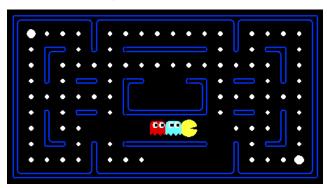# Project 2: Multi-Agent Pacman

(Thanks to John DeNero and Dan Klein!)

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. You can, however, use your search.py and searchAgents.py in any way you want. The code for this project contains the following files. Download the project either from blackboard or from here.

Files you'll edit:

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

Files 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.

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

What to submit:

You will fill in portions of multiAgents.py during the assignment. You should submit this file with your code and comments to blackboard. You may also submit other supporting files such as: search.py, searchAgents.py, Algorithms.py (if you are using your Project 1 code, then these should be included). Please *do not* change the other files in this distribution or submit any of our original files other than multiAgents.py. ***If you work with a partner, please write their name in a file named partner.txt***

Evaluation: The project is out of 50. You will be auto grading your code for technical correctness (using autograder.py that you have used for Project 0). Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. 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. The description of evaluation functions in Question 1 and Question 5 are also part of your grade (we will deduct points if we didn't find them).

Academic Dishonesty: 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; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours and Piazza are there for your support; please use them. If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask. One more piece of advice: if you don't know what a variable does or what kind of values it takes, print it out.

Multi-Agent Pacman

First, play a game of classic Pacman:

```
python pacman.py
```

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: Reflex Agent Evaluation Function (8 points)

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: you can never have more ghosts than the layout permits.

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.

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 2 points if your agent wins at least 5 times.

You will receive an addition 2 point if your agent's average score is greater than 500, or 4 points if it is greater than 1000.You can try your agent out under these conditions with

python pacman.py -p ReflexAgent -l openClassic -n 10 -q

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

Note: Make sure to describe your evaluation function.

Question 2: Minimax (10 points)

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 appears in the textbook. 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 MultiAgentAgent, 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 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 way 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.getLegalActions. If you call it any more or less than necessary, the autograder will complain. Note, however, that the autograder will accept solutions both with and without the Directions.STOP action available.

Hints and Observations

- The evaluation function 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

- To increase the search depth achievable by your agent, remove the Directions.STOP action from Pacman's list of possible actions. Depth 2 should be pretty quick, but depth 3 or 4 will be slow. Don't worry, the next question will speed up the search somewhat.

- 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: AlphaBeta Pruning (10 points)

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 pseudo-code in the textbook, 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

Question 4: Exceptimax (10 points)

Random ghosts are of course not optimal minimax agents, and so modeling them with minimax search may not be appropriate. Fill in ExpectimaxAgent, where your agent 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 RandomGhost ghosts, which choose amongst their getLegalActions uniformly at random.

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.

Question 5: Better Evaluation Function (12 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. You may use any tools at your disposal for evaluation, including your search code from the last project. With depth 2 search, your evaluation function should clear the smallClassic layout with two random ghosts 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).

> python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -q -n 10

Note: Make sure to describe your evaluation function. We're very curious about what great ideas you have, so don't be shy.

Grading: we 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 2 points. Any agent not satisfying these criteria will receive 0 points.
- +2 for winning at least 5 times.
- +2 for an average score of at least 500, +4 for an average score of at least 1000 (including scores on lost games)
- +2 if your games take on average less than 30 seconds on an inst machine.
- The additional points for average score and computation time will only be awarded if you win at least 5 times.

Hints and Observations

- As for your reflex agent evaluation function, you may want to use the reciprocal of important values (such as distance to food) rather than the values themselves.
- One way you might want to write your evaluation function is to use a linear combination of features. That is, compute values for features about the state that you think are important, and then combine those features by multiplying them by different values and adding the results together.