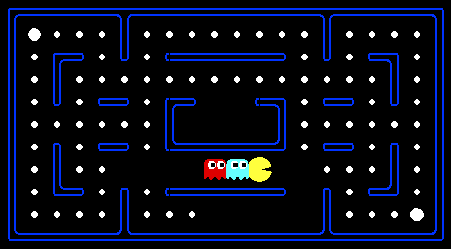
**Project 2: Multi-Agent Search**

Version 2.0. Last Updated: 19 Dec 2024

Due: **See Canvas**



Pacman, now with ghosts.

Minimax, Expectimax,

Evaluation

**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.

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 before these instructions assume you have cloned the repository as instructed in Project 0. Make a new branch for your changes and call it **p2**.

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:* On Mac or Linux, python may still refer to Python 2.7. To use Python 3, invoke python3 autograder.py instead. If you are using Anaconda, you should not have this problem.

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

python autograder.py -q q2

It can be run for one specific 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 from the GitHub repository.

|  |  |
| --- | --- |
| **Files you'll 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 |

**Files to Edit and Submit:** You will fill in portions of multiAgents.py during the assignment. Make a new **p2** branch in your repository for your submission. Once you have completed the assignment, commit your changes and push them to your remote repository. Put a link to your repository in the **p2** assignment on Canvas. Please *do not* change the other files in p2.

**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. However, the correctness of your implementation – not the autograder’s judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

**Academic Dishonesty:** Copying someone else’s code and submitting it as your own is asking for a grade you did not earn and claiming mastery of skills of which you have not demonstrated mastery. We may or may not use a plagiarism tool on your code in this class.

**Getting Help:** You are not alone, though we do expect you to know and practice basic problem-solving skills. If you find yourself stuck on something, contact the instructor or a classmate for help. Class time, Office Hours, Discord and Teams are there for your support; please use them. If you need to, set up an appointment for help. These projects should be rewarding and instructional, not frustrating and demoralizing—but I don’t know when or how to help unless you ask.

**Email, Teams and Discussion Boards:** Please be careful not to post spoilers nor executable code.

**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 ReflexAgent 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.pyto 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).

*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.   
1 point if your agent wins at least 5 times.  
2 points if your agent wins all 10 games.   
  
An additional:   
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

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

**Question 2 (5 points): 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.

Your code should also expand the game tree to an arbitrary user-defined depth. MinimaxAgent extends MultiAgentSearchAgent, which gives access to self.depth and self.evaluationFunction. These two variables are set by command line options. Score the leaves of your minimax tree with the supplied self.evaluationFunction, which defaults to scoreEvaluationFunction.

*Important:* One ply is one Pacman move and all the ghosts’ responses. A depth 2 search will involve Pacman and each ghost moving two times, or in other words 2 plys.

*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 several small trees, as well as a pacman game. To run it without graphics, use:

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

***Hints and Observations***

* Implement the algorithm recursively using helper functions.
* The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: It is correct behavior because heuristics are sometimes wrong--it will pass the tests.
* The evaluation function for the Pacman test in this part is already written for you in self.evaluationFunction. Do not change this function but recognize that now we’re evaluating *states* rather than actions. 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. Your minimax agent will often win—a recent canonical version wins 665/1000 games--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 the default mediumClassic, Pacman is good at not dying, but quite bad at winning. He often thrashes around without making progress. He might even thrash around right next to a dot without eating it because he does not know where to go after eating that dot. Don’t worry if you see this behavior, question 5 will clean up all these issues.
* When Pacman believes that 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): αβ-Pruning**

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

You should see a speed-up, for example perhaps depth 3 αβ will run as fast as depth 2 minimax. Ideally depth 3 on smallClassic should run in just a few seconds per move or faster.

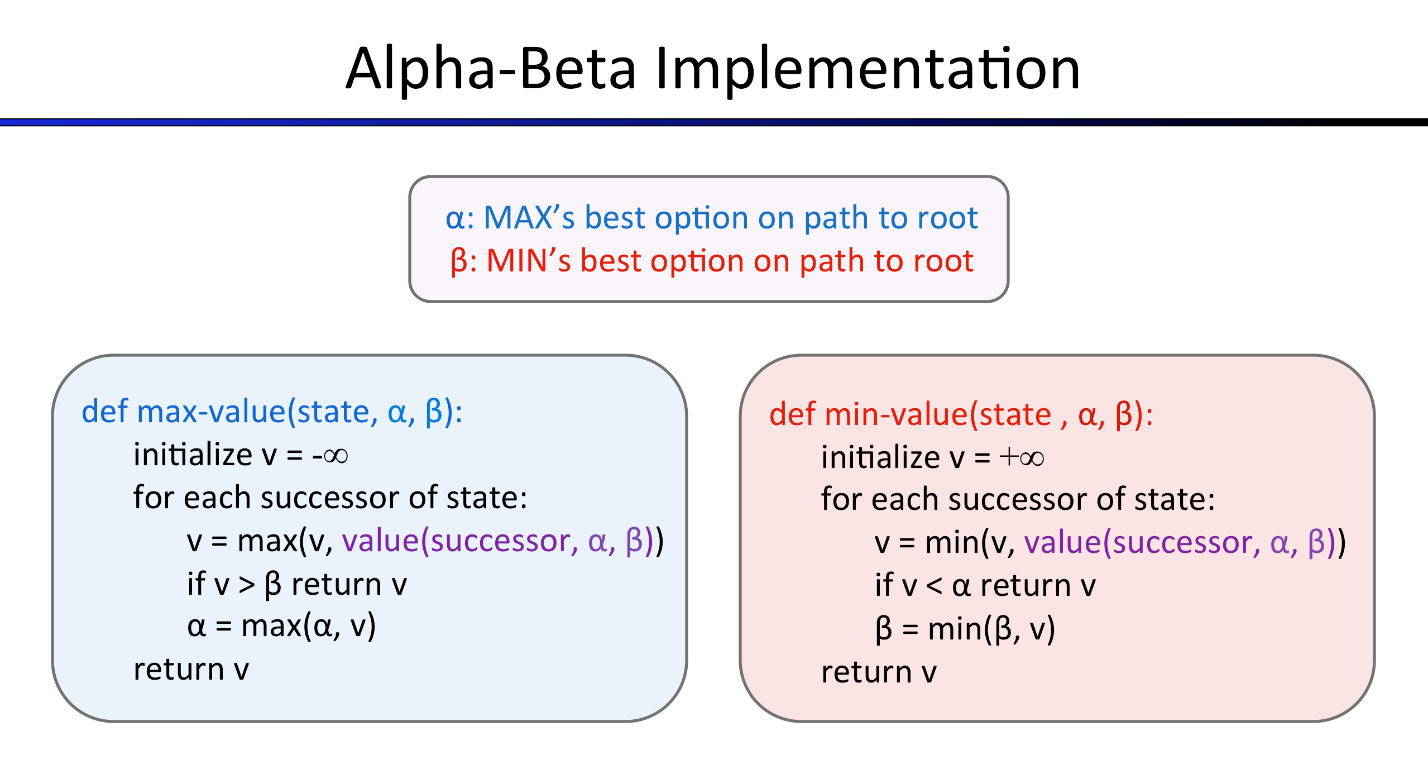
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic

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 αβ-pruning **without reordering children**. In other words, successor states should always be processed in the order returned by GameState.getLegalActions. Do not call GameState.generateSuccessor more than necessary.

***To match the set of states explored by our autograder, you must not prune on equality.*** Also incompatible with the autograder would be to allow for pruning on equality and invoke αβ once on each child of the root node.

The pseudocode below represents the algorithm you should implement for this question.



To test and debug your code, run

python autograder.py -q q3

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

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

The correct implementation of αβ-pruning will lead to Pacman losing some test scenarios sometimes. This is not a problem: it is expected behavior for heuristic algorithms and it will pass the tests.

**Question 4 (5 points): Expectimax**

Minimax and αβ are great if you 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 agents who may make suboptimal probabilistic choices.

As with the search 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. You can debug your implementation on small the 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 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 your adversary chooses actions from 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. 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, as with other heuristics. This is expected behavior and it will pass the tests.

**Question 5 (6 points): Evaluation Function**

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 point. 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 *may* run on AWS EC2, and your personal computer could be less performant or more performant.
* 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.

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

**Submission**

To submit your project, run python autograder.py on your solution, then zip all the files as instructed above and submit the zip file to the Project 2 assignment in Canvas.

Check with your instructor on where and how code files are to be submitted if different from what is written in this document.

If you work with a partner:

* In Canvas submission, please specify any partner you may have worked with.
* Each person must make their own submission, with their own code that they wrote.