Project#1: Pacman with Adversarial Search

1. Project Overview

- In this project, you will implement Reflex and Planning agents for the classic version of Pacman, with using Minimax and Expectimax search. You will also design their evaluation functions.
- You will get five different tasks within this project
- This assignment reuses UC Berkeley's Pacman AI project (https://ai.berkeley.edu/), but with minimum adjustments for CSE 17182.

2. Environment

- Projects in this class require Python 3.6+.
- For the grading, I will use vanilla Python 3.9 in a `conda` environment. If you want the same setup, install `conda` and:

```
conda create -n pacman python=3.9 -y
conda activate pacman
python --version   # should show 3.9.x
```

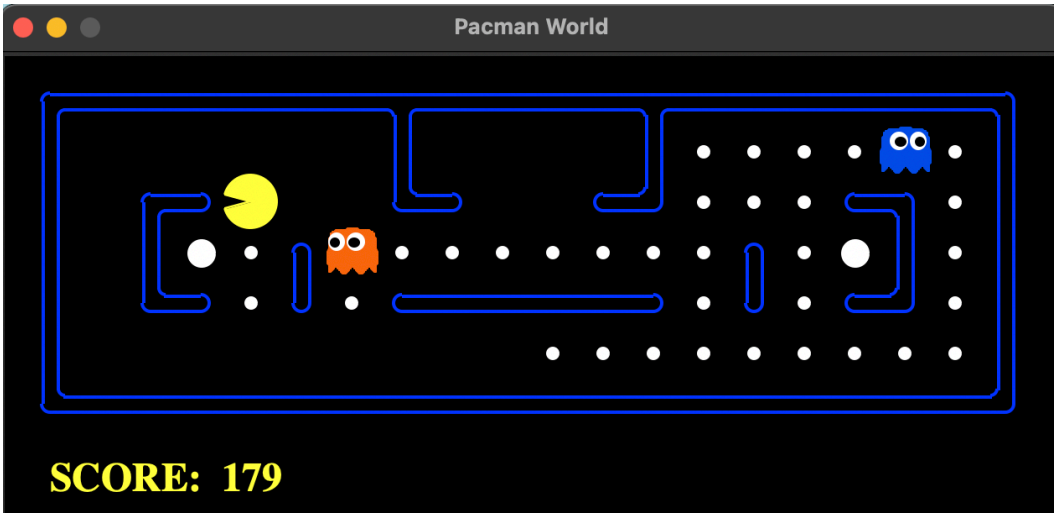
3. Submission Deliverable & Deadline (**IMPORTANT**)

- Deadline: Oct 26 (Sun) 23:59 (through the e-class system)
- Within the code files from the given zip archive (`project1.zip`), you are allowed to edit ONLY `multiAgents.py` and this is the only file you will submit (= submit ONLY your `multiAgents.py` file with no changed filename)
- If you violate this deliverable rule, huge penalty will be applied to your score.

4. Auto-grading

- You can try to grade your answers on your machine. This can be done:
  - `python autograder.py` (Run on all questions)
  - `python autograder.py -q q2` (Run on a particular question)

5. Try the Pacman World



- The given files provide you to play a classic Pacman by your self
  - `python pacman.py` (with using your keyboard)
  - `python pacman.py -p ReflexAgent` (with your built agent)
  - `python pacman.py -p ReflexAgent -l testClassic` (with specific layouts)
- From the original UC Berkeley's Pacman World, I changed a little bit of the game rule for this project
  - The ghosts will have longer attack range. Being adjacent to a non-scared ghost kills Pacman.
  - The ghosts will move at half speed.

6. Other information

- Within the zip archive, you may want to read the codes:
  - `pacman.py`: The main file that runs Pacman games. This file also describes a Pacman `GameState` type, which you will use extensively in this project.
  - `game.py`: The logic behind how the Pacman world works. This describes several supporting types like `AgentState`, `Agent`, `Direction`, and `Grid`.
  - `util.py`: Useful data structures for implementing search algorithms. You don't need to use these for this project, but may find other functions defined here to be useful.
- You can neglect other supporting files (e.g., `graphicsDisplay.py`)

Q1 (4 pts). Reflex Agent

Improve the `ReflexAgent` in `multiAgents.py` to play respectably.

- The provided reflex agent code provides some helpful examples of methods that query the `GameState` for information (e.g., Pacman position food grid, ghost states/timers).
- A capable reflex agent will have to consider both food locations and ghost locations to perform well.
- Try out your reflex agent on the openClassic layout

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

- *Note*: Remember that `newFood` has the function `asList()`
- *Note*: As features, try the reciprocal of important values (such as distance to food) rather than just the values themselves.
- *Note*: The evaluation function you're writing is evaluating state-action pairs; in later parts of the project, you'll be evaluating states.
- *Note*: You may find it useful to view the internal contents of various objects for debugging. You can do this by printing the objects' string representations. For example, you can print `newGhostStates` with `print(newGhostStates)`.

- **Grading**
  - I will run your agent on the `openClassic` layout 10 times.
  - You will receive 0 points if your agent times out, or never wins.
  - You will receive 1 point if your agent wins at least 5 times, or 2 points if your agent wins all 10 games.
  - You will receive an additional 1 point if your agent's average score is greater than 500, or 2 points if it is greater than 1000.
  - You can try your agent out under these conditions with

```
python autograder.py -q q1
```

Q2 (4 pts). Minimax Agent

Write an adversarial search agent in the provided `MinimaxAgent` in `multiAgents.py`.

- Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what you've previously seen in lecture.
- In particular, your minimax tree will 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`. Make sure your minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options. (Do not harcode calls to `scoreEvaluationFunction` since we will be writing a better evaluation function later in the project.)
- Important: A single search ply (i.e., one depth level) is considered to be one Pacman move + all the ghosts' responses.
- You can see how your minimax agent works by running, for example:

```
python pacman.py -p MinimaxAgent -l smallClassic -a depth=2
```

- You will see how Minimax search can be notoriously slow with more depth (e.g., try depth level 4).

- **Grading**
  - The autograder will check your code to determine whether it explores the correct number of game states in different game settings.
  - To test and debug your code, run

```
python autograder.py -q q2
```

Q3 (3 pts). Alpha-Beta Pruning

Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in `AlphaBetaAgent`.

- Again, your algorithm will be slightly more general than the pseudocode from lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.
- You should see a speed-up (perhaps depth 3 alpha-beta will run as fast as depth 2 minimax). Ideally, depth 3 on smallClassic should run in just a few seconds per move or faster.

```
python pacman.py -p MinimaxAgent -l smallClassic -a depth=2
python pacman.py -p AlphaBetaAgent -l smallClassic -a depth=3
```

- **Grading**
  - Again, the autograder will check your code to determine whether it explores the correct number of states.
  - Because of that, it is important that you perform alpha-beta pruning without reordering children. In other words, successor states should always be processed in the order returned by `GameState.getLegalActions`.
  - To test and debug your code, run

```
python autograder.py -q q3
```

Q4 (3 pts). Expectimax Agent

You will implement the `ExpecimaxAgent`, which is useful for modeling probabilistic behavior of agents who my make suboptimal choices.

- Random ghosts are of course not optimal minimax agents, and so modeling them with minimax search may not be appropriate.
- ExpectimaxAgent will no longer take the min over all ghost actions, but the expectation according to your agent's model of how the ghosts act.
- To simplify your code, assume you will only be running against an adversary which chooses amongst their `getLegalActions` uniformly at random.
- You should observe the two different behaviors of Minimax and Expectimax. In particular, if Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. Investigate the results of these two scenarios:

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

You should find that your `ExpectimaxAgent` wins about half the time, while your `AlphaBetaAgent` always loses. Make sure you understand why the behavior here differs from the minimax case.

- **Grading**
  - Again, the autograder will check your code to determine whether it explores the correct number of states.
  - You can debug your implementation on small the game trees using the command:

```
python autograder.py -q q4
```

Q5 (6 pts). Evaluation Function

Write a better evaluation function for Pacman in the provided function `betterEvaluationFunction`.

- The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did.
- With depth 2 search, your evaluation function should clear the `smallClassic` layout with one random ghost more than half the time and still run at a reasonable rate (to get full credit, Pacman should be averaging around 1000 points when he's winning).

- **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:
  - If you win at least once without timing out the autograder, you receive 1 points.
  - Any agent not satisfying these criteria will receive 0 points.
  - +1 for winning at least 5 times, +2 for winning all 10 times
  - +1 for an average score of at least 500, +2 for an average score of at least 1000 (including scores on lost games)
  - +1 if your games take on average less than 30 seconds on the autograder machine, when run with `--no-graphics`.
  - The additional points for average score and computation time will only be awarded if you win at least 5 times.
  - You can try your agent out under these conditions with

```
python autograder.py -q q5
python autograder.py -q q5 --no-graphics # without rendering
```

If you find any error in the grammar or project description,
please contact the instructor via email: hsmoon@cau.ac.kr

Plagiarism in code will result in an automatic F grade, regardless of the reason. CAU has the internal system of detecting possible reuse of code. You may consult tools like ChatGPT for guidance, but do not copy or submit generated content as-is. This may lead to high similarity across submissions, which will be treated as plagiarism.