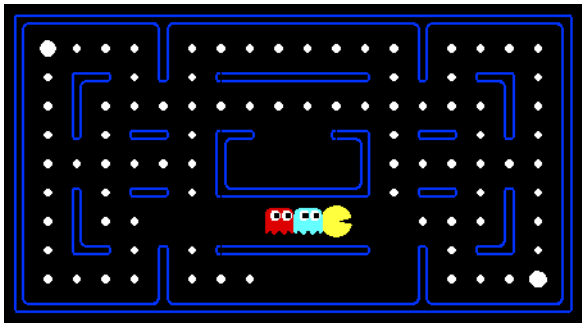
CS221 Summer 2023: Artificial Intelligence: Principles and Techniques

Homework 5: Multi-agent Pac-Man

SUNet ID: alebarro
Name: Alessandro Barro
Collaborators:

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, $a_0, \dots, a_n$, 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.

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. Note that d represents the depth of the minimax search. 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: We can formulate the recurrence for V_{minmax} as a piecewise function in this way

$$V_{\text{minmax}}(s,d) = \begin{cases} \text{Utility}(s) & \text{if IsEnd}(s) \\ \text{Eval}(s) & \text{if } d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a), d) & \text{if Player}(s) = a_0 \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a), d - \frac{1}{n}) & \text{if Player}(s) = a_i \quad \forall\ 0 < i \leq n \end{cases} \tag{1}$$

The above expression returns the whole Utility(s) if the state coincides with the end state. Else, if the depth of the minmax search is zero, then it returns the Eval(s)

function's output. After those checks, the max (Pac-Man) and min (ghosts) values and depths are handled, based on the player Player(s) affecting the current state s. In particular, the depth is decremented after every agent has made its move, hence it is reduced by a $\frac{1}{n}$ value.

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.

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 GameStates, 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.
- Utility(s) should be the final game score, returned from GameState.getScore.
- 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 Vminmax 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.
- 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 (passed into the "-a depth=[depth]" argument). You can use these numbers to verify whether your implementation is correct. To verify, you can print the minimax value of the state passed into getAction and check if the value of the initial state (first value that appears) is

equal to the value listed above. Note that your Pac-Man agent will often win, despite the dire prediction of depth 4 minimax search, whose command is shown below. With depth 4, our Pac-Man agent wins 50-70 percent of the time. Depths 2 and 3 will give a lower win rate. Be sure to test on a large number of games using the -n and -q flags:

- Example: python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
- One ”depth” includes Pac-Man and all of the ghost agents.

Further Observations (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?

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: We can model the Expectimax recurrence as follows

$$V_{\text{expectimax}}(s,d) = \begin{cases} \text{Utility}(s) & \text{if IsEnd}(s) \\ \text{Eval}(s) & \text{if } d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{expectimax}}(\text{Succ}(s,a), d) & \text{if Player}(s) = a_0 \\ \frac{1}{|\text{Actions}(s)|} \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{expectimax}}(\text{Succ}(s,a), d - \frac{1}{n}) & \text{if Player}(s) = a_i \\ & \quad \forall\ 0 < i \leq n \end{cases} \tag{2}$$

The above expression checks whether the current state concides with an end game state, or if the depth is zero. Then, if the agent is Pac-Man, it follows the path which leads to the greatest expected value (instead of the 'less worse' from the minmax model), since the ghosts are now following a stochastic unknown policy (and no more a minimizing, yet known one). Again, depth is decremented after every agent has made its move, hence it is reduced by a $\frac{1}{n}$ value. In addition, the $\frac{1}{|\text{Actions}(s)|}$ term normalizes the ghosts' move choices, so they are uniformly distirbuted.

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)? We'll ask you for your thoughts in Problem 5.

Problem 4: Evaluation function (optional, not graded)

a. [0 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.

b. [0 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:

Problem 5: AI (Mis)Alignment and Reward Hacking

In this problem we'll revisit the differences between our minimax and expectimax agents, and reflect upon the broader consequences of **AI misalignment**: when our agents don't do what we want them to do, or technically do, but cause unintended consequences along the way. Going back to Problem 3, consider the following runs of the minimax and expectimax agents on the small `trappedClassic` environment:

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

Be sure to run each command a few times, as there is some randomness in the environment and the agents' behaviors, and pay attention, as the episode lengths can be quite short. What you should see is that the minimax agent will always rush towards the closest ghost, while the expectimax agent will occasionally be able to pick up all of the pellets and win the episode. (If you don't see this behavior, your implementations could be incorrect!) Then answer the following questions:

a. [2 points] Describe why the behavior of the minimax and expectimax agents differ. In particular, why does the minimax agent, seemingly counterintuitively, always rush the closest ghost, while the expectimax agent (occasionally) doesn't?

What we expect: One sentence why the minimax agent always rushes the closest ghost, and one sentence why the expectimax agent doesn't.

Your Solution: The main difference between the minmax and the expectimax models lies in the nature of their decision-making process. The minmax agent operates in a condition of exact knowledge about the ghosts' policy (which is minimizing its score), thus it rushes to the closest ghost in order to anticipate the worst case scenario (interacting with the min agent, Pac-Max effectively tests the w.c.s.). On the other side, the expectimax agent assumes stochastic policies for his adversaries, therefore, sometimes, instead of testing the ghosts, it predicts their moves by considering the expectancy (average).

b. [1 point] We might say that the Minimax agent suffers from an **alignment** problem: the agent optimizes an objective that we have designed (our state evaluation function), but in some scenarios leads to suboptimal or unintended behavior (e.g. dying instantly). Often the burden is on the designer/programmer to design an objective that more accurately captures the behavior we want from the agent across scenarios. Suggest one potential change to the default state evaluation function (i.e. `scoreEvaluationFunction`) that would prevent the minimax agent from dying instantly in the `trappedClassic` environment, and behave more closely to that of the expectimax agent.

What we expect: 1-2 sentences describing a change in the state evaluation function and why it would work. No need to code anything up, verify that the suggested change is actually accessible in the `GameState` object, or give concrete numbers; just describe the hypothetical change in the evaluation function.

Your Solution: A valid solution would be assigning a very large negative value, or even negative infinity, in the evaluation function in the case where Pac-Man is captured by a ghost. This approach could effectively alter the behavior of the minimax agent. As a matter of fact, it aims to optimize the worst-case outcome: setting such a high penalty for being caught would compel the agent to avoid this scenario at all costs. Therefore, the agent would prioritize its survival and exhibit behavior more similar to the expectimax agent in the trappedClassic environment.

c. [2 points] Pacman's behavior above is an example of one concrete problem in AI alignment called **reward hacking**, which occurs when an agent satisfies some objective but may not actually fulfill the designer's intended goals, due e.g. to an imprecise definition of the objective function. As another example, a cleaning robot rewarded for minimizing the number of messes in a given space could optimize its reward function by hiding the messes under the rug. In this case, the agent finds a shortcut to optimize the reward, but the shortcut fails to attain the designer's goals.

Even if the agent *does* satisfy the designer's goals, another problem can arise: the agent's behavior might cause **negative side effects** that come in conflict with broader values held by society or other stakeholders. For instance, a social media content recommendation system might aim to maximize user engagement, but in doing so, spread disinformation and conspiracy theories (since such posts get the most engagement), which is at odds with societal values.

Can you think of another example of either of these problems?

What we expect: In 2-5 sentences describe another realistic scenario (outside Pacman) in which a designer might specify an objective, but the objective is either susceptible to reward hacking, or the resulting agent/model causes negative side effects. Is your example an instance of reward hacking or negative side effects (or both), and why?

Your Solution: A plausible instance of reward hacking could be exemplified in the context of autonomous vehicles. Imagine a scenario where a self-driving vehicle is programmed to navigate from point A to B in the least possible time. While the expected behavior would be to find the shortest, safe path, the model might exploit unforeseen expedients to meet its goal. For instance, it might resort to uncomfortable and risky strategies such as significantly increasing the vehicle's speed, potentially surpassing legal speed limits, or executing dangerous turns or maneuvers. This behavior deviates from the intended safe and comfortable ride design, addressing the phenomenon of reward hacking where the AI system uncovers and utilizes unintended strategies to optimize (in a poor and undesigned way) the defined reward.