend only on the agent's state, wall placement and regular food.
- Implement an A* heuristic for this problem. Write your code in ***foodHeuristic*** in **searchAgents.py**. Test your implementation for a default test domain as follows:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

- As in the previous problem, the evaluation for this part will take into account the number of nodes expanded and the time you take to generate the solution. Again, there will be a reasonably long timeout - crossing this time limit will immediately terminate your solution. Your solution must use a non-trivial, non-negative and consistent heuristic.
- Run the command below to check if your implementation passes the default test cases in autograder.py

```
python autograder.py -q test_food_clearing_search
```

5. Submission Instructions

- This assignment is to be done individually.

- Submit a single zip file named ***<A1-EntryNumber>.zip***. Upon unzipping this should yield two files named - *search.py* and *searchAgents.py* in the same directory. A sample submission (***A1-2016CS50394.zip***) has been attached.
- The assignment is to be submitted on Moodle.
- The submission deadline is November 6, 2020.
- This assignment will carry 12% of the grade.
- Late submission deduction of (20% per day) will be awarded.
- Parts 3 and 4 in the assignment will be relatively graded.
- Your code will be graded using evaluation scripts. Please do not change the names of any of the provided functions or classes within the code. Carefully follow the format. Only make changes to the two specified files and in the correct functions. Your code should use only standard python libraries (the Conda environment setup). Do not include any dependencies on third party libraries.
- No credit provided if you modify other functions which you are not supposed to.
- Please only submit work from your own efforts. Do not look at or refer to code written by anyone else. You may discuss the problem, however the code implementation must be original. Discussion will not be grounds to justify software plagiarism. Please do not copy existing assignment solutions from the internet: your submission will be compared against them using plagiarism detection software.
- Copying and cheating will result in a penalty of at least -10 (absolute). The department and institute guidelines will apply. More severe penalties such as F grade will follow.
- Queries if any should be raised on Piazza. Enrol on **piazza.com/iit_delhi** for the course **Fall 2020** term of **COL 333: Artificial Intelligence** using access code **col333.**
- Remember that the autograder given to you tests your solution on default test cases. The final evaluation will be done on new unseen test cases. Therefore you may still not receive credit even if the autograder passes on the default test cases.

6. Undertaking

The Pacman project was developed by Dan Klein at the University of California, Berkeley and is used for AI education in several universities. In order to support continued use of the Pacman framework for teaching in several universities, each student enrolled in COL333 and COL671 must take the following undertaking: “The Pacman project is freely available for educational use. Since the framework is used at multiple universities for AI education, it is mandated that we do not distribute or post solutions to any of the projects. Any redistribution of the code or release (e.g., on a Github account) by any student taking the course will be considered as a violation of the Honor Code for this class”. Penalties listed above will be applied if a student makes the solution available online.

7. List of Files in the Pacman Project

Files to edit:

search.py	Where all of your search algorithms will reside.
searchAgents.py	Where all of your search-based agents will reside.
Files 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.
A1-2016CS50394.zip	Sample submission
Files to ignore:	
graphicsDisplay.py	Graphics for Pacman
graphicsUtils.py	Support for Pacman graphics
textDisplay.py	ASCII graphics for Pacman
ghostAgents.py	Agents to control ghosts
keyboardAgents.py	Keyboard interfaces to control Pacman
layout.py	Code for reading layout files and storing their contents
autograder.py	Project autograder
testParser.py	Parses autograder test and solution files
testClasses.py	General autograding test classes
test_cases/	Directory containing the test cases for each question
searchTestClasses.py	Project 1 specific autograding test classes
searchAgents-TA.py	Required for autograder. Please **do not** change.