Lab 4: Adversarial Search & MDP

> CS410: Artificial Intelligence
> Shanghai Jiao Tong University, Fall 2021
> Due Time: 2021.11.21 23:59

Assignment

The environment of Ex. 1 is provided in the directory `Lab4/Pacman`. 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
```

The environment of Ex. 2 and Ex. 3 is provided in the directory `Lab4/MDP`. In Ex. 2 and Ex. 3, you will solve an MDP using value iteration and policy iteration.

Exercise 1: Adversarial Search

1. 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 than what you've previously seen in lecture. 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.

2. 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
   ```

3. 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
   ```

4. 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):

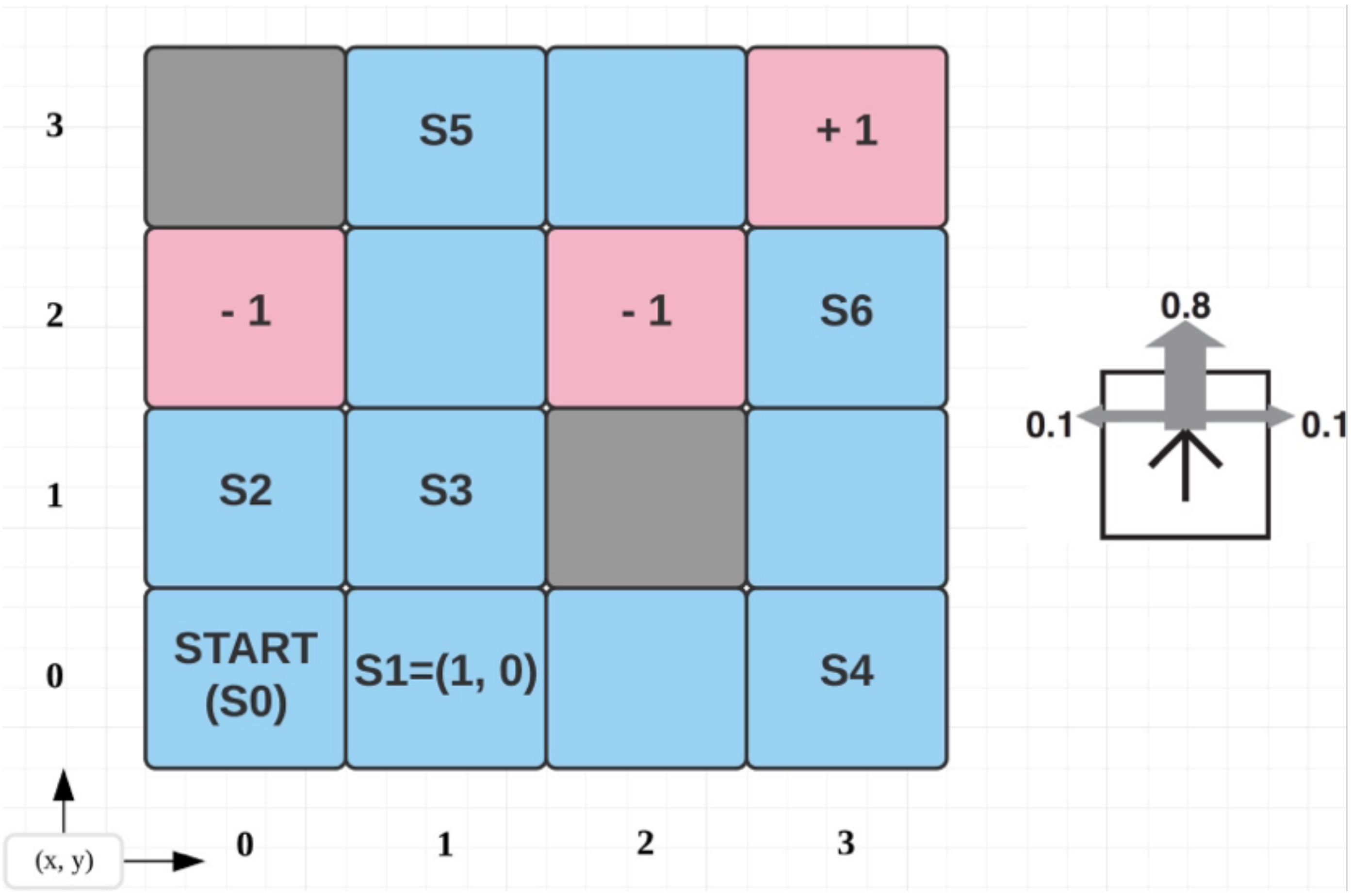
   - Minimax Pacman (with depth 4) v.s. random ghosts
   - Expectimax Pacman (with depth 4) v.s. random ghosts
   - Minimax Pacman (with depth 4) v.s. minimax ghosts (with depth 2)
   - Expectimax Pacman (with depth 4) v.s. minimax ghosts (with depth 2)

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

Exercise 2: Value Iteration

Consider the following two-dimensional grid MDP. In this problem, each grid is treated as a state in MDP. The blue grid with the word 'START' is the initial state of our agent. All the red states are the terminal states which have reward +1 or -1, and all the gray states indicate the wall. Furthermore, each action achieves the intended effect with probability 0.8, but the rest of the time, the action moves the agent at right angles to the intended direction. If the agent bumps into a wall, it stays in the same square. Assume the discount factor $\gamma = 1$.

First implement the function `expected_utility()` in `utils.py` to compute the Q value of a give q-state and then implement function `best_policy()` in `value_iteration.py` to extract the policy by using the function `expected_utility()`. Finally implement the function `value_iteration()` in `value_iteration.py` to compute the utilities (V^* value) of the blue states and find the optimal policy using value iteration (VI). You may refer to Lecture 6, Slide 64, and Figure 17.4 in the reference book (Artificial Intelligence: A Modern Approach) for the implementation details of VI. Consider three cases where all the blue states have a reward of –0.01, -0.4, -2.0 respectively in each case but the rewards of the red states remain the same. Plot the utility estimates of states $S_0 = (0,0)$, $S_1 = (1,0)$, $S_2 = (0,1)$, $S_3 = (1,1)$, $S_4 = (3,0)$, $S_5 = (1,3)$, $S_6 = (3,2)$ against the number of iterations in VI. Discuss your findings on the convergence of the utilities estimated by VI and the policies found by VI in each case.



Try your algorithm with:

```
python main.py --algo=value_iteration --problem=grid_mdp --blue_state_reward=-0.01

python main.py --algo=value_iteration --problem=grid_mdp --blue_state_reward=-0.4

python main.py --algo=value_iteration --problem=grid_mdp --blue_state_reward=-2
```

Exercise 3: Policy Iteration

Consider again the grid MDP problem in Ex. 2. First implement the function `policy_evaluation()` in `policy_iteration.py` using idea 1 in Lecture 6, Slide 75.Then implement the function `policy_improvement()` and the function `policy_iteration()` in `policy_iteration.py`. Use policy iteration (PI) to compute the utilities (V^* value) of the blue states and find the optimal policy. Plot the utility estimates of states $S_0 = (0,0)$, $S_1 = (1,0)$, $S_2 = (0,1)$, $S_3 = (1,1)$, $S_4 = (3,0)$, $S_5 = (1,3)$, $S_6 = (3,2)$ against the number of iterations in PI. Discuss the convergence results of PI as well as the optimal policies found by PI in all three cases where all the blue states have a reward of –0.01, -0.4, -2.0 respectively in each case and compare with that of VI.

Try your algorithm with:

```
python main.py --algo=policy_iteration --problem=grid_mdp --blue_state_reward=-0.01

python main.py --algo=policy_iteration --problem=grid_mdp --blue_state_reward=-0.4

python main.py --algo=policy_iteration --problem=grid_mdp --blue_state_reward=-2
```

Submission

Here are the files you need to submit (please do **NOT** rename any file):

- Directory `Lab4/Pacman` which contains all of your solutions for Ex. 1.
- Directory `Lab4/MDP` which contains all of your solutions for Ex. 2 and Ex. 3.
- `report.pdf` for your **brief** report.

Compress the above three files into one `*.zip` or `*.rar` file and name it with your student ID, and submit it on Canvas.