Assignment 1

Peng Tang 517020910038

In this assignment, we implement iterative policy evaluation, policy iteration and value iteration on the given small gridworld and compare the performance of policy iteration with value iteration.

1 Analysis

1.1 Policy Iteration

For policy iteration, we use policy evaluation and policy improvement alternatively to find the optimal policy for the given gridworld.

- Policy evaluation: For current policy π, we use the following equation to update the values of all states(except the two terminal states) until the biggest change $\Delta < \theta$, θ is a small number.

  $$V_\pi(s) = \sum_{s'} p(s'|s, \pi(s))[-1 + \gamma V_\pi(s')]$$

  Then, $V_\pi(s)$ will converge to a fixed value following π for all states.

- Policy improvement: After we get the $V_\pi(s)$ under a policy π, we can use the following equation to improve current policy.

  $$\pi'(s) = \arg\max_a \sum_{s'} p(s'|s, a)[-1 + \gamma V_\pi(s')] \quad a \in \{n, e, s, w\}$$

  Then, if there is no any change in actions of all states in this computation, we get the optimal policy.

1.2 Value Iteration

For value iteration, we just update the values for all states by using the following equation until they converge to the final values without considering policy.

$$V(s)_{k+1} = \max_{a \in A}(-1 + \gamma \sum_{s'} p^a_{ss'} V_k(s')) \quad A = \{n, e, s, w\}$$

After getting the final values, we can using the following equation to find the optimal policy.

$$\pi(s) = \arg\max_a \sum_{s'} p(s'|s, a)[-1 + \gamma V(s')] \quad a \in \{n, e, s, w\}$$

2 Code

We implement the policy iteration and value iteration in *solution.py* by defining two class, **PolicyIteration** and **ValueIteration**.

2.1 PolicyIteration Class

```python
class PolicyIteration(object):
    def __init__(self,discount):

        self.BasicAction = [[0,1],[1,0],[0,-1],[-1,0]]  #actions
        self.action_convert = ["right","down","left","up"]
        self.Value = np.zeros([6,6]) #begining values are 0s
        self.error = 0.01
        self.discntFactor = discount
        self.reward = -1
        self.finalState = [[0,1],[5,5]]

        self.changeserror = []
        #random policy
        self.Actions = np.empty([6,6],dtype = object)
        for i in range(6):
            for j in range(6):

                if [i,j]==self.finalState[0] or [i,j]==self.finalState[1]:
                    continue

                self.Actions[i][j] = [0,1,2,3]


    def PolicyEvaluation(self): #policy evaluation

        while True:
            maxerror = 0#the biggest error

            old_values = np.copy(self.Value)
            #traverse
            for i in range(6):
                for j in range(6):

                    if(i==0 and j==1) or (i==5 and j==5): #ingore the terminal states
                        continue

                    action = self.Actions[i][j] #current policy
                    lenth = len(action)
                    self.Value[i][j] = 0
                    for k in range(lenth):
                        ii = max(0,min(i+self.BasicAction[action[k]][0],5)) #compute the next state
                        jj = max(0,min(j+self.BasicAction[action[k]][1],5))
                        self.Value[i][j] += 1./lenth*(self.reward+self.discntFactor*old_values[ii][jj])

                    maxerror = max(maxerror, abs(old_values[i][j]-self.Value[i][j]))
            self.changeserror.append(maxerror)
            if maxerror<self.error:
                break

    def PolicyImprove(self): #policy improvement
        policy_state = True

        for i in range(6):
            for j in range(6):
```

```
                if [i,j]==self.finalState[0] or [i,j]==self.finalState[1]:
                    continue

                old_action = self.Actions[i][j]
                action_val = []
                for a in range(4):
                    ii = max(0,min(i+self.BasicAction[a][0],5))
                    jj = max(0,min(j+self.BasicAction[a][1],5))
                    action_val.append(self.reward+self.discntFactor*self.Value[ii][jj])

                max_action_val = max(action_val)
                current_action = [k for k in range(4) if action_val[k] == max_action_val]
                self.Actions[i][j] = current_action
                if current_action != old_action:
                    policy_state = False

        return policy_state

    #state in [0,35]
    def findPolicy(self, state):

        if state==1 or state==35: #terminal
            return ["stop"]

        i = state/6
        j = state - i*6
        current_state = [i,j]
        action = self.Actions[i][j]
        action_list = []

        while current_state != self.finalState[0] and current_state != self.finalState[0]:
            current_state = current_state + self.BasicAction[action[0]]
            action_list.append(self.action_convert[action[0]])

        return action_list
```

- This class has three functions and several variables to implement the policy iteration.
- **BasicAction** is the actions space for all states; **action_covert** is used as a mapping of **BasicAction**; **Values** is used to record the values of all states; **error** is used as the termination condition for policy evaluation; **discnFactor** is the discount factor for computing the values; **reward** is -1 in this question; **finalState** is to record the terminal states; **Actions** is to record current actions of all states for current policy and the initial policy is the random policy; **changeserror** is used to record the all Δ.
- **PolicyEvaluation()** is used to implement policy evaluation. In this function, we just traverse all states again and again to update the **Values** by using the **Actions** until the max error is smaller than **error**.
- **PolicyImprove()** is used to implement policy improvement. In this function, we traverse all states to update their actions according to new values in **Values**. If there are no changes in **Actions**, we can stop.
- **findPolicy()** is to find the optimal policy for a given state when we already update the **Actions**.

2.2 ValueIteration Class

```
class ValueIteration(object):
    def __init__(self,discount):


        self.BasicAction = [[0,1],[1,0],[0,-1],[-1,0]]
        self.action_convert = ["right","down","left","up"]
        self.Value = np.zeros([6,6])
        self.Actions = np.empty([6,6],dtype = object)
        self.error = 0.01
        self.discntFactor = discount
        self.reward = -1
        self.finalState = [[0,1],[5,5]]

        self.changerrors = []



    def FindValue(self):
        while True:
            maxerror = 0
            old_values = np.copy(self.Value)
            for i in range(6):
                for j in range(6):

                    if [i,j]==self.finalState[0] or [i,j]==self.finalState[1]:
                        continue

                    action_val_max =  -inf
                    for k in range(4):
                        ii = max(0,min(i+self.BasicAction[k][0],5))
                        jj = max(0,min(j+self.BasicAction[k][1],5))
                        action_val_max = max(action_val_max,(self.reward+self.discntFactor*old_values[ii][jj]))

                    maxerror = max(maxerror,abs(action_val_max-self.Value[i][j]))
                    self.Value[i][j] = action_val_max
            self.changerrors.append(maxerror)
            if maxerror<self.error:
                break



        for i in range(6):
            for j in range(6):
                if [i,j]==self.finalState[0] or [i,j]==self.finalState[1]:
                    continue

                action_val = []
                for k in range(4):
                    ii = max(0,min(i+self.BasicAction[k][0],5))
                    jj = max(0,min(j+self.BasicAction[k][1],5))
```

```
                    action_val.append(self.reward+self.discntFactor*self.Value[ii][jj])

                max_action_val = max(action_val)
                current_action = [k for k in range(4) if action_val[k] == max_action_val]
                self.Actions[i][j] = current_action



    def findPolicy(self, state):


        if state==1 or state==35:
            return ["stop"]
        i = state/6
        j = state - i*6
        current_state = [i,j]
        action = self.Actions[i][j]
        action_list = []

        while current_state != self.finalState[0] and current_state != self.finalState[0]:
            current_state = current_state + self.BasicAction[action[0]]
            action_list.append(self.action_convert[action[0]])

        return action_list
```

- This class has two functions and several variables to implement the value iteration.
- The variables in this class are almost the same as in the **PolicyIteration**.
- **FindValue()** is used to find the convergence values for all states for the first and then find the optimal policy by using these values.
- **findPolicy()** is to find the optimal policy for a given state when we already update the **Actions**.

2.3 Main function

```
def main():

    disfactor = 1


    #Policy Iteration
    PolicyMode = PolicyIteration(disfactor)

    flag = False
    t1 = time.time()
    while not flag:
        PolicyMode.PolicyEvaluation()
        flag = PolicyMode.PolicyImprove()
    t2 = time.time()
    PolicyTime = t2-t1
```

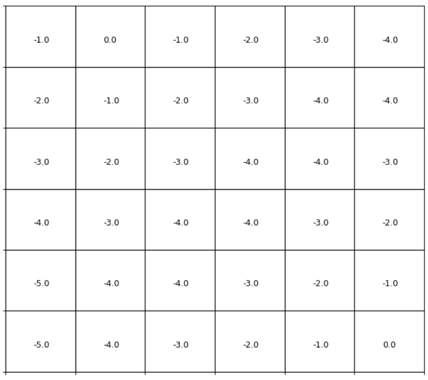
```
    plotResult(PolicyMode.Actions,PolicyMode.Value,"imgs/Policy_Iteration_result.png")



    #Value Iteration

    ValueMode = ValueIteration(disfactor)
    t1 = time.time()
    ValueMode.FindValue()
    t2 = time.time()
    ValueTime = t2-t1


    plotResult(ValueMode.Actions,ValueMode.Value,"imgs/Value_Iteration_result.png")

    #compare performance
    plotPerformanceCompare(PolicyMode.changeserror,ValueMode.changerrors)
    print("The spent time of Policy Iteration: %s" %PolicyTime)
    print("The spent time of Value Iteration: %s" %ValueTime)
```

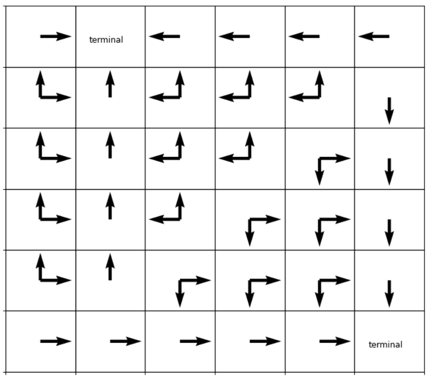
By using two class, we can solve this problem with two methods.

3 Results

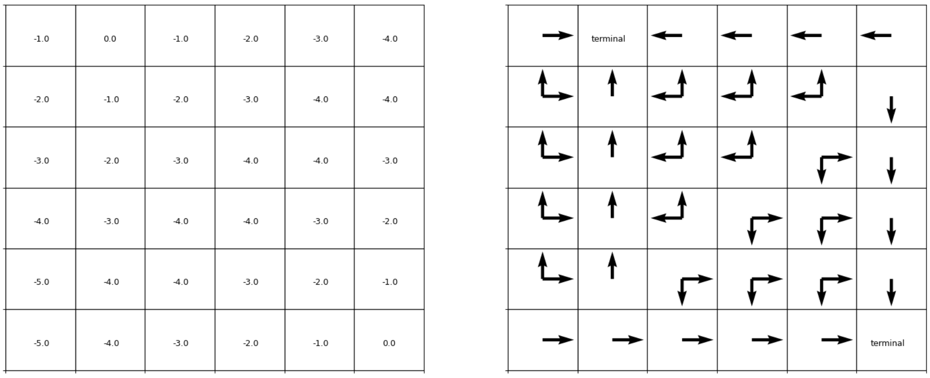
By running the code, we can get following results.

- Policy Iteration



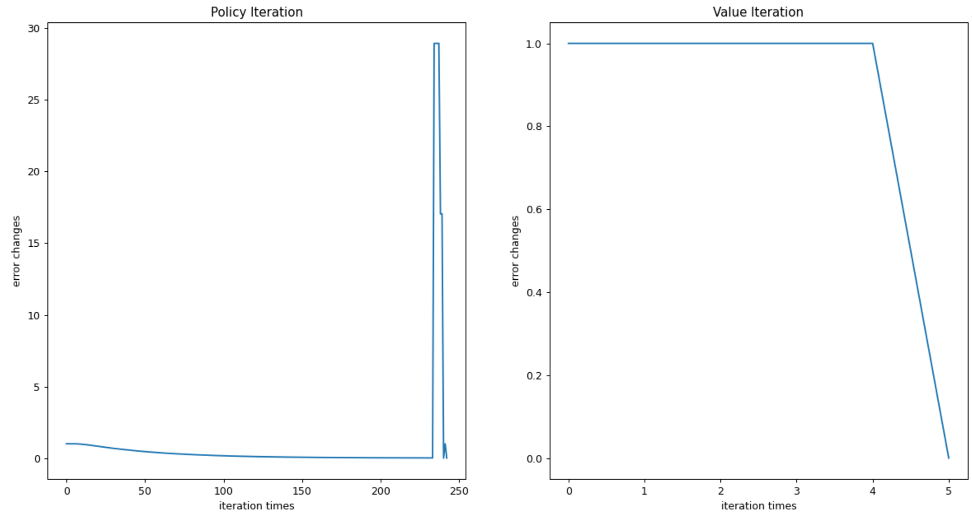


- Value Iteration



From above figures, we can see that we get the correct values and actions for all states.

What's more, we can compare the performance of the two methods.




```
PS E:\大三下\强化学习\作业\homework1> D:\python\python.exe "e:\大三下\强化学习\作业\homework1\solution.py"
The spent time of Policy Iteration: 0.05807638168334961
The spent time of Value Iteration: 0.00799703598022461
```

- From the first figure, we can see that the iteration times of value iteration is much smaller than the policy iteration. And the error change of policy iteration is much bigger than the value iteration.
- From the second figure, we can see that the spent time of policy iteration is almost ten times bigger than value iteraton's.
- From above results, we can conclude that the performance of value iteration is much better than policy iteration in this problem. However, this doesn't means value iteration is always better than policy iteration. This depends on specific problems.