Assignment 1

Name：Shiqu Wu

Student ID：518021910665

1.

(1.1)

Build the GridWorld Environment:

It is easy to build the RL environment for this assignment. Just use `np.zeros()` and `np.ones()` to build the memory for Dynamic Programming. Here are the basic environment settings:

```
gamma = 1
rewardSize = -1
gridSize = 6
terminationStates = [[0, 1], [gridSize - 1, gridSize - 1]]
actions = [[-1, 0], [1, 0], [0, 1], [0, -1]]
init_valueMap = np.zeros((gridSize, gridSize))
states = [[i, j] for i in range(gridSize) for j in range(gridSize)]

init_actionMap = 0.25 * np.ones((np.shape(states)[0],
np.shape(actions)[0]))
```

(1.2)

Policy Evaluation:

According to the lecture slides, policy evaluation update the value map in each iteration until the loop reaches the terminate condition. Here for this assignment's setting, given state s, action a, the transition function is determined that we can exactly know the next state s'. Thus, when calculate the estimated value for state s, we have $p(s', r|s, \pi(s)) = 1$. So we implement the state's value calculation like this:

```
        for si, state in enumerate(states):
            weightedRewards = 0
            for ai, action in enumerate(actions):
                if actionMap[si][ai] > 0:
                    finalPosition, reward =
actionRewardFunction(state, action)
                    weightedRewards += actionMap[si][ai] * (
                            reward + (gamma *
valueMap[finalPosition[0], finalPosition[1]]))
```

Note that the terminate condition in policy evaluation can be set differently and will have significant impact on total calculation time and policy iteration step's result. So in the experiment we use 4 different terminate conditions to see the difference. Here are the experiment setting:

- Run until converged:

  ```
  if np.all(copyValueMap == valueMap):
      value_stable = True
  ```

- Run until the updated difference is smaller than threshold:

  -
    ```
    if np.all(np.abs(copyValueMap - valueMap) < 1e-1):
        value_stable = True
    ```

  -
    ```
    if np.all(np.abs(copyValueMap - valueMap) < 1e-2):
        value_stable = True
    ```

  -
    ```
    if np.all(np.abs(copyValueMap - valueMap) < 1e-6):
        value_stable = True
    ```

And the experiment results will be displayed in part 2.

(1.3)

Policy Iteration:

Policy iteration updates the actionMap (which is set to shape (n_states,n_actions)) where we store the probability for each action at each state. It use greedy update method to choose the max value action in current valueMap as the new action. So I implement it as below:

```
        for i, state in enumerate(states):
            old_action = np.copy(actionMap[i])
            a_list = []
            for action in actions:
                finalPosition, reward = actionRewardFunction(state,
action)
                estimated_reward = reward + (gamma *
valueMap[finalPosition[0], finalPosition[1]])  # transition prob=1
                a_list.append(estimated_reward)
            actionMap[i] = 0
            actionMap[i][np.argmax(a_list)] = 1  # greedy update
```

Then if the policy is still unstable, policy iteration method will call the policy evaluation method to update the valueMap. After the policy evaluation, it takes the updated valueMap and start the next iteration to update the actionMap again. These two methods update alternatively until the policy is stable. Then we terminate the loop and output the final actionMap and valueMap as the result.

2.

(2.1)

Results after termination:

`valueMap:`

[-1. 0. -1. -2. -3. -4.]
[-2. -1. -2. -3. -4. -4.]
[-3. -2. -3. -4. -4. -3.]
[-4. -3. -4. -4. -3. -2.]

[-1. 0. -1. -2. -3. -4.]
[-5. -4. -4. -3. -2. -1.]
[-5. -4. -3. -2. -1. 0.]

`actionMap:`

```
[[0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]]
```

(2.2)

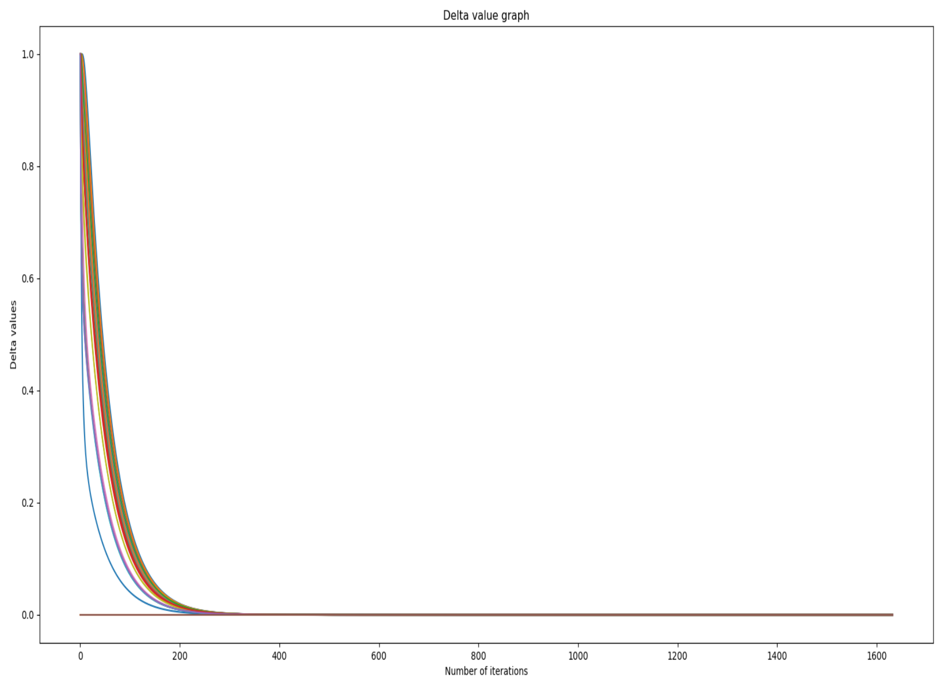
Policy Evaluation terminate condition experiment:

- **Run until converged:**

  Total number of policy update iterations: `3`

  Total number of policy evaluation iterations: `1633+7+2`

  Policy Evaluation process for initial policy:



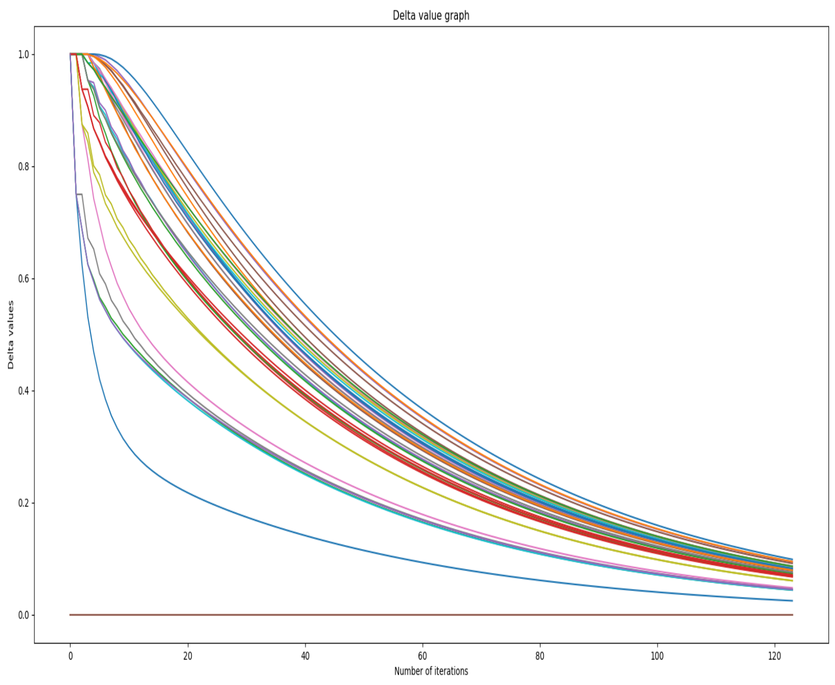
- **Run until the updated difference is smaller than threshold:**

  - threshold=0.1:

    Total number of policy update iterations: `3`

    Total number of policy evaluation iterations: `124+7+2`

    Policy Evaluation process for initial policy:

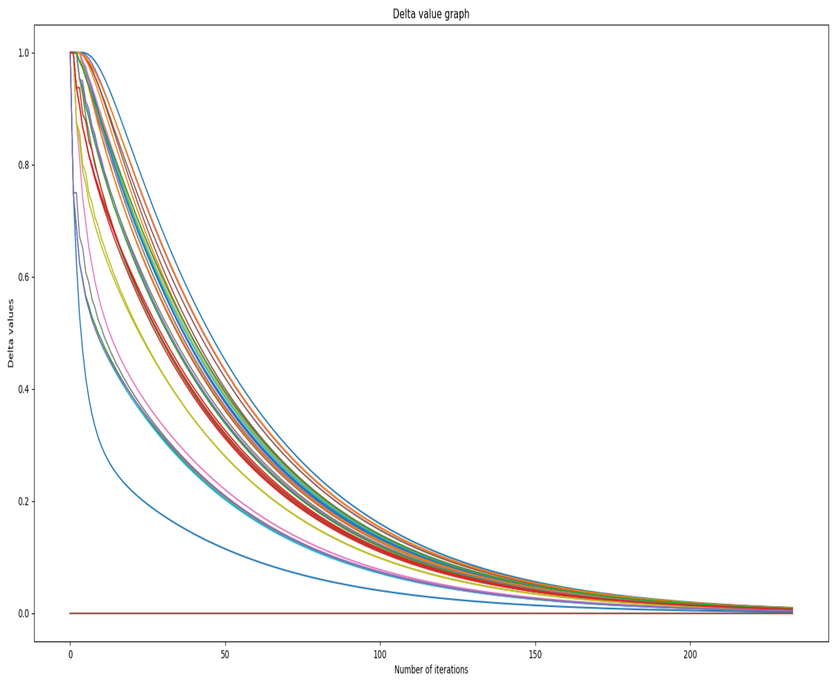


- threshold=0.01:

  Total number of policy update iterations: `3`

  Total number of policy evaluation iterations: `234+7+2`

  Policy Evaluation process for initial policy:

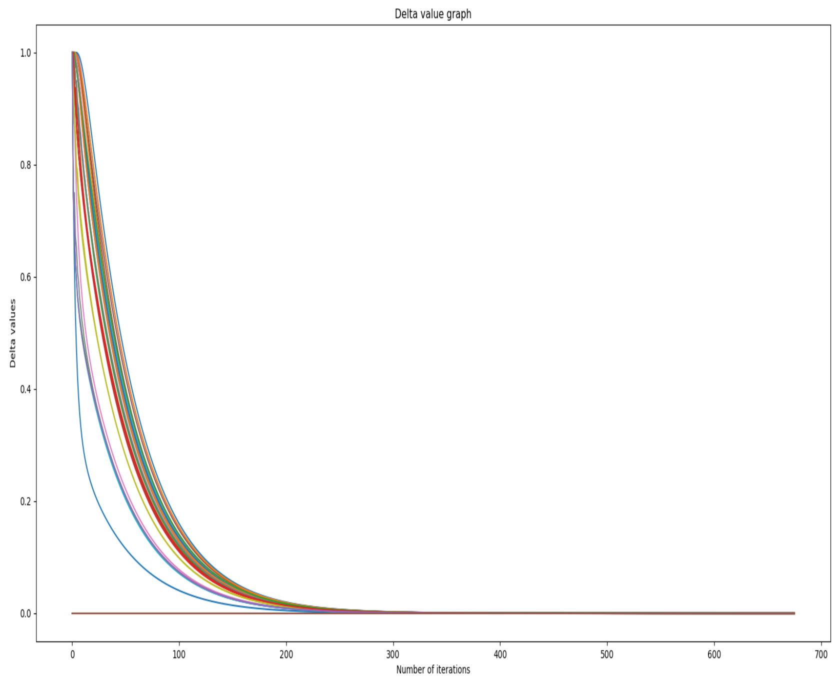


- threshold=1e-6:

  Total number of policy update iterations: `3`

  Total number of policy evaluation iterations: `676+7+2`

Policy Evaluation process for initial policy:



From the results above we can see that all four cases need `3` policy update iterations, and they all reach the same results after termination. That is to say, all four settings achieve the optimal solution. So the calculation cost and running time become the key factor when we evaluate these terminate conditions. We can see that when we set `threshold=0.1` it take the smallest number of policy evaluation iterations (only `124+7+2=133` total iterations). From the $delta - iteration$ graph we can see that the average rate of delta decline remained at a high level, and finally reached a lower delta value. So we choose `threshold=0.1` as the terminate condition for Policy Evaluation Methods.

Note: More details about result please see the `result.txt`.

3.

Mistakes made in the experiment:

- Use direct assignment to set the `old_action` variable.

  Need to use `np.copy()` here, otherwise they are the same. For the fact that python use lazy assignment.

- Do the Policy Iteration first then apply the Policy Evaluation.

  The sequence of policy evaluation and iteration is important that we need to do the `Policy Evaluation` first, then we can apply the `Policy Iteration` step. Otherwise the policy evaluation step will never stop. For the fact that in policy iteration step, it update the initial `actionMap` to all take the first action (go north here). So in policy evaluation step, the `valueMap` will be updated endlessly. That results the algorithm can never be stopped.