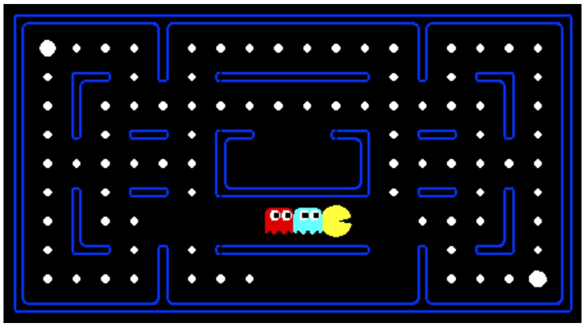
CMPE540: Principles of Artificial Intelligence

Assignment 1: Multi-agent Pac-Man

Student IDs: 20222700075, 2022700252, 2022700144
Names: Onur Porsuk, Damla Şentürk, Anıl Büyükkardeşler

By turning in this assignment, I agree by the CMPE 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 Answer:

The recurrence represents the Minimax algorithm for Pac-Man game with sequential processing of actions. It calculates the value of a state at depth d by considering the maximum or minimum values of its successor states. It depends on whether it's Pac-Man's or a ghost's turn.

The explanation of the elements of the recurrent function is provided as follows:

(a) If the state s is an end state (IsEnd(s)), the utility of that state (Utility(s)) is returned.

(b) If the depth d reaches 0, the evaluation function Eval(s) is used to estimate the state's value.

(c) If it's Pac-Man's turn (Player(s) is Pac-Man), Pac-Man tries to maximize the value by selecting the action a that results in the maximum Vminmax value among possible actions.

(d) If it's a ghost's turn (Player(s) is a ghost), the ghosts try to minimize the value by selecting the action a that results in the minimum Vminmax value among possible actions.

$$V_{\text{minmax}}(s,d) = \begin{cases} \text{Utility}(s) & \text{if IsEnd}(s) \\ \text{Eval}(s) & \text{if } d = 0 \\ \text{max}_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a),d) & \text{if Player}(s) \text{ is Pac-Man} \\ \text{min}_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a),d) & \text{if Player}(s) \text{ is a ghost} \end{cases}$$

Your Solution:

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.

Note: The code is written in the `submission.py` and it is not presented here to avoid complexity in the report.

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 Answer:

The recurrence for $V_{exptmax(s,d)}$ is expressed as a piecewise function that calculates the maximum expected utility against random ghosts who choose legal moves uniformly at random, i.e., the maximum expected utility at a state s and depth d.

- IsEnd(s) checks if the state s is a terminal state.
- Utility(s) returns the utility value of the state s.
- Eval(s) is an evaluation function for the state s.
- Player(s) returns the player whose turn it is in state s.
- Actions(s) returns the possible actions that can be taken from state s.
- Succ(s, a) returns the successor state resulting from taking action a at state s.
- The summation is performed over all ghost agents, and the expected utility is calculated as the average of the maximum utility for each action, where each ghost uniformly selects a random action.

Your Solution:

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 `RandomGhosts`, 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 (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, **for a maximum of 5 points.** 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 Answer:

For a better evaluation function, we calculated the distances of the remaining foods, capsules, scared ghosts, and non-scared ghosts. We want Pacman to be closer to the foods, and further away from the ghosts. If it is necessary to be near a ghost, as the gameplay suggests, we want Pacman to be nearer the scared ghosts than the non-scared ghosts. To do so, our evaluation function assigns higher penalties if the further Pacman's distance is from the food. It also assigns higher penalty points for the same distance to a scared ghost than a non-scared ghost to make Pacman prefer being closer to non-scared ghosts.

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 4, 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 Answer:

Minimax agent rushes to the closest ghost because it evaluates all of the possibilities and it assumes ghosts will play perfectly. On the other hand, expectimax agent add the possibility to the ghost moves so that they are sometimes play imperfectly. Expectimax sometimes avoids the closest ghost if there is a higher probability that a different ghost might move in a more advantageous way.

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 Answer:

We can think about not only the closest ghost but also the distance to the second closest ghost. By adding penalizations to situations where both the closest ghost and the second closest ghost are too close to Pac-Man, the agent would be discouraged from moving directly toward a single ghost when it's also close to another ghost. This change would encourage the agent to avoid situations where it can be easily trapped by multiple ghosts simultaneously.

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 Answer:

We can give an example of reward hacking, like a customer service chatbot. In a chatbot task: the objective is to maximize customer satisfaction by resolving user queries efficiently. But, the bot can exploit reward hacking by responding to every user query with a generic, but positive, response to maximize its satisfaction. In this case, the chatbot finds a short way to optimize the reward without doing the intended goal of resolving user queries effectively.

Additionally, the chatbot's behavior might cause negative side effects. For instance, to quickly resolve queries, the chatbot might give incorrect or misleading information, leading to customer dissatisfaction. While the chatbot achieves the objective of responding very quickly, it comes at the cost of giving misinformation and negatively impacting user experience.