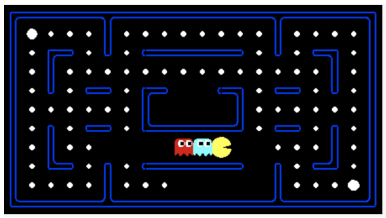
CSCI 360 Project #2: Multi-Agent Search

Released: February 18, 2022

Due: March 11, 2022



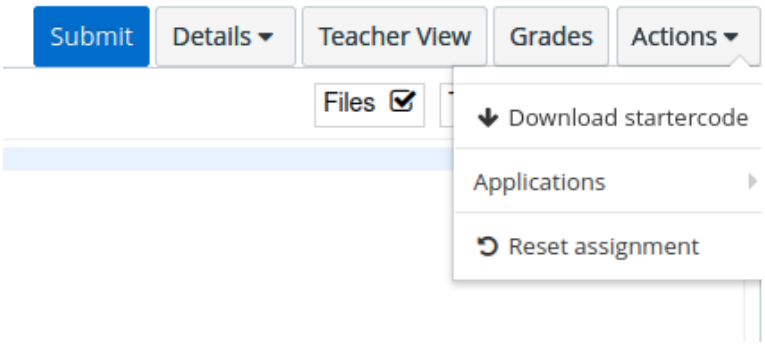
Contents

Introduction

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.

Getting Started: As with Project 1, to work on and submit this assignment:

1. on the Vocareum project 2 workspace, download the codebase by clicking "Download startercode" under "Actions" (screenshot below)



2. fill in the missing code segments within *multiAgents.py* as per the instructions in this handout
3. upload this file back to Vocareum and submit

As before, we also provides an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

If you want to run for one particular question, such as q2, by:

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

If you want to run for a particular test case, here's an example

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

Files you will edit:

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

Files you might want to look at:

- *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.

Supporting files you can ignore:

- *graphicsDisplay.py:* Graphics for Pacman
- *graphicsUtils.py:* Support for Pacman graphics
- *textDisplay.py:* ASCII graphics for Pacman

- *ghostAgents.py:* Agents to control ghosts
- *keyboardAgents.py:* Keyboard interfaces to control Pacman
- *layout.py:* Code for reading layout files and storing their contents
- *autograder.py:* Project autograder
- *testParser.py:* Parses autograder test and solution files
- *testClasses.py:* General autograding test classes
- *test_cases/ :* Directory containing the test cases for each question
- *multiagentTestClasses.py:* Project 2 specific autograding test classes

Question 1: Minimax (5 points)

In this question, 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.

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.

To test and debug your code (and also we will grade based on these test cases), run

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

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. If you are sure that your implementation is correct and you want to run the test cases without graphics, use no-graphics flags as it will save you a lot of time.

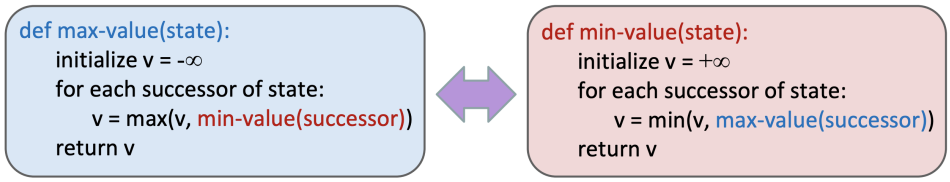
Hint:

- Recursion is extremely useful in implementing the trees !
- Even though your algorithm is correct, Pacman might lose during the game. Don't worry too much about it. Correct implementation will pass the tests.
- Pacman is always agent 0, and the agents move in order of increasing agent index.
- 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*

Hint: Here are some method calls that might be useful

- *gameState.getNextState* will return the next game state after an agent takes an action
- *gameState.getNumGhost* will return the number of ghost in the game (Note in the game you might have more than one ghost and the number does not include Pacman itself!)
- *gameState.getLegalActions* will return a list of legal actions for an agent. Please generate child states in the order returned by *GameState.getLegalActions*

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



$$V(s) = \max_{s' \in \text{successors}(s)} V(s') \qquad V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Question 2: Alpha-Beta Pruning (5 points)

In this question, you will need implement Alpha-Beta Pruning algorithm in the *AlphaBetaAgent* function in *multiAgents.py*. You can implement Alpha-Beta Pruning algorithm based on Minmax search, but more efficient than Minmax. Similarly, your algorithm should be extend to multiple agents.

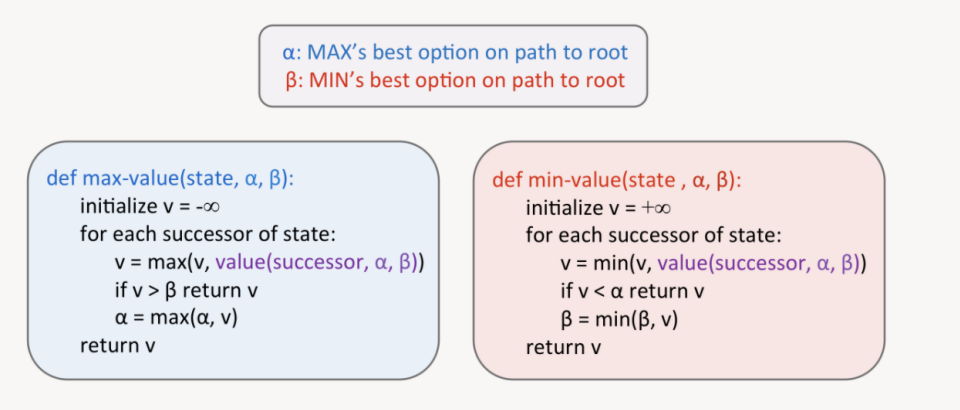
To test your code, you can run

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

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
```

You must not prune on equality in order to match the set of states explored by our autograder. The pseudo-code below represents the algorithm you should implement for this question.



Question 3: Expectimax(5 points)

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.

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.

Question 4: Evaluation Function(6 points)

Write a better evaluation function for pacman in the provided function betterEvaluationFunction. The evaluation function should evaluate states, rather than actions. 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).

You can test your evalution function with the following command.The autograder will run your agent on the smallClassic layout 10 times, therefore it may take a few seconds even you run without graphics.

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

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. Please do not copy any files from Project 1, as it will not pass the autograder.

The number of games won will be published to the Vocareum leaderboard for some more fun and friendly competition :)

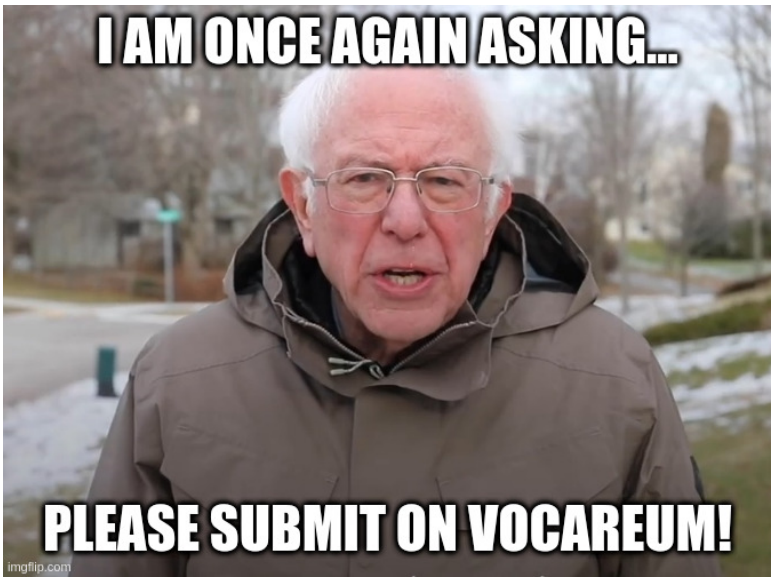
Bonus to be Released Next Week

As with Project 1, we will be releasing a bonus question one week after the initial release. it will likely be either the continuation of our mindfulness MDP exploration or an extension of our multi-agent search algorithms :)

Submission

When you're happy with your score that the *autograder* gives you, upload *multiAgents.py* to Vocareum and submit. This will run the same autograder and will publish the number of games won (out of 10) from Question 4 to the class leaderboard for more fun and friendly competition :) you are welcome to submit as many times as you'd like before the due date!

Note: apologies for the submission confusion last time! we have removed the print statement with the outdated submission instructions. Bernie is here to remind us to submit to Vocareum instead of BlackBoard!


I AM ONCE AGAIN ASKING...
PLEASE SUBMIT ON VOCAREUM!
imgflip.com