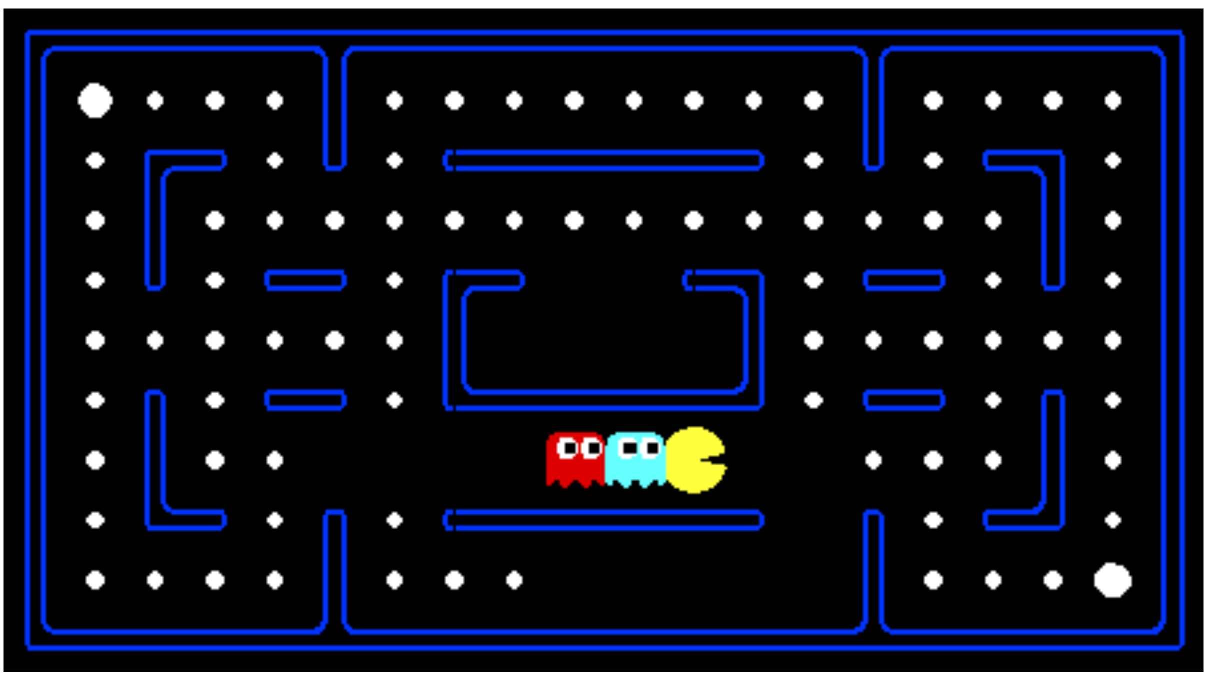
Programming Assignment 2 (Game Tree Search)

Due Oct 30 by 11:59pm **Points** 25 **Available** after Oct 9 at 12am

Programming Assignment 2: Game Tree Search (Individual assignment)

Modified version of UC Berkeley CSC188 **Project 2 (https://inst.eecs.berkeley.edu/~cs188/fa20/project2/)**

Table of Contents



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 the provided zip, rather than intermingling files from Assignment 1. You can also use code you wrote in Assignment 1, but make sure that you include that code in the file multiAgents.py since that is the only file that will be submitted.

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:

```
python3 autograder.py
```

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

```
python3 autograder.py -q q2
```

It can be run for one particular test by commands of the form:

```
python3 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.

The code for this project contains the following files, available as a [zip archive.]

Files you'll edit:	
multiAgents.py	Where all of your multi-agent search agents and helper code will reside.
Files you should not change:	
`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.
Supporting files that you can probably 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. You may also add other functions and code to this file so as to create a modular implementation. You will submit this file with your modifications. Please *do not* change the other files in this distribution Markus will not allow you to submit any other files, so if your implementation uses other files or changes to the original files it will not work during the marking phase.

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. The mark given to you by the autograder might be changed if we find any issues with our submission.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. There will be scheduled help sessions (to be announced), the piazza discussion forum will be monitored and questions answered, and you can also ask questions about the assignment during office hours. These things are for your support; please take advantage of them. If you can't make our office hours, let us know and we will arrange a different appointment. We want the assignment to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Piazza Discussion: Please be careful not to post spoilers.

What to Submit

You will be using MarkUs to submit your assignment. You will submit one file:

1. Your modified `multiAgentS.py`

Note: In the various parts below we ask a number of questions. You do not have to hand in answers to these questions, rather these questions are designed to help you understand the material.

Multi-Agent Pacman

First, play a game of classic Pacman:

```
python3 pacman.py
```

Now, run the provided `ReflexAgent` in `multiAgents.py`:

```
python3 pacman.py -p ReflexAgent
```

Note that it plays quite poorly even on simple layouts:

```
python3 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

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

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:

```
python3 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):

```
python3 pacman.py --frameTime 0 -p ReflexAgent -k 1
```

```
python3 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: 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 (i.e., how good is it to perform this action in this state); in later parts of the project, you'll be evaluating states (i.e., how good is it to be in this state). In some sense these are the same things except for the signature of the evaluationFunction. In particular, an evaluation function that determines the value or merit of a state can also be used to determine which action to execute by examining the state that the action would transition to: if the action transitions to a high value state it is better to do the action than if the action transitions to a low value state.

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: the autograder runs 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

```
python3 autograder.py -q q1
```

To run it without graphics, use:

```
python3 autograder.py -q q1 --no-graphics
```

Question 2 (5 points): Minimax

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in `multiAgents.py`. Your minimax search must work with any number of ghosts. In particular, for every max layer (where the pacman moves) your minimax tree will have multiple min layers, one for each ghost.

`gameState` does not keep track of whose turn it is to play, you will have to keep track of that in your minimax search. In particular, the pacman (MAX) plays first, followed by each ghost getting a turn; then the pacman plays again, followed by each ghost getting a turn, etc.

Score the leaves of your minimax tree with the supplied `self.evaluationFunction`, which defaults to `scoreEvaluationFunction`. You will have to implement a depth-bound, so the leaves of your minimax tree could be either terminal or non-terminal nodes. Hence, `self.evaluationFunction` will act as the game utility function, except that it will be called both on terminal and non-terminal nodes. The utility of terminal nodes is can be obtained from the `.getScore()` method, while on non-terminal nodes the evaluation is heuristic.

Terminal nodes are nodes where either `gameState.isWin()` or `gameState.isLose()` is true. However, the leaves of your tree search might also be non-terminal nodes.

Your Minimax (and all other game tree search algorithms you will implement) must utilize a depth-bound. The depth-bound you must operate under is stored in the variable `self.depth`. The depth-bound specifies number of times the pacman (MAX) gets to play. For example, if the depth-bound is 2, then MAX gets to make 2 moves and all of the ghosts get 2 moves each. When MAX is about to play a 3rd time, your search will terminate: instead of considering the possible 3rd moves of MAX it will simply return the value of `self.evaluationFunction` treating this node as if it was a terminal node. As another example, if the depth-bound is zero, your search will immediately return the `self.evaluationFunction` value of the root node.

Make sure your minimax code makes reference to the two variables, `self.depth` and `self.evaluationFunction` where appropriate as these variables will vary in response to command line options.

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.generateSuccessor`. If you call it any more or less than necessary, the autograder will complain. To test and debug your code, run

```
python3 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:

```
python3 autograder.py -q q2 --no-graphics
```

Hints and Observations

- 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.

  ```
  python3 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 `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 (5 points): Alpha-Beta Pruning

Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in `AlphaBetaAgent`. Again, your algorithm must use the depth-bound specified in `self.depth` and evaluate its leaf nodes with `self.evaluationFunction`.

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. 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`. Again, do not call `GameState.generateSuccessor` more than necessary.

To test and debug your code, run

```
python3 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:

```
python3 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 behaviour, it will pass the tests.

Question 4 (5 points): 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 and constraint satisfaction problems covered 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:

```
python3 autograder.py -q q4
```

Debugging on these small and manageable test cases is recommended and will help you to find bugs quickly. **Make sure when you compute your averages that you use floats.** Integer division in Python truncates, so that `1/2 = 0`, unlike the case with floats where `1.0/2.0 = 0.5`.

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. `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 you will only be running against an adversary which chooses amongst their `getLegalAction`s uniformly at random.

To see how the ExpectimaxAgent behaves in Pacman, run:

```
python3 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:

```
python3 pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
```

```
python3 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 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. 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 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).

```
python autograder.py -q q5
```

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. The autograder is run on the teach.cs machines which have a fair amount of resources, but your personal computer could be far less performant or far more performant. You can use your teach.cs login to run your program on the teach.cs machines.
- 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. You might decide what to multiply each feature by based on how important you think it is.

Submission

You're not done yet! You will also need to submit your code to MarkUs.