

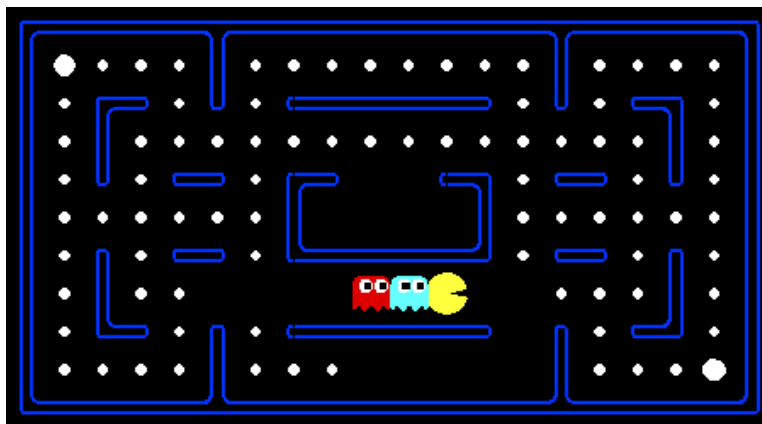
Multi-agent Pac-Man

Stanford CS221 Fall 2014-2015

Owner TA: Ilan Goodman (original assignment by John DeNero and Dan Klein)

Note: grader.py only provides basic tests. Passing grader.py does not by any means guarantee full points.

Last Update: 10/20/2014 17:20



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

Introduction

For those of you not familiar with Pac-Man, it's a game where Pac-Man (the yellow circle with a mouth in the above figure) moves around in a maze and tries to eat as many *food pellets* (the small white dots) as possible, while avoiding the ghosts (the other two agents with eyes in the above figure). If Pac-Man eats all the food in a maze, it wins. The big white dots at the top-left and bottom-right corner are *capsules*, which give Pac-Man power to eat ghosts in a limited time window (but you won't be worrying about them for the required part of the assignment). You can get familiar with the setting by playing a few games of classic Pac-Man, which we come to just after this 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.

The base code for this project contains a lot of files (which are listed towards the end of this page); you, however, **do not** need to go through these files to complete the assignment. These are there only to guide the more adventurous amongst you to the heart of Pac-Man. As in previous assignments, you will be modifying only `submission.py`. This assignment has no `grader.py` checking for functionality.

A basic `grader.py` has been included in the .zip file, but it only checks for timing issues. Passing this doesn't mean you have a correct implementation.

Remember that the assignment is due on Tuesday at 11PM. Please also note that solution code length varies widely on this assignment, so do not worry if your code works with as few as 20 lines or if you need 80 lines to get everything right.

Warmup

First, play a game of classic Pac-Man to get a feel for the assignment:

```
python pacman.py
```

You can always add `--frameTime -1` to the command line to run in "demo mode" where the game pauses after every frame.

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

```
python pacman.py -p ReflexAgent
```

Note that it plays quite poorly even on simple layouts:

```
python pacman.py -p ReflexAgent -l testClassic
```

You can also try out the reflex agent on the default `mediumClassic` layout with one ghost or two.

```
python pacman.py -p ReflexAgent -k 1
```


```
python pacman.py -p ReflexAgent -k 2
```

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

Options: Default ghosts are random; you can also play for fun with slightly smarter directional ghosts using `-g DirectionalGhost`. You can also play multiple games in a row with `-n`. Turn off graphics with `-q` to run lots of games quickly.


So, now that you are familiar enough with the interface, inspect the `ReflexAgent` code carefully (in `submission.py`) and make sure you understand what it's doing. The reflex agent code provides some helpful examples of methods that query the `GameState` (a `GameState` specifies the full game state, including the food, capsules, agent configurations and score changes: see `submission.py` for further information and helper methods) for information, which you will be using in the actual coding part. We are giving an exhaustive and very detailed description below, for the sake of completeness and to save you from digging deeper into the starter code. The actual coding part is very small - so please be patient if you think there is too much writing.

Problem 1: Minimax

- a.  **(5 points)** Before you code up Pac-Man as a minimax agent, notice that instead of just one adversary, Pac-Man could have multiple ghosts as adversaries. So we will extend the minimax algorithm from class (which had only one min stage for a single adversary) to the more general case of multiple adversaries. In particular, *your minimax tree will have multiple min layers (one for each ghost) for every max layer.*

Specifically, consider the limited depth tree minimax search with evaluation functions taught in class. Suppose there are $n + 1$ agents on the board, a_0, \dots, a_n , where a_0 is Pac-Man and the rest are ghosts. Pac-Man acts as a max agent, and the ghosts act as min agents. A single *depth* consists of all $n + 1$ agents making a move, so depth 2 search will involve Pac-Man and each ghost moving two times. In other words, a depth of 2 corresponds to a height of $2(n + 1)$ in the minimax game tree.

Write the recurrence for $V_{\text{opt}}(s, d)$, which is the minimax value with search stopping at depth d_{max} . You should express your answer in terms of the following functions: `IsEnd(s)`, which tells you if s is an end state; `Utility(s)`, the utility of a state; `Eval(s)`, an evaluation function for the state s ; `Player(s)`, which returns the player whose turn it is; `Actions(s)`, which returns the possible actions; and `Succ(s, a)`, which returns the successor state resulting from taking an action at a certain state. You may use any relevant notation introduced in lecture.

- b.  **(10 points)** Now fill out `MinimaxAgent` class in `submission.py` using the above recurrence. Remember that your minimax agent should **work with any number of ghosts**, and your minimax tree should 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`. The class `MinimaxAgent` extends `MultiAgentSearchAgent`, 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 from the command line options. Other functions that you might use in the code: `GameState.getLegalActions()` which returns all the possible legal moves, where each move is `Directions.X` for some X in the set `{NORTH, SOUTH, WEST, EAST, STOP}`. Go through `ReflexAgent` code as suggested before to see how the above are used and also for other important methods like `GameState.getPacmanState()`, `GameState.getGhostStates()` etc. These are further documented inside the `MinimaxAgent` class.

Hints and Observations

reflex agent use a function of food, scared time, etc to estimate a state. not recursive
minimax agent use recursion until leaf node to find the TRUE utility for each state,
time consuming, can be improved by alpha beta pruning

- o 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. Use `self.evaluationFunction` in your definition of V_{opt} wherever you used `Eval(s)` in part 1a.
- o The minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. **You can use these numbers to verify if your implementation is correct.** Note that your minimax agent will often win (15/20 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 problem will speed up the search somewhat.
- Pac-Man is always agent 0, and the agents move in order of increasing agent index. Use `self.index` in your minimax implementation, but only Pac-Man will actually be running your `MinimaxAgent`.
- Functions are provided to get legal moves for Pac-Man or the ghosts and to execute a move by any agent. See `GameState` in `pacman.py` for details.
- 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.
- `getAction` should use V_{opt} to determine the best action for Pac-Man.
- 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. Don't worry if you see this behavior. Why does Pac-Man thrash around right next to a dot?
- Consider the following run:

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

Why do you think Pac-Man rushes the closest ghost in minimax search on `trappedClassic`? (These questions are here for you to ponder upon; no need to include in the write-up.)

Problem 2: Alpha-Beta pruning



- a.  **(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 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 `mediumClassic` should run in just a few seconds per move or faster.

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

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.

Problem 3: Expectimax


- a.  **(5 points)** Random ghosts are of course not optimal minimax agents, so modeling them with minimax search is not optimal. Instead, write down the recurrence for $V_{\text{opt},\pi}(s)$, which is the maximum expected utility against ghosts that each follow the random policy which chooses a legal move uniformly at random. Your recurrence should resemble that of Problem 1a.
- b.  **(10 points)** 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. Assume Pac-Man is playing against `RandomGhosts`, which each choose `getLegalActions` uniformly at random.

You should now observe a more cavalier approach to 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:

```
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3
```

You may have to run this scenario a few times to see Pac-Man's gamble pay off. Pac-Man would win half the time on an average and for this particular command, the final score would be -502 if Pac-Man loses and 532 if it wins (you can use these numbers to validate your implementation). Why does Pac-Man's behavior in expectimax differ from the minimax case (i.e., why doesn't he head directly for the ghosts)? Again, just think about it; no need to write it up.

Problem 4: Extra Credit: Evaluation Function

- a.  **(15 points)** Write a better evaluation function for Pac-Man 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 uniform cost search from the last assignment. With depth 2 search, your evaluation function should clear the `smallClassic` layout with two random ghosts more than half the time for full credit and still run at a reasonable rate.

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

Document your evaluation function! We're very curious about what great ideas you have, so don't be shy. Extra credit would be determined based on the average score in the 10 runs (Pac-Man should win at least half the time to get considered for the average score calculation). The three people that obtain the highest scores in the class will receive full extra credit irrespective of the score, although almost assuredly most people will attain some form of extra credit through this problem. :)

Hints and Observations

- 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.

Go Pac-Man!

Files:

<code>submission.py</code>	Where all of your multi-agent search agents will reside and the only file you need to concern yourself with for this assignment.
<code>pacman.py</code>	The main file that runs Pac-Man games. This file also describes a Pac-Man <code>GameState</code> type, which you will use extensively in this project
<code>game.py</code>	The logic behind how the Pac-Man world works. This file describes several supporting types like <code>AgentState</code> , <code>Agent</code> , <code>Direction</code> , and <code>Grid</code> .
<code>util.py</code>	Useful data structures for implementing search algorithms.
<code>graphicsDisplay.py</code>	Graphics for Pac-Man
<code>graphicsUtils.py</code>	Support for Pac-Man graphics
<code>textDisplay.py</code>	ASCII graphics for Pac-Man
<code>ghostAgents.py</code>	Agents to control ghosts
<code>keyboardAgents.py</code>	Keyboard interfaces to control Pac-Man
<code>layout.py</code>	Code for reading layout files and storing their contents

