XCS221 Assignment 4 — Pacman

Due Sunday, April 21 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs221-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Also note that your answers should be in order and clearly and correctly labeled to receive credit. Be sure to submit your final answers as a PDF and **tag all pages correctly when submitting to Gradescope**.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be found at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.
- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

This error caused the test to fail.

In this case, start your debugging in line 23 of grader.py.

Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

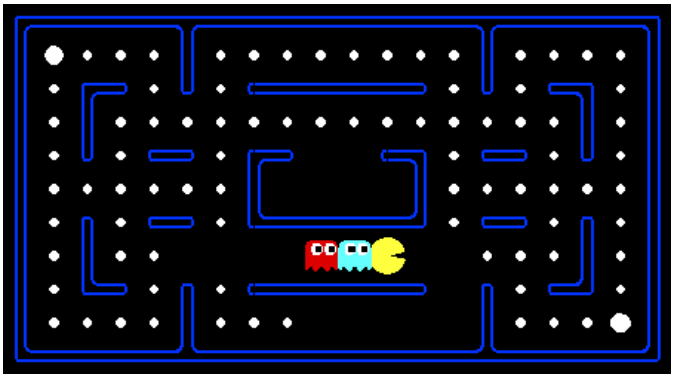
This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)



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 (instructions provided later).

In this assignment, 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 assignment 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 present only to guide the more adventurous amongst you to the heart of Pac-Man. As in previous assignments, you will only be modifying `submission.py`.

A basic `grader.py` has been included, but it only checks for timing issues, not functionality. You can check that Pac-Man behaves as described in the problems, and run `grader.py` without timeout to test your implementation. However, the best way to ensure that your implementation does not time out is to submit test submissions early to Gradescope.

Warmup

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

```
(XCS221)$ 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`:

```
(XCS221)$ python pacman.py -p ReflexAgent
```

Note that it plays quite poorly even on simple layouts:

```
(XCS221)$ 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.

```
(XCS221)$ python pacman.py -p ReflexAgent -k 1
(XCS221)$ python pacman.py -p ReflexAgent -k 2
```

Note: You can never have more ghosts than the layout permits (see `src/layouts/mediumClassic.lay`).

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.

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` object specifies the full game state, including the food, capsules, agent configurations, and score changes: see `submission.py` for further information and helper methods, 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.

Note: If you wish to run the game in the terminal using a text-based interface, check out the `terminal` directory.

Coding Files

`submission.py`: Where all of your multi-agent search agents will reside, and the only file that you need to concern yourself with for this assignment.

`pacman.py`: The main file that runs Pac-Man games. This file also describes a Pac-Man `GameState` type, which you will use extensively in this assignment.

`game.py`: The logic behind how the Pac-Man world works. This file describes several supporting types like `AgentState`, `Agent`, `Direction`, and `Grid`.

`util.py`: Useful data structures for implementing search algorithms.

`graphicsDisplay.py`: Graphics for Pac-Man.

`graphicsUtils.py`: Support for Pac-Man graphics.

`textDisplay.py`: ASCII graphics for Pac-Man.

`ghostAgents.py`: Agents to control ghosts.

`keyboardAgents.py`: Keyboard interfaces to control Pac-Man.

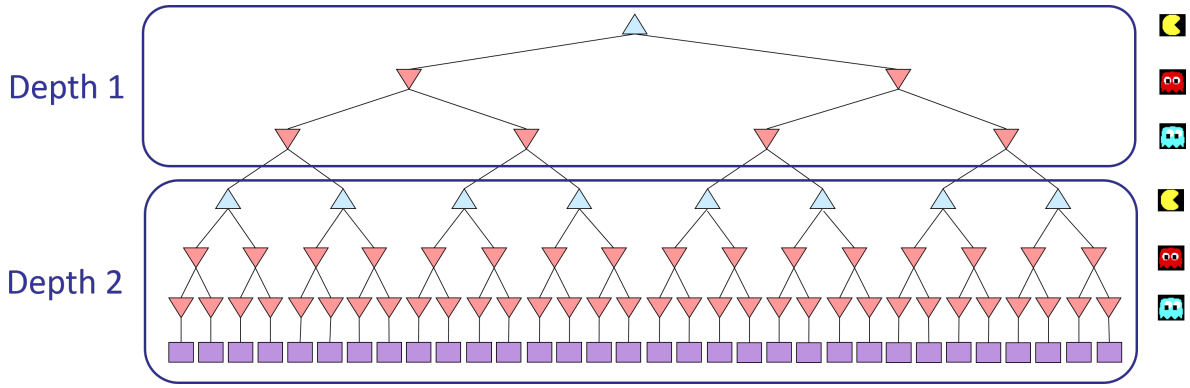
`layout.py`: Code for reading layout files and storing their contents.

`search.py`, `searchAgents.py`, `multiAgentsSolution.py`: These files are not relevant to this assignment and you do not need to modify them.

1. **Minimax**

(a) **[4 points (Written)]** 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.*

Formally, consider the limited depth tree minimax search with evaluation functions taught in class. Suppose there are $n+1$ agents on the board, $a_0, \ldots, a_n$, 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 (see diagram below).



Comment: In reality, all the agents move simultaneously. In our formulation, actions at the same depth happen at the same time in the real game. To simplify things, we process Pac-Man and ghosts sequentially. You should just make sure you process all of the ghosts before decrementing the depth.

Write the recurrence for $V_{\text{minimax}}(s, d)$ in math as a *piecewise function*. 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 s.
- Eval(s), an evaluation function for the state s.
- Player(s), which returns the player whose turn it is in state s.
- Actions(s), which returns the possible actions that can be taken from state s.
- Succ(s, a), which returns the successor state resulting from taking an action a at a certain state s.

You may use any relevant notation introduced in lecture.

(b) **[10 points (Coding)]**

Now fill out the `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.

Implementation Hints

- **Read the comments in submission.py thoroughly before starting to code!**
- Pac-Man is always agent 0, and the agents move in order of increasing agent index. Use `self.index` in your minimax implementation to refer to the Pac-Man's index. Notice that only Pac-Man will actually be running your `MinimaxAgent`.

- All states in minimax should be `GameState`s, either passed in to `getAction` or generated via `GameState.generateSuccessor`. In this assignment, you will not be abstracting to simplified states.
- You might find the functions described in the comments to the `ReflexAgent` and `MinimaxAgent` useful.
- The evaluation function for this part is already written (`self.evaluationFunction`), and you should call this function without changing it. Use `self.evaluationFunction` in your definition of V_{minmax} wherever you used Eval(s) in part 1a. 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.
- If there is a tie between multiple actions for the best move, you may break the tie however you see fit.
- 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 whether your implementation is correct.** Note that your minimax agent will often win, despite the dire prediction of depth 4 minimax search, whose command is shown below. Our agent wins 50-70% of the time: Be sure to test on a large number of games using the `-n` and `-q` flags.

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

Further Observations

These questions and observations are here for you to ponder upon; no need to include in the write-up.

- 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`?

2. **Alpha-beta Pruning**

(a) **[17 points (Coding)]** 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. Running the command given above this paragraph, which uses the default `mediumClassic` layout, the minimax values of the initial state should be 9, 18, 27, and 36 for depths 1, 2, 3, and 4, respectively.

3. **Expectimax**

(a) **[5 points (Written)]** Random ghosts are of course not optimal minimax agents, so modeling them with minimax search is not optimal. Instead, write down the recurrence for $V_{\text{exptmax}}(s, d)$, 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, which means that you should write it in terms of the same functions that were specified in problem 1a.

(b) **[19 points (Coding)]** 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 multiple `RandomGhost`, which each chooses `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 average, and for this particular command, the final score would be -502 if Pac-Man loses and 532 or 531 (depending on your tiebreaking method and the particular trial) if it wins. **You can use these numbers to validate your implementation.**

Why does Pac-Man's behavior as an expectimax agent differ from his behavior as a minimax agent (i.e., why doesn't he head directly for the ghosts)? Again, just think about it; no need to write it up.

4. **Evaluation Function**

(a) **[8 points (Coding, Extra Credit)]** Write a better evaluation function for Pac-Man in the provided function `betterEvaluationFunction`. The evaluation function should evaluate states rather than actions. You may use any tools at your disposal for evaluation, including any `util.py` code from the previous assignments. 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 20
```

We will run your Pac-Man agent 20 times, and calculate the average score you obtained in the winning games. Starting from 1300, you obtain 1 point per 100 point increase in your average winning score. In `grader.py`, you can see the how extra credit is awarded. For example, you get 2 points if your average winning score is between 1500 and 1600. In addition, the top 3 people in the class will get additional points.

Check out the current top scorers on the leaderboard! You will be added automatically when you submit. You can also access the leaderboard by opening your submission on Gradescope and clicking "Leaderboard" on the top right corner.

Hints and Observations

- Having gone through the rest of the assignment, you should play Pac-Man again yourself and think about what kinds of features you want to add to the evaluation function. How can you add multiple features to your evaluation function?
- You may want to use the reciprocal of important values rather than the values themselves for your features.
- The `betterEvaluationFunction` should run in the same time limit as the other problems.

Go Pac-Man Go!

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

1.a

3.a