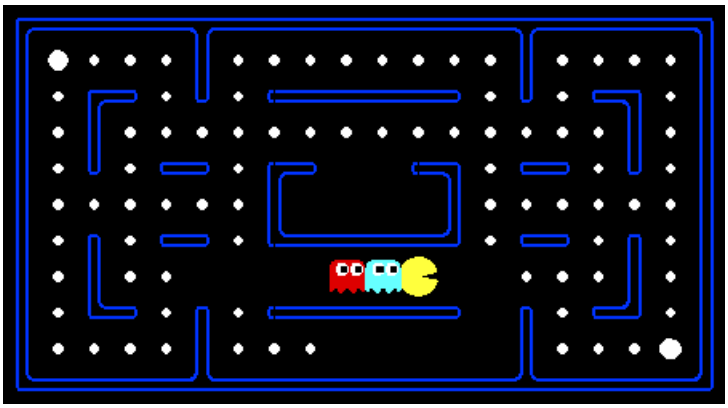
Project 2: Fightin' Pac-Man



Pac-Man, now with ghosts,
Minimax, Expectimax,
Evaluation.

Introduction

In this project, you will design agents for the classic version of Pac-Man, 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.

The code for this project contains the following files, available as a zip archive from the Canvas page.

As in project 1, this project includes an autograder for you to evaluate your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

It can be run for one particular questions, such as q2 by:

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

It can be run for one particular 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.

Key files to read

`multiAgents.py`	Where all of your multi-agent search agents will reside.
`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
`autograder.py`	Project autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test cases
`test_cases/`	Directory containing the test cases for each question
`multiagentTestClasses.py`	Project 2 specific autograding test classes

What to submit

You will fill in portions of `multiAgents.py` for the assignment. You should submit **this file only** with your code and comments, using the department server. Please *do not* change the other files in this distribution or submit any other files besides this file.

Multi-Agent Pac-Man

First, play a game of classic Pac-Man:

```
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 (3 points)

Improve the `ReflexAgent` in `multiAgents.py` to play respectably. The provided reflex agent code gives 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 two ghosts on the default board, unless your evaluation function is quite good.

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

Hint 1: As features, try the reciprocal of important values (such as distance to food or capsules) rather than just the values themselves.

Hint 2: The evaluation function you're writing is evaluating state-action pairs; in the 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 cna 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.

The autograder will check that your agent can rapidly clear the `openClassic` layout ten times without dying more than twice or thrashing around infinitely (i.e. repeatedly moving back and forth between two positions, making no progress).

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

Don't spend too much time on this question! The meat of the project lies ahead.

Question 2 (5 points)

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in `multiAgents.py`. Specifically, your mission is to implement the minimax algorithm described in the lectures, and demonstrated in class. As an example, see the `tictactoe.py` demo and the original AIMA code, `aima-examples.zip` (in Canvas). Note: the tictactoe code (and provided minimax decision implementation) is not sufficient – do not use as is, or your solution will be wrong!

Specific differences/extensions from the "vanilla" minimax algorithm:

- Your minimax agent should work with **any number of ghosts**, so you'll have to write an algorithm that is more general than what appears in the lectures. In particular, your minimax tree should have *multiple min layers* (one for each ghost) for every max layer.
- Your code should only **expand the game tree to a fixed depth**, which will be specified at the command line. 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 uses these two variables where appropriate, as these variables are populated for you, from the command-line options.
- Important: A single search ply is considered to be one Pac-Man move and all the ghosts' responses, so **depth 2 search** would involve **both Pac-Man** and **each ghost moving two times**, that is:

Ply 1:

1. Agent 0 (Pacman) move
2. Agent 1 (Ghost 1) move
3. Agent 2 (Ghost 2) move

Ply 2:

1. Agent 0 (Pacman) move
2. Agent 1 (Ghost 1) move
3. Agent 2 (Ghost 2) move

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.getLegalActions`. 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 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 Pac-Man'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.
- Pac-Man 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 Pac-Man 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 Pac-Man 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 Pac-Man rushes the closest ghost in this case.

Question 3 (3 points)

Implement an agent that uses **alpha-beta pruning** to more efficiently explore the minimax tree, in `AlphaBetaAgent`. Again, your algorithm will be *more general* than the pseudo-code in the slides, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

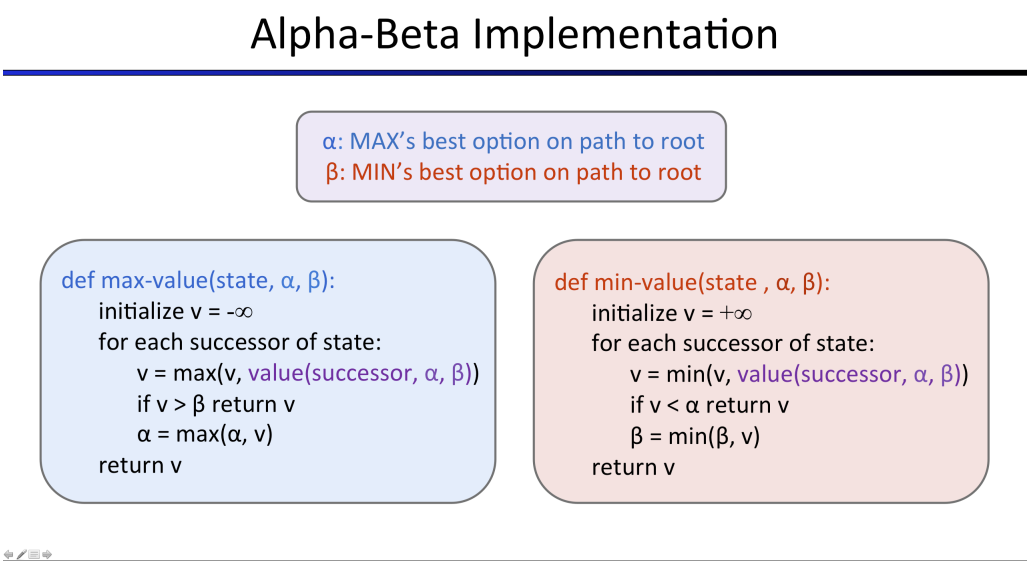
You should see a speed-up (perhaps depth 3 alpha-beta will run as fast as depth 2 minimax). Ideally, depth 3 on `smallClassic` should run in just a few seconds per move or faster.

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

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.

You must *not* prune on equality in order to match the set of states explored by out 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.)

The pseudo-code 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 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.

Question 4 (3 points)

Random ghosts are of course not optimal minimax agents, 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 agent, 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 Pac-Man 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 (6 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, Pac-Man should be averaging around 1000 points when he's winning).

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

Document your evaluation function! We're curious about what great ideas you have, so don't be shy. We reserve the right to award bonus points for clever solutions, and show demonstrations in class.

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. You might decide what to multiply each feature by based on how important you think it is.

Mini Contest (3 points extra credit)

Pac-Man's been doing well so far, but things are about to get a bit more challenging. This time, we'll pit Pac-Man against smarter foes in a trickier maze. In particular, the ghosts will actively chase Pac-Man instead of wandering around randomly, and the maze features more twists and dead-ends, but also extra pellets to give Pac-Man a fighting chance. You're free to have Pac-Man use any search procedure, search depth, and evaluation function you like. The only limit is that games can last a maximum of 3 minutes (with graphics off), so be sure to use your computation wisely. We'll run the contest with the following command:

```
python pacman.py -l contestClassic -p ContestAgent -g DirectionalGhost -q -n 10
```

The three students with the highest scores (details: we run 10 games; games longer than 3 minutes get score 0; lowest and highest 2 scores discarded; the remaining 6 scores averaged) will receive extra credit points (3 point for first place, 2 for second and 1 for third). Be sure to document what you are doing, as we may award additional extra credit to creative solutions even if they're not in the top 3. We may also demonstrate the performance of winning or creative solutions in class!