Project 2: Multi-Agent Search

Submit Assignment

Due Thursday by 11:59pm **Points** 22 **Submitting** a file upload **File Types** py
Available until Oct 25 at 11:59pm

Introduction

Download project code: p2_multiagent.zip

This project applies the concepts of adversarial search to the world of Pacman. You will implement Minimax, Expectimax, and design an evaluation function.

The provided code above is very similar to Project 1, but please use what is provided here instead of trying to make files from Project 1 work with your multi-agent algorithms.

An autograder is included in the code. You can run the autograder from the command line with the following prompt.

```
python3 autograder.py
```

The points you receive from the autograder reflect what you will receive for each question in this project. Keep in mind that not all questions in the autograder may be required. Be sure to consult the rubric for a breakdown of what is needed to complete the project.

You can also autograde a single question using the `-q` option with an argument of the form `q#`. See the example below, which only autogrades question 1.

```
python3 autograder.py -q q1
```

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.

Files that you will edit and submit

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

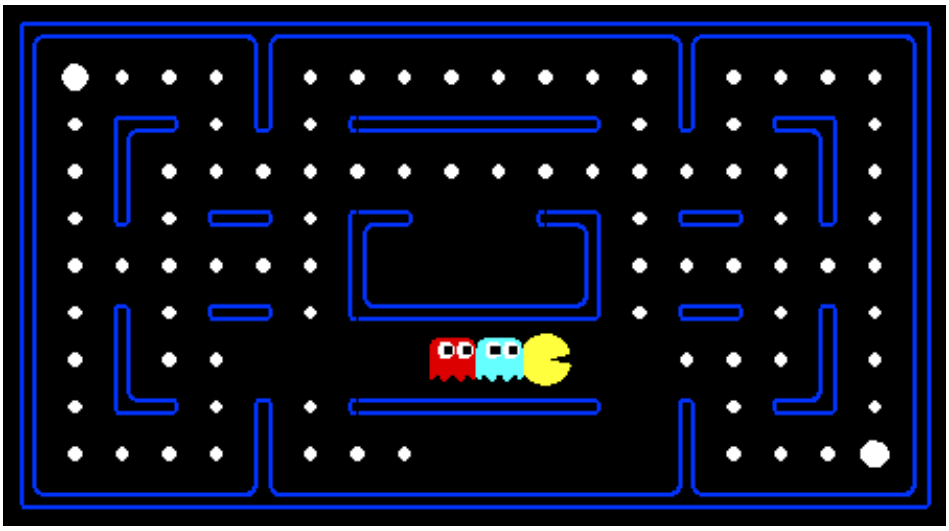
Files that you may want to look at

- `pacman.py` - The main file that runs Pacman games. This file describes a Pacman GameState type, which you will 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 don't need to use these data structures in this project, but you may find other functions defined here useful.

You are welcome to look at the other files included in the project, but you should only need to read through the files specified above in order to complete the project.

Do not edit any files other than the ones that you will submit. Your code will be graded against the provided project code, so if your code is dependent on changes you've made to files that are not required of you to submit, you will end up losing points.



Welcome to Multiagent Pacman

You should be able to play a game of Pacman by running the following command:

```
python3 pacman.py
```

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

```
python3 pacman.py -p ReflexAgent
```

Note that it plays quite poorly even on simple layouts:

```
python3 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 by improving the evaluation function. 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:

```
python3 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):

```
python3 pacman.py --frameTime 0 -p ReflexAgent -k 1
```

```
python3 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.

Notes:

- You will want to think of features to use as part of your evaluation function. That is, depending only on score is clearly not very effective. We want to incorporate other numerical values that serve as a means of indicating that a state is "good" or "bad". The typical means of combining these features is by summing the values of each feature together. If you want more control over the "importance" of each feature, you can scale each feature with a weight value.
- As features, you may want to try the reciprocal of important values (such as distance to food) rather than just the values themselves.
- The evaluation function you're writing is evaluating state-action pairs; in later parts of the project, you'll be evaluating states.
- 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(str(newGhostStates))`.

Some useful values:

- The position of each ghost in `newGhostStates` can be accessed using `getPosition()`.
- `newFood` is provided as `Grid` object of booleans. You can get elements of `newFood` using `newFood[i][j]` where i and j are rows and columns respectively. The width and height of the grid can be accessed via `newFood.width` and `newFood.height`.
- `newFood` can also be converted to a list of tuples (where each tuple is the location of a remaining food pellet) via `newFood.asList()`.

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: The autograder 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

```
python3 autograder.py -q q1
```

To run it without graphics, use:

```
python3 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

Be sure to reference the pseudocode provided in the slides for this question as well as questions 3 and 4. You will have to adapt the pseudocode to utilize aspects of the provided code. In particular, your recursive functions will need to use a parameter for keeping track of depth. The depth that the agent expands to is given by `self.depth`. *Also important is that* `getAction` *needs to return an action. So you're not returning the Minimax value, but rather the action associated with that overall value.*

A notable difference between what you need in your implementation and what is in the pseudocode is the dispatcher function `value`. *This is the function that* `getAction` *will use to determine which action to return. The pseudocode is more flexible and considers the case where we might start at a min node, but you can assume that we'll always start at a max node (since Pacman is the maximizing agent).*

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 (that is, going from one node to its children) 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: Your code will be checked 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

```
python3 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:

```
python3 autograder.py -q q2 --no-graphics
```

Hints and Observations

- The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: it is correct behavior, so it will pass the tests.
- You can store infinity or negative infinity as a float by using `float("inf")` and `float("-inf")`.
- 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.~~ (Question 5 is not required.)
- 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
  ```

Question 3 (5 points): 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 in the slides, 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.

```
python3 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 your code is checked 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 alpha-beta once on each child of the root node, but this will not match the autograder.)

Again, be sure to reference the slides for the pseudocode.

Question 4 (5 points): Expectimax

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, you are supplied with some test cases based on generic trees. You can debug your implementation on small game trees using the command:

```
python3 autograder.py -q q4
```

Debugging on these small and manageable test cases is recommended and will help you to find bugs quickly.

Once your algorithm is working on small trees, you can observe its success in Pacman. Random ghosts are of course not optimal minimax agents, 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:

```
python3 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:

```
python3 pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
```

```
python3 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: it is correct behavior, so it will pass the tests.

You are not required to complete Question 5. This question asks you to write `betterEvaluationFunction`. If you're curious about this question, feel free to ask your instructor about it.

Submission

When you are finished, submit your completed `multiAgents.py` file to Canvas.

Some Rubric			
Criteria	**Ratings**		**Pts**
Question 1 Completed as described	**4.0 pts** **Full Marks**	**0.0 pts** **No Marks**	4.0 pts
Question 2 Completed as described	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Question 3 Completed as described	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Question 4 Completed as described	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Coding Style Code is cleanly written with comments providing non-redundant supplementary explanation	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
			Total Points: 22.0