Lab 4

姓名：游灏溢
班级：F1903302
时间：3/11/2021

Exercise1 Adversarial Search

Minimax Agent

In minimax agent, I construct a minimax search tree. Each depth in the search means a max layer and k min layer, which means the pacman moves 1 step and k ghosts moves 1 step perspectively.

Min layer

The function is to determine the action of the $ghost\_id^{th}$ ghost.

```
def minlayer(now_gameState:GameState,layer,ghost_id):
```

And if it is the last ghost, the next layer is the max layer, otherwise is min layer of next ghost.

```
if ghost_id==now_gameState.getNumAgents()-1:
    ...
    new_state=
    now_gameState.generateSuccessor(ghost_id,action)
    value=maxlayer(new_state,layer+1)
else:
    ...
    new_state=
    now_gameState.generateSuccessor(ghost_id,action)
    value=minlayer(new_state,layer,ghost_id+1)
```

Max layer

The function is to determine the action of the pacman.

```
def maxlayer(now_gameState:GameState,layer):
```

If the depth is equal to depth limitation or the game terminates, I return the value now.

```
if layer==self.depth or now_gameState.isWin() or
now_gameState.isLose():
     return self.evaluationFunction(now_gameState)
```

When the pacman is not the first layer, I need to return the best action. And when there are actions with same value, I need to randomly choose(Break tie).

```
if layer==0:
    return random.choice(best_action)
```

Otherwise I only need to return the best value.

```
else:
    return best_value
```

Result

The results are different betlen two situations.

When I break ties, it will win for 40~50 percentages of time

```
PS C:\交大\大三上\ai\lab4\Lab4\Pacman> python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 -q -n 20
Pacman died! Score: -494
Pacman died! Score: -494
Pacman emerges victorious! Score: 516
Pacman died! Score: -492
Pacman died! Score: -495
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman died! Score: -495
Pacman emerges victorious! Score: 516
Pacman died! Score: -497
Pacman died! Score: -495
Pacman died! Score: -494
Pacman emerges victorious! Score: 516
Pacman died! Score: -514
Pacman died! Score: -492
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman died! Score: -504
Average Score: -41.1
Scores:        -494.0, -494.0, 516.0, -492.0, -495.0, 516.0, 516.0, 516.0, 516.0, -495.0, 516.0, -497.0, -495.0, -494.0,
 516.0, -514.0, -492.0, 516.0, 516.0, -504.0
Win Rate:      9/20 (0.45)
Record:        Loss, Loss, Win, Loss, Loss, Win, Win, Win, Win, Loss, Win, Loss, Loss, Loss, Win, Loss, Loss, Win, Win,
Loss
```

But if I do not break ties, the percentage will rise to more than 60 percentages.

```
PS C:\交大\大三上\ai\lab4\Lab4\Pacman> python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 -q -n 20
Pacman emerges victorious! Score: 514
Pacman emerges victorious! Score: 516
Pacman died! Score: -492
Pacman died! Score: -495
Pacman died! Score: -495
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 513
Pacman died! Score: -492
Pacman died! Score: -492
Pacman died! Score: -495
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman died! Score: -492
Pacman emerges victorious! Score: 516
Average Score: 162.5
Scores:        514.0, 516.0, -492.0, -495.0, -495.0, 516.0, 516.0, 516.0, 516.0, 513.0, -492.0, -492.0, -495.0, 516.0, 5
16.0, 516.0, 516.0, 516.0, -492.0, 516.0
Win Rate:      13/20 (0.65)
Record:        Win, Win, Loss, Loss, Loss, Win, Win, Win, Win, Win, Loss, Loss, Loss, Win, Win, Win, Win, Win, Loss, Win
```

When observating the pacman step by step, I discover that usually the first step is `stop`, so the pacman will stay at the same place rather than wander around causing death.

AlphaBeta Pruning

This method is similar to the above method. But we need to add two parameter α, β. α means the lower bound, β means the upper bound. And for the max layer

```
if value>=beta:return value
   alpha=max(alpha,value)
```

For the min layer

```
if value<=alpha:return value
   beta=min(beta,value)
```

Moreover, break ties cannot be applied in the alpha beta pruning because some branch may be pruned.

Expectimax Agent

It's the similar to the minimax agent, but it need to modify the min layer. Here min layer is the average value of all its children rather than the minimum value.

```
for action in actions_space:
    new_state=now_gameState.
generateSuccessor(ghost,action)
    average_value+=maxlayer(new_state,layer)
average_value/=len(actions_space)
```

For expectimax agent, it will win for 70-80 percentage times because the real ghost is the random ghost.

```
PS C:\交大\大三上\ai\lab4\Lab4\Pacman> python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3 -q -n 20
Pacman emerges victorious! Score: 507
Pacman emerges victorious! Score: 513
Pacman emerges victorious! Score: 515
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 507
Pacman emerges victorious! Score: 507
Pacman emerges victorious! Score: 511
Pacman died! Score: -494
Pacman emerges victorious! Score: 515
Pacman emerges victorious! Score: 515
Pacman emerges victorious! Score: 508
Pacman emerges victorious! Score: 511
Pacman emerges victorious! Score: 508
Pacman emerges victorious! Score: 511
Pacman died! Score: -494
Pacman emerges victorious! Score: 512
Pacman died! Score: -495
Pacman died! Score: -508
Pacman emerges victorious! Score: 512
Pacman emerges victorious! Score: 510
Average Score: 308.7
Scores:        507.0, 513.0, 515.0, 503.0, 507.0, 507.0, 511.0, -494.0, 515.0, 515.0, 508.0, 511.0, 508.0, 511.0, -494.0
, 512.0, -495.0, -508.0, 512.0, 510.0
Win Rate:      16/20 (0.80)
Record:        Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Loss, Win, Loss, Loss, Win, Win
```

Minimax Ghost

Design the ghost just like the minimax agent.

I still judge the deepest layer value at the max layer, for pacman. But I determine whether to return the action at the min layer, with the following

```
if ghost==self.index and layer==0:
    return random.choice(worst_action)
```

Here I still try to break ties.

Results

Here I try 4 situations

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

And test 100 times for each situation respectively, the result is below

WinRate, AverageScores	Minimax Pacman	Expectimax Pacman
Random Ghost	0.44, -50.56	0.69,172.64
Minimax Ghost	0.55,12.36	0.40,-78.69

After testing, I surprisingly find that

- When facing random ghost, expectimax strategy has better performance than minimax. While facing minimax ghost, expectimax strategy is better.
- Expectimax pacman v.s random ghost has the highest win rate and scores. Because the random ghost is not as intelligence as minimax, and the expectimax strategy is suitable for it.
- On the contrary, the expectimax pacman v.s minimax ghost has the lowest win rate and scores, for minimax is more clever and expectimax is not suitable for it.
- When discussing why this results, in my opinion, when facing random ghost, the expectation of the ghost's actions is exactly the average, so the evaluation of state value is more accurate. The minimax agent, however, overestimated the ghost's, so the performance is not so better.

Exercise2 Value Iteration

Expected_utility function

This function is used to calculate the $Q_\pi(s,a)$. According to the defination of MDP,

$$Q_\pi(s,a)=\sum_{s'\in\mathbb{S}}T(s'|a)\cdot(R(s,a,s')+\gamma\cdot V(s'))$$

In this grid MDP, there are

$$R(s,a,s')=R(s')$$

So I write the function that

```
destination=mdp.calculate_T(s,a)
value=0
for prob,x in destination:
    value+=prob*(mdp.gamma*U[x]+mdp.R(x))
return value
```

Best_policy function

In each iteration, I need to provide a policy based on the state value. The best policy is defined as

$$\pi(s)=argmax_{a\in\mathbb{A}}Q(s,a)$$

So I code it that

```
    max=99999
    policy=dict()
    for s in mdp.states:
        if s in mdp.terminals:
            policy[s]=None
            continue
        value=-max
        for a in mdp.actions(s):
            now_value=expected_utility(a,s,U,mdp)
            if now_value>value:
                value=now_value
                action=a
        policy[s]=action
    return policy
```

Process of Value Iteration

In each iteration, I do the following steps

1. According to the state value, generate the best policy
2. According to the best policy, calculate the new state value.
3. Compare the distance of old and new value, if it is less than ϵ, terminate the iteration and return the policy and state value

Here the distance I use the $\infty-$norm distance, which means

$d(V_1, V_2) := max_i|V_1[i] - V_2[i]|$

Results

1. move cost=0.01

```
utilities found by value_iteration is
{(0, 1): 0.8086366907266546, (1, 2): 0.44788850445151895, (0, 0): 0.82077
71496259646, (3, 1): 0.8793140132867806, (1, 1): 0.8207771496259646, (2,
0): 0.8511392558114114, (3, 0): 0.8644392294705349, (2, 3): 0.84971647860
6675, (0, 2): 0, (3, 3): 0, (2, 2): 0, (1, 0): 0.835839894203222, (3, 2):
 0.8924372645069849, (1, 3): 0.7934249599683836}
 policy found by value_iteration is
N > ^ .
. v . >
v v N ^
> > > ^
```

2. move cost=0.4

```
utilities found by value_iteration is
{(0, 1): -1.07888320089472, (1, 2): -0.38863042451151997, (0, 0): -1.5464
9431810832, (3, 1): 0.23333329819760007, (1, 1): -0.9098069572300802, (2,
 0): -0.8298442272768001, (3, 0): -0.3293676708544, (2, 3): 0.73333333333
21401, (0, 2): 0, (3, 3): 0, (2, 2): 0, (1, 0): -1.2841926706564801, (3,
2): 0.7333333333323401, (1, 3): 0.16422239405126}
 policy found by value_iteration is
N > > .
. ^ . ^
^ ^ N ^
^ > > ^
```

3. move cost=2

```
utilities found by value_iteration is
{(0, 1): -1.779430736152, (1, 2): -1.794262856435, (0, 0): -4.51392059929
6001, (3, 1): -1.944444683944, (1, 1): -4.014847324776, (2, 0): -7.258008
820224001, (3, 0): -4.757260633984, (2, 3): 0.5555555555330001, (0, 2): 0
, (3, 3): 0, (2, 2): 0, (1, 0): -6.389236752128, (3, 2): 0.55555555554100
01, (1, 3): -1.927758723141}
 policy found by value_iteration is
N > > .
. > . ^
^ ^ N ^
^ ^ > ^
```

Exercise 3 Policy Iteration

Policy Evaluation

Here I use the iteration method to solve matrix equation

$$V = \mathbb{E}_{\pi}[R + \gamma \cdot T \cdot V]$$

Here I use the deterministic policy, so for each state I only need to consider one action

```
U = {s: 0 for s in mdp.states}
for i in range(iteration_num):
    tmp_U=dict()
    for s in U:
    if s in mdp.terminals:
        tmp_U[s]=0
    else:
        tmp_U[s]=expected_utility(pi[s],s,U,mdp)
    U=tmp_U
return U
```

More to mention, the terminal states keep 0.

Policy Improvement

Based on the state, get the best policy.

Then comparing it to the previous judge whether the policy improved.

The process is similar to best policy function, and add

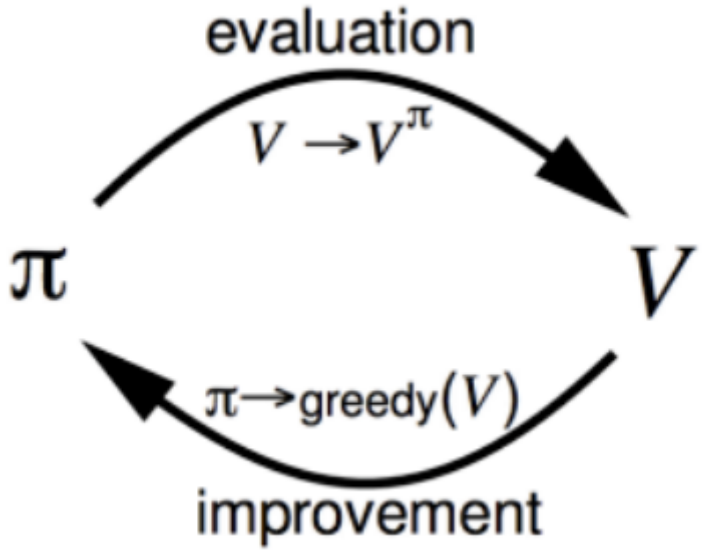
```
return pi==policy,policy
```

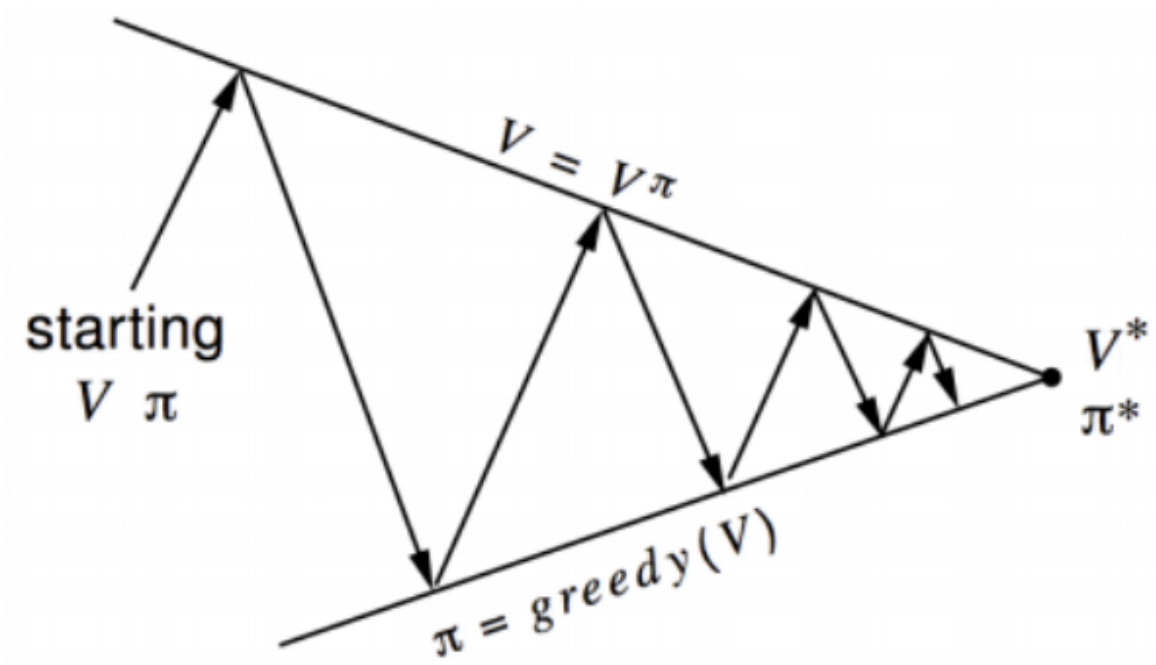
Process of Policy Iteration

In each iteration, I do the following steps

1. According to the policy, get the state value
2. According to the state value, improve the policy
3. Judge whether the policy changes, if not, then terminate.

You can also refer to the following figure





Results

1. move cost=0.01

```
Result:
utilities found by policy_iteration is
{(0, 1): 0.8004788911134249, (1, 2): 0.4414319634188817, (0, 0): 0.813414
1858882511, (3, 1): 0.8749234330835611, (1, 1): 0.8134141858882511, (2, 0
): 0.8455037297248397, (3, 0): 0.85942142352407, (2, 3): 0.84412901878598
03, (0, 2): 0, (3, 3): 0, (2, 2): 0, (1, 0): 0.8293833523077321, (3, 2):
0.8885279436888252, (1, 3): 0.7871297871713124}
 policy found by policy_iteration is
N > ^ .
. v . >
v v N ^
> > > ^
```

2. move cost=0.4

```
Result:
utilities found by policy_iteration is
{(0, 1): -1.0788617886178862, (1, 2): -0.3886178861788617, (0, 0): -1.545
939977918298, (3, 1): 0.2333333333333334, (1, 1): -0.9097560975609756, (2
, 0): -0.8291666666666667, (3, 0): -0.32916666666666666, (2, 3): 0.733333
3333333334, (0, 2): 0, (3, 3): 0, (2, 2): 0, (1, 0): -1.2825654923215901,
 (3, 2): 0.7333333333333334, (1, 3): 0.16422764227642284}
 policy found by policy_iteration is
N > > .
. ^ . ^
^ ^ N ^
^ > > ^
```

3. move cost=2

```
Result:
utilities found by policy_iteration is
{(0, 1): -1.7794259030352977, (1, 2): -1.7942590303529737, (0, 0): -4.513
816648379885, (3, 1): -1.9444444444444446, (1, 1): -4.014833127317677, (2
, 0): -7.256944444444455, (3, 0): -4.756944444444445, (2, 3): 0.55555555
55555556, (0, 2): 0, (3, 3): 0, (2, 2): 0, (1, 0): -6.388942611136575, (3
, 2): 0.5555555555555556, (1, 3): -1.9277571762120589}
 policy found by policy_iteration is
N > > .
. > . ^
^ ^ N ^
^ ^ > ^
```

You can surpringly find that the value iteration and policy iteration provide the same answer, which is the best policy for those MDPs.