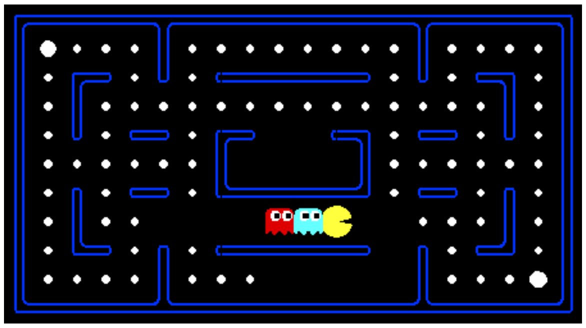
CS221 Spring 2022: Artificial Intelligence: Principles and Techniques

Homework 5: Multi-agent Pac-Man

SUNet ID:	jchan7
Name:	Jason Chan
Collaborators:	None

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.



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 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.

Before you get started, please read the Assignments section on the course website thoroughly.

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.*

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.

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.

What we expect: Write the recurrence for $V_{\text{minmax}}(s,d)$ 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.

Your Solution:

$$V_{minmax}(s, agent, d) = \begin{cases} Utility(s) & \text{IsEnd}(s) \\ Eval(s) & d = 0 \\ \max\limits_{a \in Actions(s)} V_{minmax}(Succ(s,a), nextAgent, d) & player(s) = a_0 \\ \min\limits_{a \in Actions(s)} V_{minmax}(Succ(s,a), nextAgent, d) & player(s) = a_i \text{ where } 1 \leq i \leq n-1 \\ \min\limits_{a \in Actions(s)} V_{minmax}(Succ(s,a), nextAgent, d-1) & player(s) = a_n \end{cases} \tag{1}$$

$$nextAgent = \begin{cases} a_{i+1} & player(s) = a_i \text{ where } 0 \leq i \leq n-1 \\ a_0 & player(s) = a_n \end{cases} \tag{2}$$

b. [10 points] Now fill out the `MinimaxAgent` class in `submission.py` using the above recurrence. Remember that your minimax agent (Pac-Man) 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 be able to expand the game tree to any given 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.

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 ghost 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. To ensure your implementation does not time out, please observe the 0-point test results of your submission on Gradescope.

```
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. Again, you can verify by printing the minimax value of the state passed into `getAction`. Note when comparing the time performance of the `AlphaBetaAgent` to the `MinimaxAgent`, make sure to use the same layouts for both. You can manually set the layout by adding for example `-l minimaxClassic` to the command given above this paragraph.

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 $V_{\text{exptmax}}(s, d)$, which is the maximum expected utility against ghosts that each follow the random policy, which chooses a legal move uniformly at random.

What we expect: 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.

Your Solution:

$$V_{exptmax}(s, agent, d) = \begin{cases} Utility(s) & \text{IsEnd}(s) \\ Eval(s) & d = 0 \\ \max_{a \in Actions(s)} V_{exptmax}(Succ(s,a), nextAgent, d) & player(s) = a_0 \\ \frac{\sum_{a \in Actions(s)} V_{exptmax}(Succ(s,a), nextAgent, d)}{|Actions(s)|} & player(s) = a_i \text{ where } 1 \leq i \leq n-1 \\ \frac{\sum_{a \in Actions(s)} V_{exptmax}(Succ(s,a), nextAgent, d-1)}{|Actions(s)|} & player(s) = a_n \end{cases} \tag{3}$$

$$nextAgent = \begin{cases} a_{i+1} & player(s) = a_i \text{ where } 0 \leq i \leq n-1 \\ a_0 & player(s) = a_n \end{cases} \tag{4}$$

b. [10 points] Fill in `ExpectimaxAgent`, where your Pac-Man agent no longer assumes ghost agents take actions that minimize Pac-Man's utility. Instead, Pac-Man tries to maximize his expected utility and assumes he is playing against multiple `RandomGhost`s, each of which chooses from `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.

Problem 4: Evaluation function (extra credit)

Some notes on problem 4:

- On Gradescope, your programming assignment will be graded out of 30 points total (including basic and hidden tests). However, there is an opportunity to earn up to 10 extra credit points, as described below.
- CAs will not be answering specific questions about extra credit; this part is on your own!

a. [10 points] 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 (extra) credit and still run at a reasonable rate.

```
python pacman.py -l smallClassic -p ExpectimaxAgent -a evalFn=better -q -n
20
```

For this question, we will run your Pac-Man agent 20 times with a time limit of 10 seconds and your implementations of questions 1-3. We will 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 how extra credit is awarded. For example, you get 2 points if your average winning score is between 1400 and 1500. In addition, the top 3 people in the class will get additional points of extra credit: 5 for the winner, 3 for the runner-up, and 1 for third place. Note that late days can only be used for non-leaderboard extra credit. If you want to get extra credit from the leaderboard, please submit before the normal deadline.

b. [2 points] Clearly describe your evaluation function. What is the high-level motivation? Also talk about what else you tried, what worked, and what didn't. Please write your thoughts in `pacman.pdf`, not in code comments.

What we expect: A short paragraph answering the questions above.

Your Solution: The high-level motivation my evaluation function is to encode how I observed myself playing pacman. I played with three modes: eat food, kill the ghosts when I ate the capsule, and finally, run away from the ghosts when they got too close. The limitation with a single evaluation function is that it can't represent these three modes. Hence I implemented a rudimentary state machine to represent my play styles based on the current game conditions.

1. Evade mode
   - Game condition: If a distance threshold is crossed then strongly evade the Ghosts. This distance threshold is defined as the Manhattan distance between Pacman and the nearest Ghost.
   - Objective is to minimise the GhostScore. The ghostScore is negative weight multiplied by the sum of the inverse Manhattan distances of each ghost. We want to avoid all ghosts not just the nearest one.
   - Evaluation function = currentScore - w_gGhostScore
2. Kill mode
   - Game condition: Ghost timer is greater than 0.
   - Objective is to eat the ghosts. I reuse the ghostScore from Evade mode but set a positive weight (closer = higher score).
   - Evaluation function = currentScore + w_g GhostScore
3. Eat mode
   - If not in Evade or Kill mode then default to Eat mode.
   - Objective is to be in the area with the most food. Requires global knowledge of all food location. I value areas with higher density food clusters closest to Pacman. This is represented as a weight multiplied by the inverse of the sum of the squared Manhattan distances between each food to Pacman. While in eat mode we also want to cautiously avoid Ghosts and opportunistically get capsules but these are lower priorities. But if pacman is close to a capsule, prioritise capsule over food! For capsule score, I set that as a weight multiplied by the square of the Manhattan distance between capsule and Pacman.
   - Evaluation function = currentScore + w_fFoodScore - w_gGhostScore + w_cCapsuleScore.

Some earlier experiments that resulted in lower scores.

- No state machine i.e. my evalutaion function was just the 'EAT' state. The average score was very low no matter what combinations of weights I tried.
- Ghost score as the inverse of the distance betweeen the nearest ghost and pacman. This resulted in lower scores. We actually need to be aware of the location of all ghosts not just one.

- Ghost score as the sum of inverse of the distance betweeen the nearest ghost and pacman squared performed worse than sum of inverse distances.
- Capsule score as the inverse of the distance between the nearest capsule and pacman cubed performed worse than squared. Interestingly, capsule distance on its own also performed badly.
- Tuning the weights
  - During Eat mode, be ghost avoidant by turning up ghostScore weight until I saw a reduction in average scores for the winning games.
  - During Evade state boost further decrease the negative weight to strongly incentivise running away. Decrease this negative weight until I saw a reduction in average scores for the winning games. During Kill state strong incentivise the positive weight of ghostScore. I increased this to its maximum until I saw a reduction in average scores for the winning games.

Problem 5: Wicked Problems

Games like Pac-Man are nicely formulated to be solved by AI. They have defined states, a limited set of allowable moves, and most importantly, clearly defined win conditions.

In the Modeling part of our Modeling-Inference-Learning paradigm, we formulate a real world problem as a model, a mathematically precise abstraction in order to facilitate tackling it. Modeling is inherently lossy, but in some domains, it is difficult and controversial even to specify the broad conceptual direction.

Problems that have multiple, potentially conflicting objectives, a high degree of uncertainty and risk, and stakeholder disagreement about what would count as a solution to the problem [1] are sometimes called "wicked problems". Here we contrast a wicked problem with a simple game such as Pac-Man. You can find a more exhaustive list of the characteristics of a wicked problem below [2]:

Characteristics of Simple Games	Example: Pac-Man	Characteristics of Wicked Problems	Example: California Water Management
Clear formulation of the problem	Eat as many pills as possible	No agreement as to formulation of the problem	Fairly allocating water between farmers and cities? Conserving water such that surrounding natural ecosystems still thrive? What combination of these priorities?
Can try multiple times	Can re-play the game until you get it right	Only one shot	Each choice has implications for people at the time it is made, even if another choice is made later
Clear "test" of success	See how many pills you eat	No definitive metric of solution's success	Is it the number of acres of crops watered? ecosystem restored? number of houses that no longer receive water?

a. [5 points] Some classic wicked problems arise in trying to address issues in poverty, homelessness, crime, global climate change, terrorism, healthcare, ecological health, and pandemics [2]. Pick one of these nine wicked problem domains (or another one you clearly define; [3] contains many more wicked problems).

What we expect: Explain in 4-6 sentences why this problem is a wicked problem by explaining why the problem fits into at least two criteria for a wicked problem and explaining why therefore it would be difficult to formulate, model, and solve as an AI problem. It is not enough to just mention the criteria.

Your Solution: Ecological health meets the following criteria for a wicked problem.

- The problem: The solution is not true or false - the end is assessed as 'better' or 'worse' or 'good enough'.
- The role of stakeholders: Many stakeholders are likely to have differing ideas about what the 'real' problem is and what its causes are.
- Nature of the problem: There are not shared values with respect to societal goals.

Firstly, the solution to ecological health is not binary and is best approximated qualitatively. The components that describe ecological health are far too numerous: from the micro (bacteria, microbes to mycelia etc) to the macro (super forest systems, interactions between oceans and weather systems etc). It is intractable to measure and gather data on the impact of any proposed solution on each component of ecological health.

Secondly, stakeholders have diverse opinions on what constitutes the root cause of poor ecological health. We are over-saturated with information about ecological health from campaigners. Some stakeholders represent special interests (specific species of mammals) or local interests (wealthy coastal town centre for example). Social media has amplified the voices of all stakeholders campaigning for ecological health. The net result is that it feels impossible to prioritise what to do, for whom and by when. Making any choice would outrage at least one stakeholder.

Finally, ecological health isn't a shared value with respect to societal goals. For some individuals and organisations ecological health ranks low on their priorities or not at all. Such stakeholders have other objectives they wish to solve and consider the spending of time, effort and money on improving ecological health to be a misallocation of resources.

In conclusion, these three reasons makes ecological health a difficult problem for AI to formulate, model and solve. 1) It's hard to formulate an evaluation metric for ecological health because measuring it is intractable, 2) there are far too many hypotheses to investigate and test for far too many outspoken stakeholder groups, and 3) some stakeholder groups would challenge the allocation of resources toward launching a significant project to improve ecological health because they don't value and prioritise ecological health as a societal outcome as much as others.