COSE361(03) - Assignment #2

Due Date: 2022/4/18(Monday) 23:59

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 1~3 in *https://inst.eecs.berkeley.edu/~cs188/sp20/project2/*. 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.

Introduction

In this project, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement the minimax search .
The code base has not changed much from the previous project, but please start with a fresh installation, rather than intermingling files from project 1.
As in project 1, this project includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
$ python autograder.py
```

Note: If your `python` refers to Python 2.7, you may need to invoke `python3` `autograder.py` (and similarly for all subsequent Python invocations) or create a conda environment as described in Project 0.

It can be run for one particular question, such as q2, by:

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

It can be run for one particular test by commands of the form:

```
$ python autograder.py -t test_cases/q2/0-small-tree
```

By default, the autograder displays graphics with the `-t` option, but doesn't with the -q option. You can force graphics by using the `--graphics` flag, or force no graphics by using the `--no-graphics` flag.

See the autograder tutorial in Project 0 for more information about using the autograder. The code for this project contains the following files, available as a zip archive.

Files to Edit and Submit: You will fill in portions of `multiAgents.py` during the assignment. Please do not change the other files in this distribution or submit any of our original files other than this file.

Welcome to Multi-Agent Pacman

First, play a game of classic Pacman by running the following command:

```
$ python pacman.py
```

and using the arrow keys to move. Now, run the provided `ReflexAgent` in `multiAgents.py`

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

Note that it plays quite poorly even on simple layouts:

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

Inspect its code (in `multiAgents.py`) and make sure you understand what it's doing.

Question 1 (4 points): 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. 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: 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: We 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 addition 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

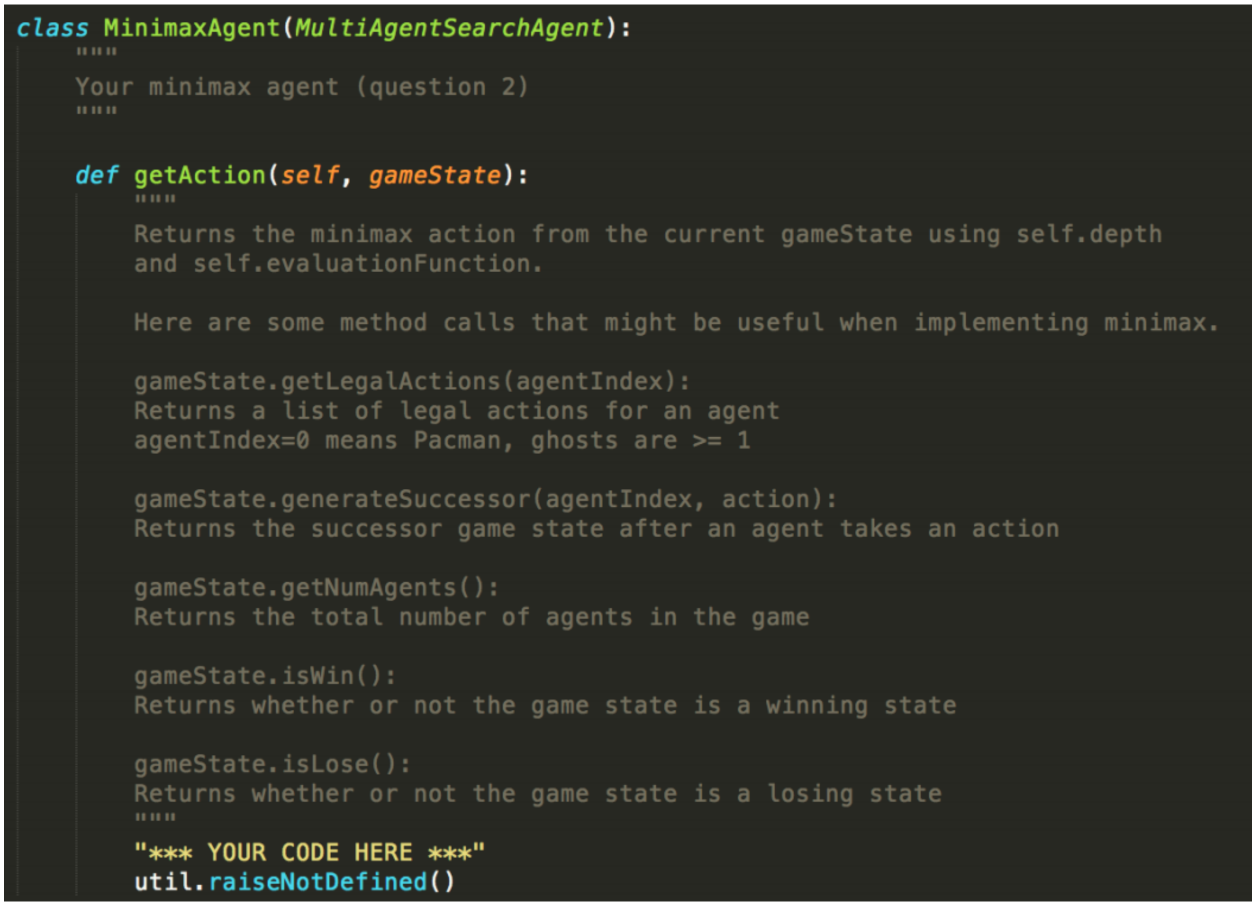
```
$ python autograder.py -q q1
```

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 q1 --no-graphics
```

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

Question 2 (5 points): Minimax



```python
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """

    def getAction(self, gameState):
        """
        Returns the minimax action from the current gameState using self.depth
        and self.evaluationFunction.

        Here are some method calls that might be useful when implementing minimax.

        gameState.getLegalActions(agentIndex):
        Returns a list of legal actions for an agent
        agentIndex=0 means Pacman, ghosts are >= 1

        gameState.generateSuccessor(agentIndex, action):
        Returns the successor game state after an agent takes an action

        gameState.getNumAgents():
        Returns the total number of agents in the game

        gameState.isWin():
        Returns whether or not the game state is a winning state

        gameState.isLose():
        Returns whether or not the game state is a losing state
        """
        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()
```

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub 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.

Important: 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.

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
```

Hints and Observations

- Hint: Implement the algorithm recursively using helper function(s).
- The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: as it is correct behavior, 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
  ```

- Pacman is always agent 0, and the agents move in order of increasing agent index.

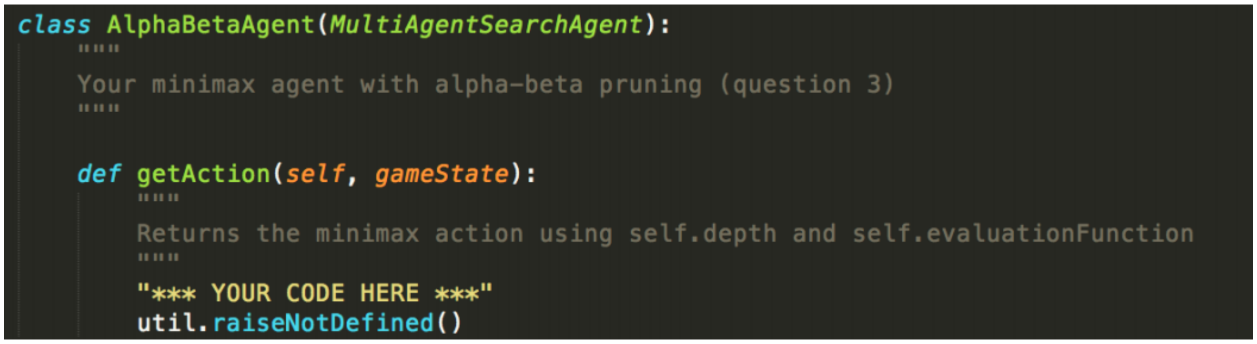
- All states in minimax should be `GameStates` , either passed in to `getAction` or generated via `GameState.generateSuccessor`. In this project, you will not be abstracting to simplified states.
- On larger boards such as `openClassic` and `mediumClassic` (the default), you'll find Pacman to be good at not dying, but quite bad at winning. He'll often thrash around without making progress. He might even thrash around right next to a dot without eating it because he doesn't know where he'd go after eating that dot. Don't worry if you see this behavior, question 5 will clean up all of these issues.
- 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.

Question 3 (5 points): Alpha-Beta Pruning

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



```
class AlphaBetaAgent(MultiAgentSearchAgent):
    """
    Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()
```

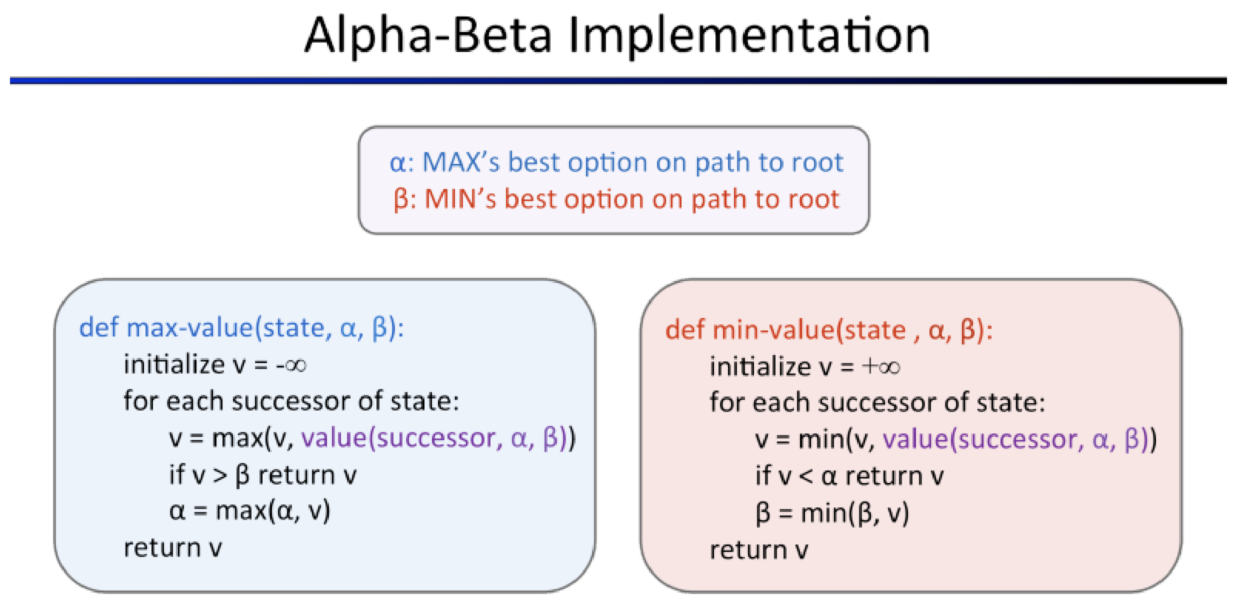
Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in `AlphaBetaAgent` . Again, your algorithm will be slightly more general than the pseudocode from lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer 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 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.

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`. Again, do not call `GameState.generateSuccessor` more than necessary.
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 alphabeta once on each child of the root node, but this will not match the
autograder.)
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
```

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 behavior, it will pass the tests.

Submission

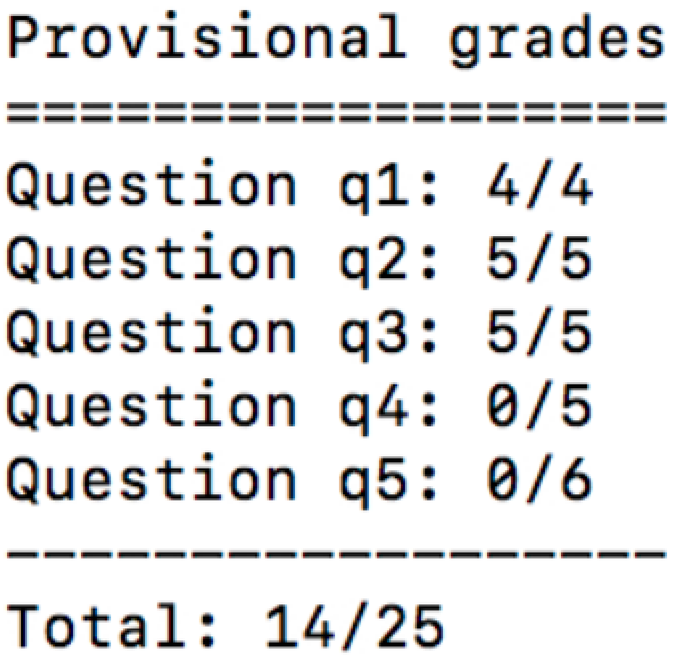
You only need to solve Question 1~3. To get full credit, run `autograder.py` and it should look like below.

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

```
Provisional grades
==================
Question q1: 4/4
Question q2: 5/5
Question q3: 5/5
Question q4: 0/5
Question q5: 0/6
------------------
Total: 14/25
```

▼ There are two files you need to submit.

1. Submit `multiagent.py` on Blackboard.
2. Submit a pdf file containing
   a. capture the result of `autograder.py` in terminal(4/5/5 points for Q1/Q2/Q3. Total 14 points). (***Note :*** Although you implement Question 1~3 in `multiagent.py` and get full grades with `autograder.py`, you will get 0 score without the capture on pdf.)



```
Provisional grades
==================
Question q1: 4/4
Question q2: 5/5
Question q3: 5/5
Question q4: 0/5
Question q5: 0/6
------------------
Total: 14/25
```

   b. Three discussions when playing Pacman (2 points for each discussion. Total 6 points.)

    - How can alpha-beta pruning make it more efficient to explore the minimax tree?
    - Is there a situation where the Reflex agent performs better than the minimax or alpha-beta pruning algorithm?
    - Ask yourself one question and answer.

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

Note:

- Do not compress your result files. Upload them respectively
- It is okay to make out a report in Korean

About Plagiarism

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, please upload your question on discussion board in Blackboard.