Lab 4: Adversarial Search & MDP

519021910702 游克垚

1 Exercise 1: Adversarial Search

1.1 MinimaxAgent

My implemention is as follows:

```
class MinimaxAgent(MultiAgentSearchAgent):
    def value(self, gameState, index, depth):
        if index % gameState.getNumAgents() == 0:
            return self.maxValue(gameState, depth + 1)
        else:
            return self.minValue(gameState, index, depth)

    def maxValue(self, gameState, depth):
        if gameState.isLose() or gameState.isWin() or depth >= self.depth:
            return self.evaluationFunction(gameState)

        v = -10000
        for action in gameState.getLegalActions(0):
            next = gameState.generateSuccessor(0, action)
            v = max(v, self.value(next, 1, depth))
        return v

    def minValue(self, gameState, index, depth):
        if gameState.isLose() or gameState.isWin():
            return self.evaluationFunction(gameState)

        v = 10000
        for action in gameState.getLegalActions(index):
            next = gameState.generateSuccessor(index, action)
            v = min(v, self.value(next, index+1, depth))
        return v

    def getAction(self, gameState):
        actions = gameState.getLegalActions(0)

        scores = [self.value(gameState.generateSuccessor(0, action), 1, 0) for action
            in actions]
```

```
32          bestScore = max(scores)
33          bestIndices = [index for index in range(len(scores)) if scores[index] ==
                bestScore]
34          chosenIndex = random.choice(bestIndices)
35
36          return actions[chosenIndex]
```

I use function value(), maxValue(), minValue() to recursively implement the algorithm. If some actions have the same score, I randomly choose one from them.

1.2 AlphaBetaAgent

My implemention is as follows:

```
1   class AlphaBetaAgent(MultiAgentSearchAgent):
2       def value(self, gameState, index, depth, a, b):
3           if index % gameState.getNumAgents() == 0:
4               return self.maxValue(gameState, depth + 1, a, b)[0]
5           else:
6               return self.minValue(gameState, index, depth, a, b)[0]
7
8       def maxValue(self, gameState, depth, a, b):
9           if gameState.isLose() or gameState.isWin() or depth >= self.depth:
10              return [self.evaluationFunction(gameState), None]
11
12          v = -10000
13          ac = gameState.getLegalActions(0)[0]
14          for action in gameState.getLegalActions(0):
15              next = gameState.generateSuccessor(0, action)
16              value = self.value(next, 1, depth, a, b)
17              if value > v:
18                  v = value
19                  ac = action
20              if v >= b:
21                  return [v, ac]
22              a = max(a, v)
23          return [v, ac]
24
25      def minValue(self, gameState, index, depth, a, b):
26          if gameState.isLose() or gameState.isWin():
27              return [self.evaluationFunction(gameState), None]
28
29          v = 10000
30          ac = gameState.getLegalActions(index)[0]
31          for action in gameState.getLegalActions(index):
32              next = gameState.generateSuccessor(index, action)
33              value = self.value(next, index+1, depth, a, b)
34              if value < v:
```

```
                v = value
                ac = action
            if v <= a:
                return [v, ac]
            b = min(b, v)
        return [v, ac]

    def getAction(self, gameState):
        return self.maxValue(gameState, 0, -10000, 10000)[1]
```

Different from MinimaxAgent, the function maxValue() and minValue() return both value and action so that the alpha and beta value can be passed to the root node. But it can't break ties randomly for action selection.

1.3 ExpectimaxAgent

My implemention is as follows:

```
class ExpectimaxAgent(MultiAgentSearchAgent):
    def value(self, gameState, index, depth):
        if index % gameState.getNumAgents() == 0:
            return self.maxValue(gameState, depth + 1)
        else:
            return self.expValue(gameState, index, depth)

    def maxValue(self, gameState, depth):
        if gameState.isLose() or gameState.isWin() or depth >= self.depth:
            return self.evaluationFunction(gameState)

        v = -10000
        for action in gameState.getLegalActions(0):
            next = gameState.generateSuccessor(0, action)
            v = max(v, self.value(next, 1, depth))
        return v

    def expValue(self, gameState, index, depth):
        if gameState.isLose() or gameState.isWin():
            return self.evaluationFunction(gameState)

        v = 0
        p = len(gameState.getLegalActions(index))
        for action in gameState.getLegalActions(index):
            next = gameState.generateSuccessor(index, action)
            v += self.value(next, index+1, depth) / p
        return v

    def getAction(self, gameState):
        actions = gameState.getLegalActions(0)
```

```
31
32          scores = [self.value(gameState.generateSuccessor(0, action), 1, 0) for action
                in actions]
33          bestScore = max(scores)
34          bestIndices = [index for index in range(len(scores)) if scores[index] ==
                bestScore]
35          chosenIndex = random.choice(bestIndices)
36
37          return actions[chosenIndex]
```

It is similar to the MinimaxAgent except that it use expValue() instead of minValue() for ghosts.

1.4 MinimaxGhost

My implemention is as follows:

```
1  class MinimaxGhost(GhostAgent):
2      def __init__(self, index, evalFn = 'scoreEvaluationFunctionGhost', depth = '2'):
3          self.index = index # Ghosts are always with index > 0
4          self.evaluationFunction = util.lookup(evalFn, globals())
5          self.depth = int(depth)
6
7      def value(self, gameState, index, depth):
8          if index % gameState.getNumAgents() == 0:
9              return self.maxValue(gameState, depth + 1)
10         else:
11             return self.minValue(gameState, index, depth)
12
13     def maxValue(self, gameState, depth):
14         if gameState.isLose() or gameState.isWin() or depth >= self.depth:
15             return self.evaluationFunction(gameState)
16
17         v = -10000
18         for action in gameState.getLegalActions(0):
19             next = gameState.generateSuccessor(0, action)
20             v = max(v, self.value(next, 1, depth))
21         return v
22
23     def minValue(self, gameState, index, depth):
24         if gameState.isLose() or gameState.isWin():
25             return self.evaluationFunction(gameState)
26
27         v = 10000
28         for action in gameState.getLegalActions(index):
29             next = gameState.generateSuccessor(index, action)
30             v = min(v, self.value(next, index+1, depth))
31         return v
32
```

```
33      def getAction(self, gameState):
34          actions = gameState.getLegalActions(self.index)
35
36          scores = [self.value(gameState.generateSuccessor(self.index, action), self.
                index+1, 0) for action in actions]
37          bestScore = min(scores)
38          bestIndices = [index for index in range(len(scores)) if scores[index] ==
                bestScore]
39          chosenIndex = random.choice(bestIndices)
40
41          return actions[chosenIndex]
```

It is also similar to MinimaxAgent but the ghost will choose the minimum score. The result of experiments under layout testClassic is shown in Table 1-1.

	Random Ghosts	Minimax Ghosts (with depth 2)
Minimax Pacman (with depth 4)	Won 13/20 Avg.Score: 142.9	Won 10/20 Avg.Score: -7.0
Expectimax Pacman (with depth 4)	Won 15/20 Avg.Score: 243.0	Won 12/20 Avg.Score: 91.05

Table 1-1 the performance of Pacman against different types of ghost agent

So Minimax Ghost generally performs better than Random Ghost and Expectimax Pacman generally performs better than Minimax Pacman.

2 Exercise 2: Value Iteration

2.1 expected_utility()

$$Q^*(s,a) = \sum_{s'}^{a} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

```
1  def expected_utility(a, s, U, mdp):
2      q = 0
3      R, T, gamma = mdp.R, mdp.T, mdp.gamma
4      for p in T(s, a):
5          q += p[0] * (R(p[1]) + gamma * U[p[1]])
6      return q
```

2.2 best_policy()

```
1  def best_policy(mdp, U):
2      policy = {}
3      for s in mdp.states:
4          actions = mdp.actions(s)
```

```
        qs = [expected_utility(a, s, U, mdp) for a in actions]
        bestq = max(qs)
        bestIndex = qs.index(bestq)
        policy[s] = actions[bestIndex]
    return policy
```

2.3 value_iteration()

$V^*(s) = maxQ^*(s, a)$

```
def value_iteration(mdp, epsilon=0.001):
    U1 = {s: 0 for s in mdp.states}
    while True:
        convergent = True
        for s in mdp.states:
            qs = [expected_utility(a, s, U1, mdp) for a in mdp.actions(s)]
            v = max(qs)
            if -epsilon > v - U1[s] or v - U1[s] > epsilon:
                convergent = False
            U1[s] = v
        if convergent:
            break
    return [U1, best_policy(mdp, U1)]
```

2.4 result

The convergence of the utilities under three cases is shown below.

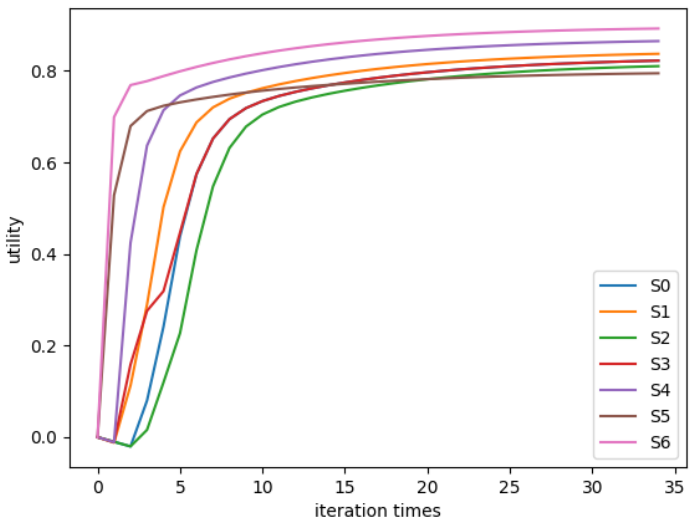

Figure 2-1 VI convergence when blue states have a reward of −0.01

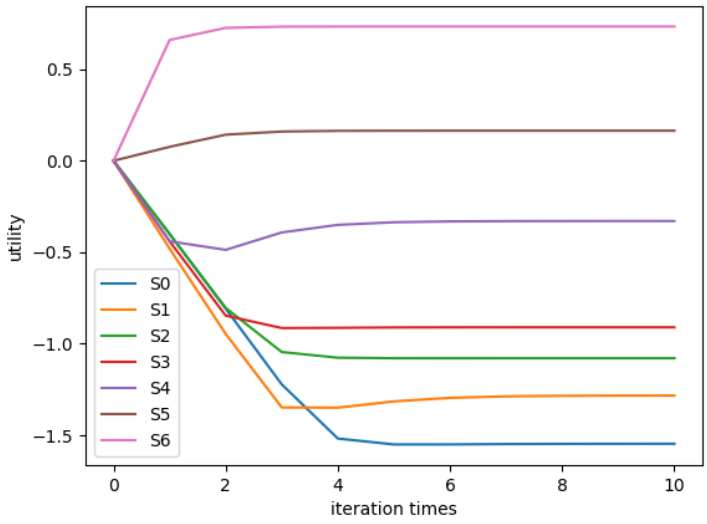

Figure 2-2 VI convergence when blue states have a reward of −0.4

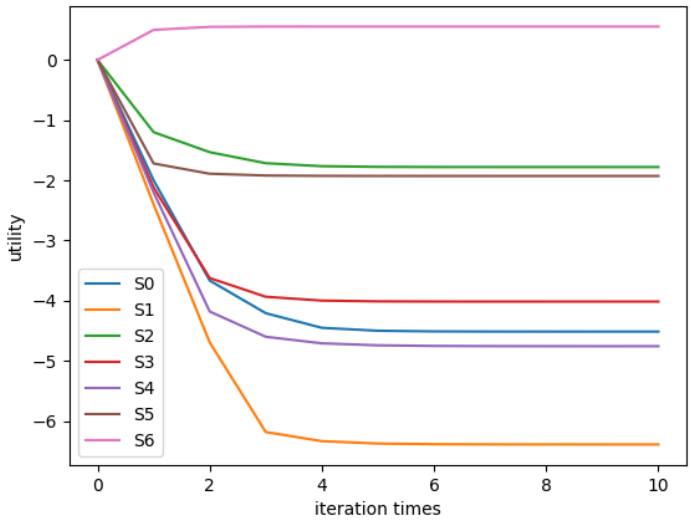

Figure 2-3 VI convergence when blue states have a reward of −2

The policy found by VI when blue states have a reward of −0.01:

```
N > ^ .
. v . >
v v N ^
> > > ^
```

The policy found by VI when blue states have a reward of −0.4:

```
N > > .
. ^ . ^
^ ^ N ^
^ > > ^
```

The policy found by VI when blue states have a reward of −2:

```
N > > .
. > . ^
^ ^ N ^
^ ^ > ^
```

We can find that when blue states have a reward of −0.01, the policy will avoid going to -1 because it's much larger than walking loss. When blue states have a reward of −0.4, it won't consider so much. When blue states have a reward of −2, it will even go to -1 because walking loss is larger than it.

3 Exercise 3: Policy Iteration

3.1 policy_evaluation()

```
1  def policy_evaluation(pi, mdp, iteration_num=50):
2      U = {s: 0 for s in mdp.states}
3      R, T, gamma = mdp.R, mdp.T, mdp.gamma
4      for i in range(iteration_num):
5          for s in mdp.states:
6              q = 0
7              for p in T(s, pi[s]):
8                  q += p[0] * (R(p[1]) + gamma * U[p[1]])
9              U[s] = q
10     return U
```

3.2 policy_improvement()

```
1  def policy_improvement(pi, U, mdp):
2      pi_new = best_policy(mdp, U)
3      convergent = pi_new == pi
```

3.3 policy_iteration

```
1  def policy_iteration(mdp):
2      U = {s: 0 for s in mdp.states}
3      pi = {s: random.choice(mdp.actions(s)) for s in mdp.states}
4      while True:
```

```
        U = policy_evaluation(pi, mdp)
        convergent, pi = policy_improvement(pi, U, mdp)
        if convergent:
            break
    return [U, pi]
```

3.4 result

The convergence of the utilities under three cases is shown below.

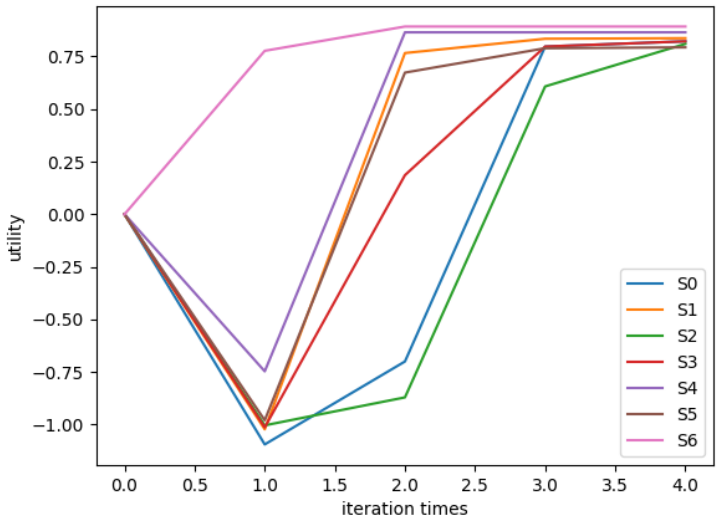

Figure 3-4 PI convergence when blue states have a reward of −0.01

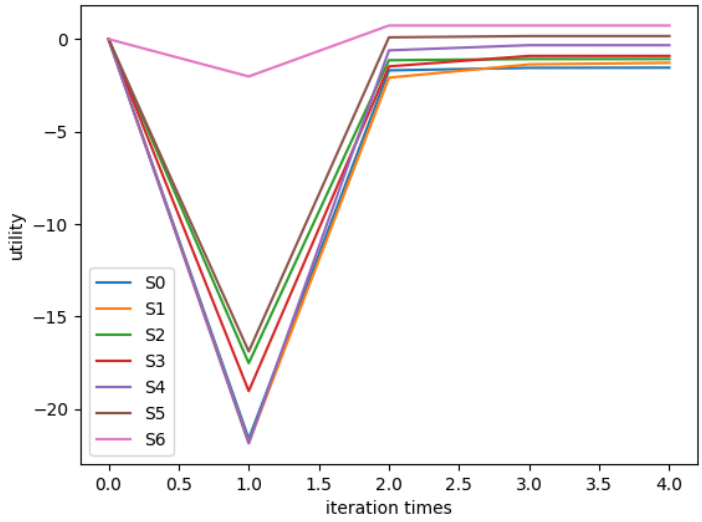

Figure 3-5 PI convergence when blue states have a reward of −0.4

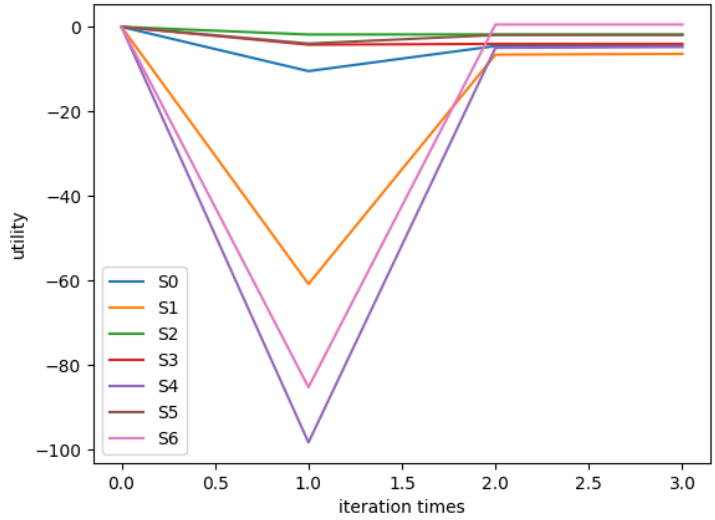

Figure 3-6 PI convergence when blue states have a reward of −2

The policy found by PI when blue states have a reward of −0.01:

```
N > ^ .
. v . >
v v N ^
> > > ^
```

The policy found by PI when blue states have a reward of −0.4:

```
N > > .
. ^ . ^
^ ^ N ^
^ > > ^
```

The policy found by PI when blue states have a reward of −2:

```
N > > .
. > . ^
^ ^ N ^
^ ^ > ^
```

The policy found by PI is the same as that found by VI. PI can converge much faster than VI.