Assignment II: Multi-Agent Search

This assignment involves enabling the Pacman agent to act appropriately in the game where there are ghosts in the world. The Pacman still aims at eating all the dots but must plan its action taking the behaviour of the ghosts into account. This exercise will involve modeling the decision-making task as an adversarial search problem that allows the Pacman to decide actions while taking into account the behaviour of ghosts.

1. Getting Started

- Please download the starter code for the assignment from this link and do a fresh install.
- As before, please use Python 3.6 for this assignment.
- You can invoke the Pacman game as follows.

```
<download and extract multiagent.zip>
cd <path-to-extracted-multiagent>
python pacman.py
```

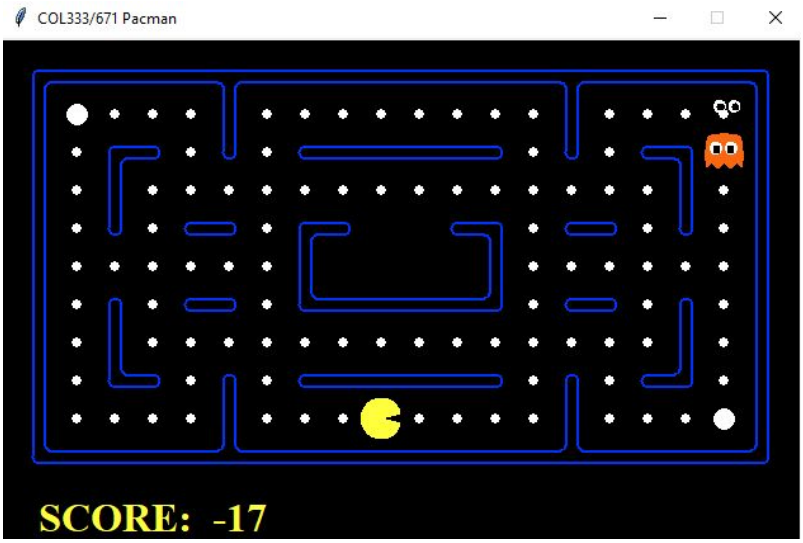
- A simple ReflexAgent is provided in the package and can be invoked in game environments populated with ghosts as a demonstration. The following command will invoke the ReflexAgent present in **multiAgents.py**. Running the command will invoke the default **mediumClassic** layout and you should see the Pacman moving in the presence of Ghosts.

```
python pacman.py -p ReflexAgent
```

- The ReflexAgent can be invoked in other environments such as the **testClassic** environment with the -l flag.

```
python pacman.py -p ReflexAgent -l testClassic
```

- Review the **ReflexAgent** code in **multiAgents.py** that provides examples to query the **GameState** for information. The **GameState** object specifies the full game state, including the food locations and the ghosts location which the Pacman needs to consider while playing the game.



Screenshot of a **ReflexAgent** moving in the **mediumClassic** layout in the presence of ghosts.

- A list of options that may be useful during the assignment.

--frameTime 1	Allows the game to be paused at every frame
--frameTime 0	Turns off the animation to speed up the display
-g	The default ghosts are random. The option -g can be used to invoke DirectionalGhosts
-n	Used to play multiple games in a row
-f	Run with a fixed random seed. This allows the same random choices in every game
-k	Specifies the number of ghosts (will not exceed the number the layout allows)

2. Improving the Reflex Agent [4 points]

- This part involves improving the **ReflexAgent** class in **multiAgents.py** to play better. In particular, you have to fill out the **evaluationFunction** in the **ReflexAgent** class.
- A capable ReflexAgent must consider both the food locations and ghost locations to perform well. Improve the **ReflexAgent** such that it can clear the **testClassic** layout.

```
python pacman.py -p ReflexAgent -l testClassic
```

- Next, try the ReflexAgent on the default **mediumClassic** layout by varying the number of ghosts. Note that there cannot be more ghosts than the number that the layout permits. The animation can be turned off to speed up the display.

```
python pacman.py --frameTime 0 -p ReflexAgent -k 1

python pacman.py -p --frameTime 0 -p ReflexAgent -k 2
```

- You may use the following notes/hints for this part:
    - As features, you may try the reciprocal of important values (such as distance to food) rather than the values themselves.
    - Remember that **newFood** has a function **asList()**
    - You may view object internals by printing object internals. For example, **print(newGhostStates)** to print the **newGhostStates**.
- The grading for this part will be as follows:
    - Your agent will be run on **openClassic** layout 10 times. If the agent times out or never wins then 0 points will be awarded. If the agent wins at least 5 times then 1 point will be awarded. If the agent wins all 10 times then 2 points will be awarded.
    - If your agent's average score is greater than 500 then an additional 1 point is awarded. If the agent's score is greater than 1000 then 2 points are awarded.
    - You may test your agent under the above-mentioned conditions by invoking the following command.

```
python autograder.py -q test_reflex
```

    - The above can be run without graphics using the following command:

```
python autograder.py -q test_reflex --no-graphics
```

- Please note that the evaluation function written in this part of the assignment uses the state-action pairs and does not perform look ahead. Note that part 6 of this assignment will involve writing an evaluation function based on states for a game tree.

3. Minimax [5 points]

- Please implement an adversarial search agent in the provided **MinimaxAgent** class stub in **multiAgents.py**. The minimax agent should work with any number of ghosts, and your minimax tree should have multiple min layers (one for each ghost) for every max layer.
- Your code should also expand the game tree to an *arbitrary depth*. Score the leaves of your minimax tree with the supplied **self.evaluationFunction**, which defaults to **scoreEvaluationFunction**. **MinimaxAgent** extends **MultiAgentSearchAgent**, which gives access to **self.depth** and **self.evaluationFunction**. Please make sure your minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options.
- Pacman is always agent 0, and the agents move in order of increasing agent index. Note that only Pacman will be running your implementation of the MinimaxAgent.
- Note that a single search ply is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times.
- The minimax values of the initial state in the **minimaxClassic** layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. You can use these numbers to verify whether your implementation is correct.

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

- Implementation hints:
    - The correct implementation of **minimax** will lead to Pacman losing the game in some tests. This is not a problem:as it is correct behaviour, it will *pass* the tests. Note that in part 6, you will work on improving the evaluation function which will improve the Pacman's performance.
    - Please use the evaluation function **self.evaluationFunction** provided. Please do not change the function but recognize that the function is evaluating states instead of actions (which was the case for the reflex agent). Note that look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.
    - All states in minimax should be GameState's, either passed in to **getAction** or generated via **GameState.generateSuccessor**. Please do not simplify the states. Please do not call the GameState.generateSuccessor more than necessary.
    - You may test on a large number of games using the **-n** and **-q** flags.

Run the command below to check if your implementation passes the default test cases in **autograder.py.** The default test cases check the exploration of the correct number of game states and examine the number of calls to the **GameState.generatedSuccessor.**

```
python autograder.py -q test_minimax
python autograder.py -q test_minimax –-no-graphics
```

Grading: Remember that the autograder given to you tests your solution on default test cases. The final evaluation will also involve new unseen test cases as was done in A1.

4. Alpha-Beta Pruning [5 points]

- This part involves implementing the alpha-beta pruning technique to efficiently explore the minimax tree. The implementation should be done in **AlphaBetaAgent**.
- Please note that the implementation must account for multiple minimizer agents.
- You may test and observe the speed up on the **smallClassic** layout with depth 3 with the following command.

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

- The implementation will be evaluated based on the exploration of the correct number of game states. Please do not reorder the child nodes in the alpha-beta pruning implementation. The successor states must always be processed in the order returned by **GameState.getLegalActions**. As before, please do not call **GameState.generateSuccessor** more than necessary.
- For this assignment, do not prune on equality, this is necessary for evaluation.
- Run the command below to check if your implementation passes the default test cases in **autograder.py.**

```
python autograder.py -q test_alpha_beta_pruning
```

- Note that the correct implementation of alpha-beta pruning will lead to Pacman losing some of the tests. As in the Minimax case, this is not a problem, as it is correct behaviour and it will pass the tests.
- **Grading:** Remember that the autograder given to you tests your solution on default test cases. The final evaluation may be done on new unseen test cases as was done in A1.

5. Expectimax [5 points]

- Note that both Minimax and Alpha-beta assume that you are playing against an adversary who makes optimal decisions. Random ghosts are not optimal minimax agents and hence modeling them as Minimax agents may not be appropriate.
- In this part of the assignment, we will model the probabilistic behaviour of ghosts who may make suboptimal choices. Note that there are multiple adversaries.
- Please fill in the **ExpectimaxAgent** that will take an expectation according to your agent's model of how the ghosts act. Assume that Pacman is playing against multiple adversaries, which each chooses **getLegalActions** uniformly at random.
- Run the command below to check if your implementation passes the default test cases in **autograder.py.**

```
python autograder.py -q test_expectimax
```

- The ExpectimaxAgent can be run with the command below. You may now observe different behaviour of the **ExpectimaxAgent** when in close proximity of the ghosts that before with the **AlphaBetaAgent**. The Pacman can attempt to get more food pieces even when it could be imminently trapped with ghosts.

```
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
```

- You may try the following two scenarios and observe that the ExpectimaxAgent wins about half the time while the AlphaBetaAgent always loses.

```
python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

- The correct implementation of expectimax will lead to Pacman losing some of the tests. This is not a problem: as it is correct behaviour and it will pass the tests.
- **Grading:** Remember that the autograder given to you tests your solution on default test cases. The final evaluation may be done on new unseen test cases as was done in A1.

6. Evaluation Function [6 points]

- Please implement an improved evaluation function for the **ExpectimaxAgent**. Please fill in the provided function **betterEvaluationFunction** that estimates values for states. You can test the agent on **smallClassic** layout with the following commands:

```
python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -q -n 20
```

- Run the following command to check your implementation on the default test setup.

```
python autograder.py -q test_evaluation_fn
```

```
python autograder.py -q test_evaluation_fn --no-graphics
```

- The evaluation for this will be done in two parts: 1. : Absolute and 2: Relative amongst students. The points obtained in both the parts will be combined appropriately.
  1. **Absolute grading:** The autograder will run your agent on the **smallClassic** layout 10 times. We will assign points to your evaluation function in the following way:
     (a) 1 point awarded if your agent wins at least once without timing out the autograder.
     (b) +1 for winning at least 5 times and +2 for winning all 10 times.
     (c) +1 for an average score of at least 500, +2 for an average score of at least 1000 (including scores on lost games)
     (d) +1 if your games take on average less than 30 seconds, when run with –no-graphics option.
     (e) The additional points for average score and computation time will be awarded if you win at least 5 times.
  2. **Relative Grading:** We will generate a new layout and run your agent on it 10 times. Your final score will be the average across these 10 runs. For grading, your final score will be compared with the score obtained by the other students.

7. Submission Instructions

- **This assignment is to be done individually or in pairs.**
- Submit a single zip file named ***<A2-EntryNumber1-EntryNumber2>.zip or <A2-EntryNumber>.zip***. Upon unzipping this should yield a single file named - ***multiAgents.py***.
- The assignment is to be submitted on Moodle.
- **The submission deadline is 5pm on December 9, 2020.**
- There are five parts in this assignment that total to 25 points.
- This assignment will carry 14% of the grade.
- Late submission deduction of (10% per day) will be awarded.
- There are no buffer days. Please submit by the submission date.
- 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 may be done on new unseen test cases. Therefore, you may still not receive credit even if the autograder passes on the default test cases.

8. 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 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 in the class and will lead to penalties”. A penalty will be applied if a student makes the solution available online.

9. List of Files in the Pacman Project

Files to edit:	
multiAgents.py	Where all the multi-agent search 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. You do not need to use these for this project, but may find other functions defined here to be useful.
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
multiagentTestClasses.py	Assignment 2 specific autograding test classes