Lab 4

> CS3317: Artificial Intelligence
> Shanghai Jiao Tong University, Spring 2025
> Due Time: Due: 23:59:59 (GMT +08:00), May 8,2025

Assignment

The environment of Ex. 1 is provided in the directory `Lab4/pac` and the environment of Ex. 2 is provided in the directory `Lab4/mdp`.

In this assignment, you should modify the code below

```
"*** YOUR CODE HERE ***"
```

or between

```
# BEGIN_YOUR_CODE
```

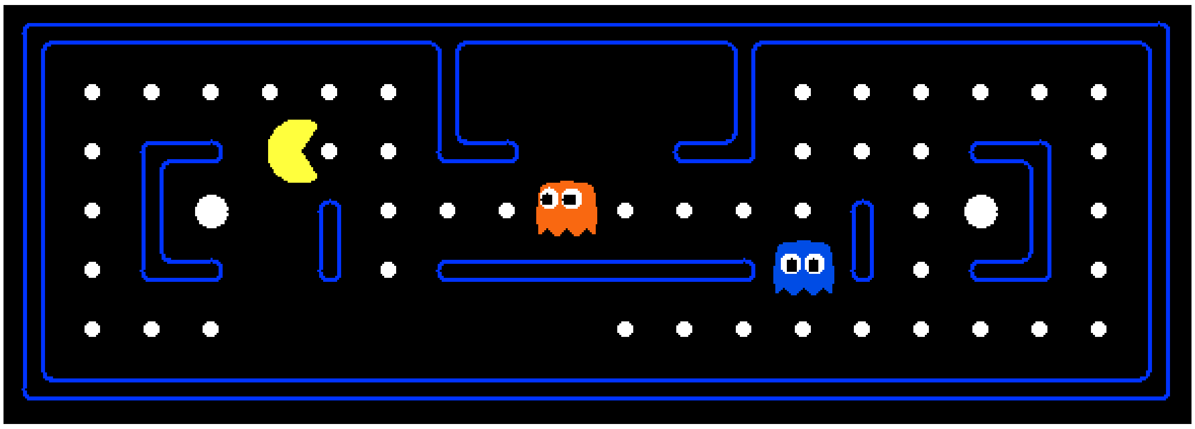
and

```
# END_YOUR_CODE
```

, but you can add other helper functions outside this block if you want.

Exercise 1: Adversarial Search

In Ex. 1, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement both minimax and expectimax search.



Before getting started, you are highly suggested to read through `multiAgents.py`, `ghostAgents.py`, the files that you will edit. You might also want to look at `pacman.py` and `game.py` to be aware of the logic behind how the Pacman world works. After this, try the following commands to learn the meaning of each argument:

```
python pacman.py -h
```

Exercise 1.1 Minimax Search

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in `multiAgents.py`. Your minimax agent should work with **any number of ghosts**, so you'll have to write an algorithm that is slightly more general . In particular, your minimax tree will 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`. `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 in response to command line options.

Important:

- A single search ply is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times.
- Break ties **randomly** for action selection (for all algorithms implemented).

Try your algorithm with:

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

> Hints:
>
> 1. Implement the algorithm recursively using helper function(s).
> 2. The evaluation function for the Pacman test in this part is already written (`self.evaluationFunction`). You shouldn't change this function, but recognize that now we're evaluating *states* rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.
> 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 (665/1000 games for us) despite the dire prediction of depth 4 minimax.
> 4. The provided reflex agent code provides some helpful examples of methods that query the `GameState` for information. Start from here to implement your algorithm.

Exercise 1.2 Alpha-Beta Pruning

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 pseudocode from lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

Important: 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.

Try your algorithm with:

```
python pacman.py -p AlphaBetaAgent -l smallClassic -a depth=3
```

Exercise 1.3 Expectimax Search

In this question you will implement the `ExpectimaxAgent`, which is useful for modeling probabilistic behavior of agents who may make suboptimal choices. To simplify your code, assume you will only be running against an adversary which chooses amongst their `getLegalActions` uniformly at random.

To see how the `ExpectimaxAgent` behaves in Pacman, run:

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

Exercise 1.4 Minimax Ghost

Default ghosts are random, which makes it less challenging. Implement the `MinimaxGhost` in `ghostAgents.py` to create much smarter ghosts.

Important: To simplify, each ghost will run a minimax algorithm with **incomplete** depth $d = 2$. That is, under the same order as that used in `MinimaxAgent`, depth d search for the ghost is considered to be $d - 1$ Pacman moves, $d - 1$ moves for prior ghosts, and d moves for posterior ones (including itself).

Perform some experiments under layout `testClassic` to compare the performance of Pacman against different types of ghost agent, and discuss your findings (with a table similar to that presented in Lecture 5):

- Minimax Pacman (with depth 4) v.s. random ghosts
- Expectimax Pacman (with depth 4) v.s. random ghosts
- Minimax Pacman (with depth 4) v.s. minimax ghosts (with depth 2)
- Expectimax Pacman (with depth 4) v.s. minimax ghosts (with depth 2)

> Hint: Run repeated experiments with `-n` and `-q` option.

Exercise 2: MDP & RL

Exercise 2.1: Value Iteration and Policy Iteration

We have provided a simple example of an MDP called "**Number Line MDP**" in `mdp.py`, where states are integers in $[-n, +n]$ and actions involve moving left and right by one position. We get rewarded for going to the right. We first consider solve this MDP using **Value Iteration** and **Policy Iteration**.

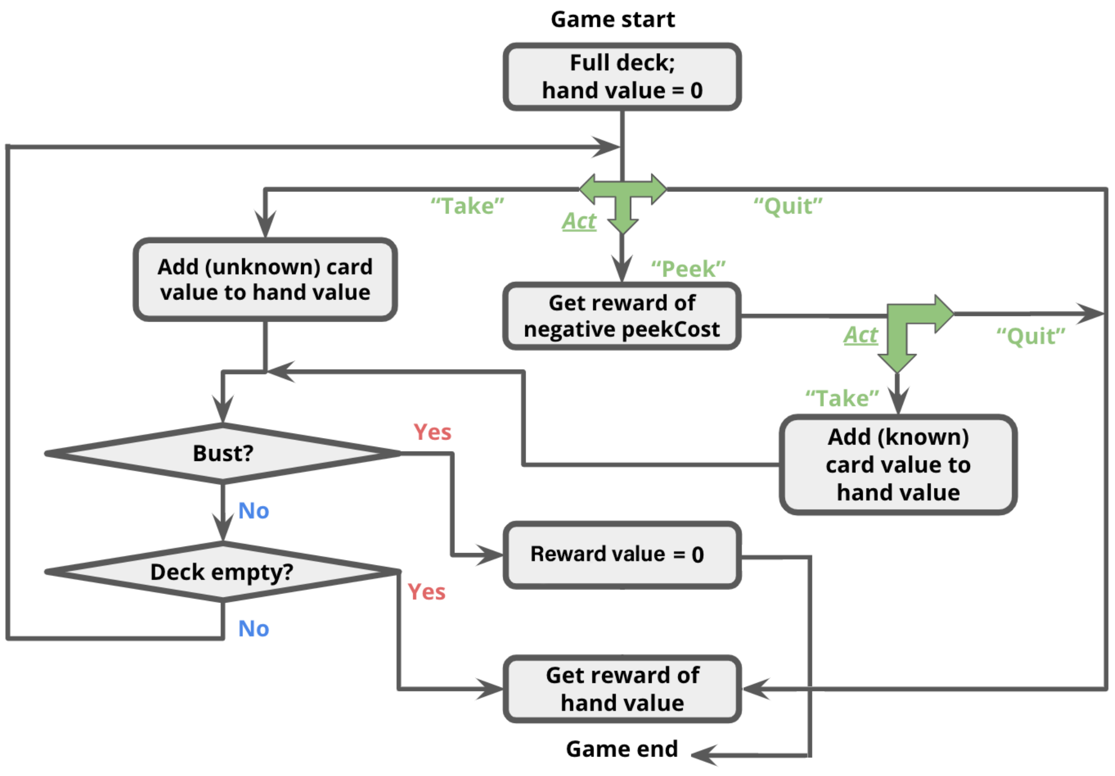
Check `mdp_algorithm.py` . You need to first implement 2 helper functions : `computeQ()` and `computeOptimalPolicy()` in the base class of `MDPAlgorithm`, then finish the function `solve()` in `ValueIteration` and `PolicyIteration` respectively. After doing so, you can run the following commands:

```
python main.py --mdp numberline --algorithm value_iteration --n 5
python main.py --mdp numberline --algorithm policy_iteration --n 5
```

Exercise 2.2: Peeking Blackjack

Now that we have gotten a bit of practice with general-purpose MDP algorithms, let's use them to play (a modified version of) Blackjack. For this problem, you will be creating an MDP to describe states, actions, and rewards in this game. More specifically, after reading through the description of the state representation and actions of our Blackjack game below, you will implement the transition and reward function of the `Blackjack MDP` inside `succAndProbReward()`.

For our version of Blackjack, the deck can contain an arbitrary collection of cards with different face values. At the start of the game, the deck contains the same number of each cards of each face value; we call this number the 'multiplicity'. For example, a standard deck of 52 cards would have face values $[1, 2, \ldots, 13]$ and multiplicity 4. You could also have a deck with face values $[1, 5, 20]$; if we used multiplicity 10 in this case, there would be 30 cards in total (10 each of 1s, 5s, and 20s). The deck is shuffled, meaning that each permutation of the cards is equally likely.


Game start
Full deck; hand value = 0
"Take"
Act
"Quit"
"Peek"
Add (unknown) card value to hand value
Get reward of negative peekCost
Act
"Quit"
"Take"
Add (known) card value to hand value
Bust?
Yes
No
Reward value = 0
Deck empty?
Yes
No
Get reward of hand value
Game end

The game occurs in a sequence of rounds. In each round, the player has three actions available to her:

- `take` - Take the next card from the top of the deck.
- `peek` - Peek at the next card on the top of the deck.
- `quit` - Stop taking any more cards.

In this problem, your state s will be represented as a 3-element tuple:

```
(totalCardValueInHand, nextCardIndexIfPeeked, deckCardCounts)
```

As an example, assume the deck has card values [1, 5] with multiplicity 2, and the threshold is 10. Initially, the player has no cards, so her total is 0; this corresponds to state `(0, None, (2, 2))`.

- For `take`, the two possible successor states (each with equal probability of 1/2) are:

```
(1, None, (1, 2))
(5, None, (2, 1))
```

Two successor states have equal probabilities because each face value had the same amount of cards in the deck. In other words, a random card that is available in the deck is drawn and its corresponding count in the deck is then decremented. Remember that `succAndProbReward()` will expect you return both of the successor states shown above. Note that R(s, take, s') = 0, ∀s,s' ∈ S. Even though the agent now has a card in her hand for which she may receive a reward at the end of the game, the reward is not actually granted until the game ends (see termination conditions below).

- For `peek`, the two possible successor states are:

  ```
  (0, 0, (2, 2))
  (0, 1, (2, 2))
  ```

  Note that it is not possible to peek twice in a row; if the player peeks twice in a row, then `succAndProbReward()` should return `[]`. Additionally, R(s, peek, s') = -peekCost, ∀s,s' ∈ S. That is, the agent will receive an immediate reward of `-peekCost` for reaching any of these states.

  Things to remember about the states after taking `peek`:

  - From `(0, 0, (2, 2, 2))`, taking a card will lead to the state `(1, None, (1, 2, 2))` deterministically (that is, with probability 1.0).
  - The second element of the state tuple is not the face value of the card that will be drawn next, but the index into the deck (the third element of the state tuple) of the card that will be drawn next. In other words, the second element will always be between 0 and `len(deckCardCounts)-1`, inclusive.
- For `quit`, the resulting state will be `(0, None, None)`. (Remember that setting the deck to `None` signifies the end of the game.)

The game continues until one of the following termination conditions becomes true:

- The player chooses `quit`, in which case her reward is the sum of the face values of the cards in her hand.
- The player chooses `take` and "goes bust". This means that the sum of the face values of the cards in her hand is strictly greater than the threshold specified at the start of the game. If this happens, her reward is 0.
- The deck runs out of cards, in which case it is as if she selects `quit`, and she gets a reward which is the sum of the cards in her hand. *Make sure that if you take the last card and go bust, then the reward becomes 0 not the sum of values of cards.*

As another example with our deck of [1, 5] and multiplicity 2, let's say the player's current state is `(6, None, (1, 1))`, and the threshold remains 10.

- For `quit`, the successor state will be `(6, None, None)`.
- For `take`, the successor states are `(7, None, (0, 1))` or `(11, None, None)`. Each has a probability of 1/2 since 2 cards remain in the deck. Note that in the second successor state, the deck is set to `None` to signify the game ended with a bust (since 11 > 10). You should also set the deck to `None` if the deck runs out of cards.

Implement the game of Blackjack as an MDP by filling out the `succAndProbReward()` function of class `BlackjackMDP`.
Note: if you are experiencing TimeOut, it's very likely due to incorrect implementations instead of optimization related issues.

Run the following commands:

```
python main.py --mdp blackjack --algorithm value_iteration --card-values 1 2 3 4
5 6 7 8 9 10 --multiplicity 4 --threshold 21 --peek-cost 1
python main.py --mdp blackjack --algorithm policy_iteration --card-values 1 2 3 4
5 6 7 8 9 10 --multiplicity 4 --threshold 21 --peek-cost 1
```

Submission

There are **only** seven files you need to submit:

- `multiAgents.py`
- `ghostAgents.py`
- `mdp_algorithm.py`
- `mdp.py`
- `report.pdf` for your report;
- `homework.pdf` for your answer to homework problems 1 and 2.

Requirements for your report:

- The report should include an explanation of each component of your algorithm implementation .
- The report should include **all** the results of the commands we provide. You are also encouraged to try other commands and present your findings.
- The report should include some simple comparison and analysis of the algorithms and results.

Name your **zip** file containing the above five as `xxxxxxxxxxxx.zip` with your student ID, and submit it on Canvas.