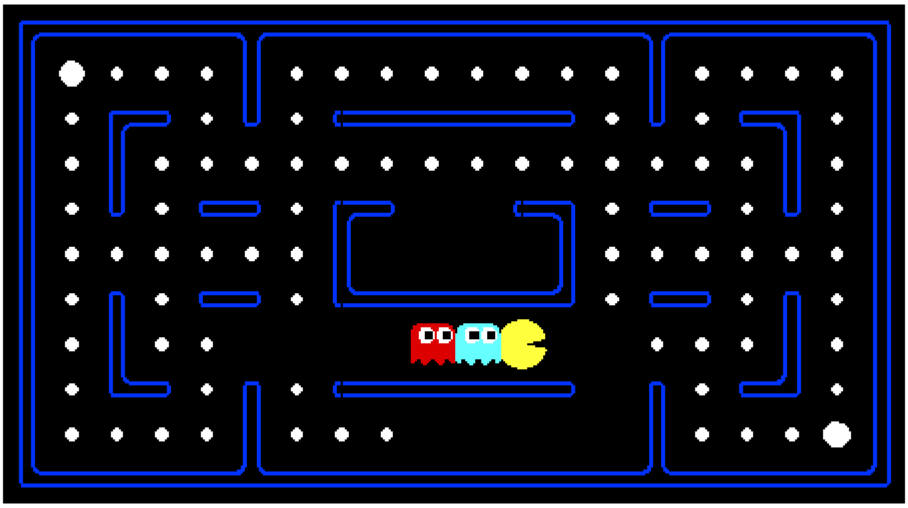
Spring 2021
Introduction to Artificial Intelligence

Homework 3: Multi-Agent Search

Due Date: 2021/5/7 23:55



Introduction

For those of you not familiar with Pac-Man, it's a game that pacman (the yellow circle with a mouth) 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). If pacman eats all the food in a maze, it wins. The big white dots are capsules, which give pacman power to eat ghosts in a limited time.

In this assignment, you will design agents for the classic version of Pac-Man. You will implement both minimax search, alpha-beta pruning and expectimax search.

Welcome to Multi-Agent Pac-Man

The code base of Pac-Man was developed at UC Berkeley. (https://inst.eecs.berkeley.edu/~cs188/sp21/project2/)

You can only run the code base on a local machine. Google Colab cannot execute it because it has GUI. Please install python 3 on your own machine and be familiar with the code with CLI.

First, play a game of classic Pac-Man by running the following command:

```
python pacman.py
```

and using the arrow keys to move.
Next, run the given 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
```

Other Options:

1. Default ghosts are random. You can also play for fun with slightly smarter directional ghosts using -g DirectionalGhost.
2. You can play multiple games in one command with -n.
3. You can turn off graphics with -q to run games quickly.
4. Use -h to know more options.

The code base contains the following files:

Files you will edit:	
multiAgents.py	All of your multi-agent search agents will be resided.
Files you might want to look at:	
pacman.py	The main file that runs Pac-Man games. This file also describes a pacman 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. You don't need to use these for this assignment, but may find other functions defined here to be useful.
Other files you might want to look at, if you are interested in the details of this game.	

Autograding

In this assignment, TAs will use an autograder to grade your implementation. The autograder has been included in the code base. You can use the following command to test by yourself.

```
python autograder.py
```

The autograder will check your code to determine whether it explores the correct number of game states. This will show what your implementation does on some simulated trees and Pac-Man games. After that, it will show the score you get.

Using the autograder to debug is recommended and will help you to find bugs quickly. To test and debug your code for one particular part, run the following command:

```
python autograder.py -q part1
```

To run it without graphics, use the following command:

```
python autograder.py -q part1 --no-graphics
```

Please do not change the names of any provided functions or classes within the code for technical correctness, or you will wreak havoc on the autograder.

Requirements

Please modify the codes in multiAgents.py between # Begin your code and # End your code. In addition, do not import other packages.

All agents you will implement should work with any number of ghosts. In particular, your search tree will have multiple min/chance layers (one for each ghost) for every max layer.

Your code should also expand the game tree to arbitrary depth with the supplied self.depth. A single level of the search is considered to be one pacman move and all the ghosts' responses, so depth 2 search will involve pacman and each ghost moving twice.

Your code should score the leaves of your search tree with the provided self.evaluationFunction, which defaults to scoreEvaluationFunction.

All agents you will implement extend MultiAgentSearchAgent, which gives access to self.depth and self.evaluationFunction. Those two variables are populated in response to command line options.

Part 1: Minimax Search (25%)

- Write an adversarial search agent in the provided MinimaxAgent class stub in multiAgents.py.
- The actual ghosts operating in the environment may act partially randomly, but the minimax algorithm assumes the worst.

Observations:

- Make sure you understand why pacman rushes to the closest ghost in this case:

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

- 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, despite the dire prediction of depth 4 minimax, 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
```

Part 2: Alpha-Beta Pruning (30%)

- Implement a new agent that uses alpha-beta pruning to efficiently explore the minimax tree in AlphaBetaAgent class in multiAgents.py.
- Again, your algorithm will be slightly more general than the pseudo-code discussed in the lecture. This part of the homework is to extend the alpha-beta pruning logic appropriately to multiple MIN agents.
- You must not prune on equality in order to match the set of states explored by our autograder. The pseudo-code below represents the algorithm you should implement for this part.

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v > β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v < α return v
        β = min(β, v)
    return v
```

- The autograder will check your code to determine whether it explores the correct number of game states. It is important that you perform alpha-beta pruning without reordering children. In other words, child states should always be processed in the order returned by GameState.getLegalActions. Again, do not call GameState.getNextState more than necessary.

Observations:

- 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 -l smallClassic
```

Part 3: Expectimax Search (30%)

- 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 part, you will implement the ExpectimaxAgent class in multiAgents.py, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices.
- Random ghosts are of course not optimal minimax agents, and so modeling them with minimax search may not be appropriate. Rather than taking the MIN over all ghost actions, expectimax agent will take the expectation according to your agent's model of how the ghosts act. To simplify your code, please assume you will only be running against an adversary that chooses among its legal actions uniformly at random.

Observations:

- To see how the ExpectimaxAgent behaves in Pac-Man, run the following command:

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

  You should 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 cases:

```
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.

Part 4: Evaluation Function (Bonus) (10%)

- Write a better evaluation function for pacman in the provided function betterEvaluationFunction in multiAgents.py. The evaluation function should evaluate only states not including actions.

Observations:

- 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.

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 get 1 point. Any agent not satisfying these criteria will receive 0 points.
    - +1 for winning at least 4 times, +2 for winning at least 7 times, +3 for winning all 10 times
    - +2 for an average score of at least 500, +4 for an average score of at least 1000 (including scores on lost games)
    - +1 for no timeout at least 5 times, +2 for no timeout all 10 times.
- The autograder will be run on the same machine with --no-graphics.

Report (15%)

- **A written report is required.**
- The report should be written in **English**.
- Save the report as a **.pdf** file.
    - font size: 12
- For part 1 ~ 4, please take some screenshots of your code and explain how you implement codes **in detail**.
- Describe problems you meet and how you solve them.

Submission

Please prepare your multiAgents.py and report (.pdf) into STUDENTID_hw3.zip.

e.g. 309123456_hw3.zip

Late Submission Policy

20% off per late day