AI23S – Homework#2: Multi-agent Search

Introduction

Modified from Berkeley CS188

In this assignment, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement both minimax and expectimax search and try your hand at evaluation function design.

Detail Info

Files you'll edit:

multiAgents.py	Where all of your multi-agent search agents will reside.

Files you might concern:

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 file 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.

Important Note

Files to Edit and Submit:

You will fill in portions of **multiAgents.py** during the assignment. You should submit this file with your code and comments. **Please do not change the other files in this distribution or submit any of our original files other than this file.**

Evaluation:

Your code will be autograded for technical correctness. **Please do not change the**

names of any provided functions or classes within the code, or you will wreak havoc on the autograder.

Academic Dishonesty:

We will be checking your code against other submissions in the class for logical redundancy. **If you copy someone else's code and submit it with minor changes, we will know**. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. **If you do, we will pursue the strongest consequences available to us.**

Problem 1: Reflex Agent (10 pts + 10 Bonus Pts):

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. A capable reflex agent will have to consider both food locations and ghost locations to perform well. Your agent should easily and reliably clear the testClassic layout:

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

Try out your reflex agent on the default mediumClassic layout with one ghost or two (and animation off to speed up the display):

```
python pacman.py --frameTime 0 -p ReflexAgent -k 1
python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

How does your agent fare? It will likely often die with 2 ghosts on the default board, unless your evaluation function is quite good.

Note: As features, try the reciprocal of important values (such as distance to food) rather than just the values themselves.

Options:

Default ghosts are random; you can also play for fun with slightly smarter directional ghosts using **-g DirectionalGhost.**
If the randomness is preventing you from telling whether your agent is improving, you can use **-f** to run with a fixed random seed (same random choices every game). You can also play multiple games in a row with **-n.**
Turn off graphics with **-q** to run lots of games quickly.

Grading:

we will run your agent on the openClassic layout 10 times.

You will receive basic

- **0 points** if your agent **times out,** or **never wins**.
- **5 points** if your agent wins at least **5 games**,

Bonus

- **8 points** if your agent wins **9 games**
- **10 points** if your agent wins all **10 games.**

You will receive additional

- **5 points** if your agent's average score is **greater than 500**

Bonus

- **8 points** if your agent's average score is **greater than 1000**
- **10 points** if your agent's average score is **greater than 1450**

You can try your agent out under these conditions with

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

To run it without graphics, use:

```
python autograder.py -q q1 --no-graphics
```

Don't spend too much time on this question, though, as the meat of the project lies ahead.

Problem 2: Minimax (30 pts)

Now you will write an adversarial search agent in the provided MinimaxAgent class tub in multiAgents.py. Your minimax agent should work with any number of ghost, 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.

Important:

1. A single search 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.**
2. Pacman is always agent 0, and **the agents move in order of increasing agent index**
3. **Notice the return condition in minimax, and evaluationFunction can only be used then**
4. **Do not call GameState.generateSuccessor() more than necessary.**

Hints and Observations:

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.

The evaluation function for the pacman test in this part is already written (self.evaluationFunction). You shouldn't change this function, but recognize that now we're evaluating *states* rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from
the current state.

The minimax values of the initial state in the minimaxClassic layout are 9, 8, 7, -492
for depths 1, 2, 3 and 4 respectively. Note that your minimax agent will often win (665/1000 games for us) despite the dire prediction of depth 4 minimax.

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

.
All states in minimax should be GameStates, either passed in to getAction or generated via GameState.generateSuccessor. In this project, you will not be abstracting to simplified states.

When Pacman believes that his death is unavoidable, he will try to end the game as soon as possible because of the constant penalty for living.

Sometimes, this is the wrong thing to do with random ghosts, but minimax agents always assume the worst:

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

Make sure you understand why Pacman rushes the closest ghost in this case.

Grading:

We will be checking your code to determine whether it explores the correct number of game states. This is the only reliable way to detect some very subtle bugs in implementations of minimax. As a result, **the autograder will be very picky about how many times you call GameState.generateSuccessor**. If you call it any more or less than necessary, the autograder will complain.

To test and debug your code, run:

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

This will show what your algorithm does on a number of small trees, as well as a pacman game

To run it without graphics, use:

```
python autograder.py -q q2 --no-graphics
```

Problem 3: Alpha-Beta Pruning (30 pts)

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 the 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.

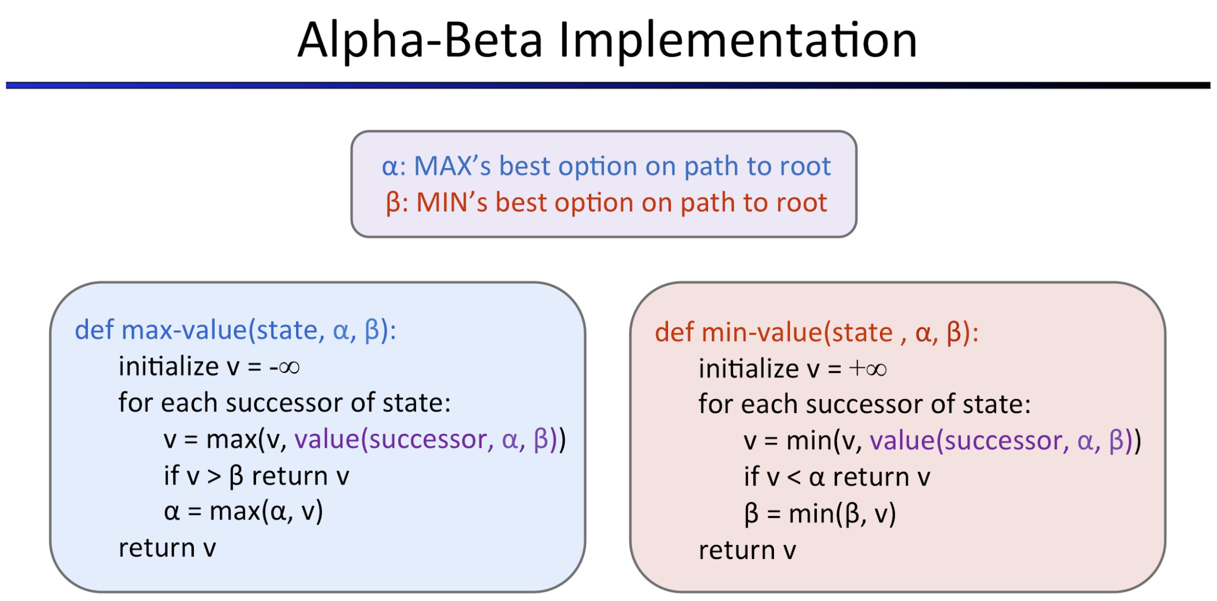
```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

The AlphaBetaAgent minimax values should be identical to the MinimaxAgent minimax values, although the actions it selects can vary because of different tie-breaking behavior. Again, the minimax values of the initial state in the minimaxClassic layout are 9, 8, 7 and -492 for depths 1, 2, 3 and 4 respectively.

Important

1. **successor states should always be processed in the order returned by GameState.getLegalActions()**
2. **You must not prune on equality in order to match the set of states explored by our autograder**. (Indeed, alternatively, but incompatible with our autograder, would be to also allow for pruning on equality and invoke alpha-beta once on each child of the root node, but this will not match the autograder.)
3. **Do not call GameState.generateSuccessor() more than necessary.**

The pseudo-code below represents the algorithm you should implement for this question.



Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v > β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v < α return v
        β = min(β, v)
    return v
```

Grading

Because we check your code to determine whether it explores the correct number of states, 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
```

This will show what your algorithm does on a number of small trees, as well as a pacman game. To run it without graphics, use:

```
python autograder.py -q q3 --no-graphics
```

The correct implementation of alpha-beta pruning will lead to Pacman losing some of the tests. This is not a problem: as it is correct behaviour, it will pass the tests.

Problem 4: Expectimax (30 pts):

Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case. In this question you will implement the ExpectimaxAgent, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.

As with the search and constraint satisfaction problems covered so far in this class, the beauty of these algorithms is their general applicability. To expedite your own development, we've supplied some test cases based on generic trees.

Debugging on these small and manageable test cases is recommended and will help you to find bugs quickly. Make sure when you compute your averages that you use floats.
Integer division in Python truncates, so that 1/2 = 0, unlike the case with floats where 1.0/2.0 = 0.5.

Once your algorithm is working on small trees, you can observe its success in Pacman. 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.

To see how the ExpectimaxAgent behaves in Pacman, run:

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

You should now observe a more cavalier approach in close quarters with ghosts. 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 -q -n 10
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

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.

The correct implementation of expectimax will lead to Pacman losing some of the tests. This is not a problem: as it is correct behaviour, it will pass the tests.

Grading

To test and debug your code, run

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

This will show what your algorithm does on a number of small trees, as well as a pacman game. To run it without graphics, use:

```
python autograder.py -q q4 --no-graphics
```

To Check if you Got All the Points

```
python autograder.py
```

Submission Deadline and Method

Language : python3.6

Package : Do not import any other package

Deadline : Sunday, 9 APR 2023 23:59 (UTC+8)

Delay policy : No late submission is allowed!

Submission Method:
Upload a zip file to NTU Cool with format :

r1234567_hw2.zip
- r1234567_hw2.py (this is your multiAgents.py file)

Do not include other python file except **m1234567_hw2.py (multiAgents.py)*
All file name should be in **lower case and **only zip file** (No rar and 7zip)*
*****Incompatible format will not be graded.***

Notice : Zero point for plagiarism!!!
(either from internet or copy from classmate)