SWE3011 Homework 2: Multi-Agent Search*

JinYeong Bak (jy.bak@skku.edu)

Deadline: Oct 30 2022, 23:59†

1 Introduction

Please read the homework description carefully and make sure that your program meets all the requirements stated. The homework is an individual task. You can discuss the problem with your friends but you should not program together. You will get an F on the entire course if your homework includes any plagiarism.

2 Goal

In this project, 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 [1].

The code base has not changed much from the homework 1, but please start with a fresh installation, rather than intermingling files from homework 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
```

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

3 Description

All descriptions in this section are based on the page - `https://inst.eecs.berkeley.edu/~cs188/sp21/project2/`. You can read the page to understand basic tasks (questions 1, 2, 3, and 4) and extra task (questions 5). But we add some more requirements for each task. Please read this description carefully.

3.1 Welcome to Multi-Agent Pacman

You are able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

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.

*Source: `http://ai.berkeley.edu`

†You are encouraged to ask questions about this homework on icampus Q&A board. Please do not write your code on the board.

3.2 Question 1: 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.

Important note: Remember that newFood has the function asList()

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

Important note: The evaluation function you're writing is evaluating state-action pairs; in later parts of the project, you'll be evaluating states.

Important 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).

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 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

To run it without graphics, use:

python autograder.py -q q1 –no-graphics

Hint: Don't spend too much time on this question, though, as the meat of the project lies ahead. (It is true!)

3.3 Question 2: Minimax

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.

The actual ghosts operating in the environment may act partially randomly, but Pacman's minimax algorithm assumes the worst:

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

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

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 note: A single level of the search is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving twice.

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.getNextState. 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

- Implement the algorithm recursively using helper function(s). (Yes, Please!)
- 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.
- 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.getNextState. In this project, you will not be abstracting to different state representations, such as in project 1's search algorithms you implemented.
- 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.

3.4 Question 3: 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 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, child states should always be processed in the order returned by GameState.getLegalActions. Again, do not call GameState.getNextState 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.)

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.

3.5 Question 4: 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 algorithms 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. You can debug your implementation on the small game trees using the command:

python 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, and so modeling them with minimax search may not be appropriate. Rather than taking the min over all ghost actions, ExpectimaxAgent will take 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 that chooses among its 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.

3.6 Question 5: Evaluation Function

This task is an extra task. Please solve this question if you want to take additional points.

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 autograder is run on EC2, so this machine will have a fair amount of resources, but your personal computer could be far less performant (netbooks) or far more performant (gaming rigs).
- The additional points for average score and computation time will only be awarded if you win at least 5 times.
- Please do not copy any files from Project 1, as it will not pass the autograder on Gradescope.

You can try your agent out under these conditions with

python autograder.py -q q5

To run it without graphics, use:

python autograder.py -q q5 –no-graphics

4 Submission and Evaluation

You need to submit the followings:

- multiAgents.py

You MUST compress the above folders with your own source code files into 'HW2_yourid.zip' (e.g.), HW2_2022123456.zip and submit the compressed file to icampus [2].

If you do not follow the above submission policy, you will get penalty.

Your code will be autograded for technical correctness including the original testcases in autograder.py and our own testcases. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.

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. If you do, we will give F grade.

You are not alone! If you find yourself stuck on something, contact the TA for help. Please use icampus Q&A board. If you are in a struggle, other students are also the same. You can ask common questions, and you can find the solutions by searching. But please follow the communication rules in the first week's pdf file. If not, we might ignore your questions.

References

[1] Dawn Song Stuart Russell. Cs 188. `https://inst.eecs.berkeley.edu/~cs188/sp21/`, 2022.

[2] SKKU i-Campus. `https://icampus.skku.edu/`, 2022.