Report for Assignment 1

519021910913 黃喆敏

Directory Structure

```
lab1
├── code
│   ├── GridWorld.py
│   ├── PolicyIteration.py
│   └── ValueIteration.py
└── docs
    ├── assets                    # images for report
    │   ├── policy_algorithm.png
    │   ├── policy_iter_1.png
    │   ├── policy_iter_2.png
    │   ├── small_policy_iter_1.png
    │   ├── small_policy_iter_2.png
    │   ├── value_iter_1.png
    │   └── value_iter_2.png
    ├── report.md
    └── report.pdf
```

All codes are placed in `./code` directory. To run the code, you can either run `PolicyIteration.py`, or `ValueIteration.py`, in which I have implemented two methods separately.

GridWorld

Firstly, I have implemented a class `GridWorld` in `GridWorld.py`, which provides some function that can be used in both methods and includes some basic parameters, such as `terminal`, `discount factor`, `reward`, etc. Details can been seen in code annotation.

Besides, we use `matplotlib` library to visualize the results of action. I have adjusted some parameters to improve the visual effect. The code is given below:

```python
def draw_action(self, filename='2.png'):
    fig_pos = [(0.5, 1.0), (0.5, 0.0), (0.0, 0.5), (1.0, 0.5)]
    fig = plt.figure(figsize=(6, 6))
    plt.subplots_adjust(wspace=0, hspace=0)
    for x in range(self.n):
        for y in range(self.m):
            ax = fig.add_subplot(self.n, self.m, x * self.m + y + 1)
            ax.axes.get_xaxis().set_ticks([])     # remove the axis label
            ax.axes.get_yaxis().set_ticks([])
```

```
            ax.spines['bottom'].set_linewidth(2)  # change the linewidth between
subplots
            ax.spines['top'].set_linewidth(2)
            ax.spines['left'].set_linewidth(2)
            ax.spines['right'].set_linewidth(2)

            if self.is_terminal(x, y):
                ax.set_facecolor('grey')          # color the terminal grid
            else:
                pos = np.array([x, y]) + np.array(self.action[int(self.policy[x][y])])
                value = self.value[pos[0], pos[1]]

                for idx in range(len(self.action)):
                    next_pos = np.array([x, y]) + np.array(self.action[idx])
                    if next_pos[0] < 0 or next_pos[0] >= self.n or
                    next_pos[1] < 0 or next_pos[1] >= self.m:
                        continue
                    elif abs(self.value[next_pos[0], next_pos[1]]
                             - value) < 0.0001:   # draw arrow
                        ax.annotate('', fig_pos[idx], xytext=(0.5, 0.5),
                                    xycoords='axes fraction',
                                    arrowprops={'width': 1.5, 'headwidth': 10,
                                                'color': 'black'})
    plt.savefig(filename, dpi=1000)
    plt.show()
```

Policy Iteration

For policy iteration, My solution is based on the following algorithm:

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Iterative Policy Evaluation

For policy evaluation, we need to iteratively update the value matrix, until the marix is converged. We must remember that we need to add all values of possible action, instead of selecting the maximum value. Here is the code.

```
def policy_evaluation(self, theta=0.001):
    is_converge = False
    iter_num = 0
    n = self.gridworld.n
    m = self.gridworld.m
    while not is_converge:
        iter_num += 1
        delta = 0
        tmp_value = np.zeros_like(self.gridworld.value)
        for x in range(n):
            for y in range(m):
              # do not need to calc the value of terminal
                if self.gridworld.is_terminal(x, y):
                    continue
                else:
                # calc the value of current grid
                    tmp_value[x][y] = self.calc_value(x, y)
                    delta = max(delta, abs(tmp_value[x][y] - self.gridworld.value[x]
[y]))
        self.gridworld.value = tmp_value
        if delta < theta:
```

```
            is_converge = True
```

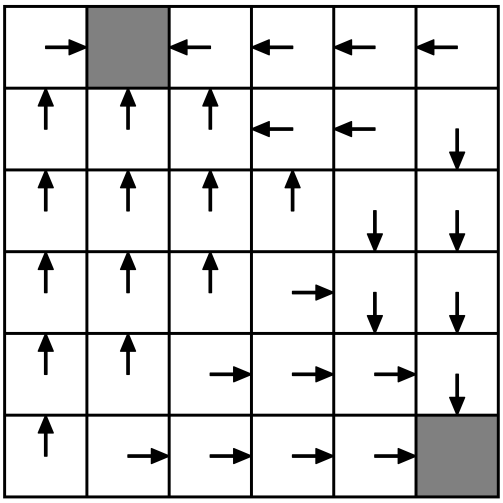
Policy Improvement

For policy improvement, we need to update the policy according the new value matrix. If there exists some policy changed, the policy has not been stable. We need to return to policy evaluation and iterate again.

```
def policy_improvement(self):
    policy_stable = True
    n = self.gridworld.n
    m = self.gridworld.m
    for x in range(n):
        for y in range(m):
            if self.gridworld.is_terminal(x, y):
                continue
            else:
                tmp_idx = 0
                tmp = -float('inf')
                for idx in range(len(self.gridworld.action)):
                    next_pos = np.array([x, y]) + np.array(self.gridworld.action[idx])
                    if next_pos[0] < 0 or next_pos[0] >= n
                    or next_pos[1] < 0 or next_pos[1] >= m:
                        continue
                    elif self.gridworld.value[next_pos[0], next_pos[1]] > tmp:
                        tmp_idx = idx
                        tmp = self.gridworld.value[next_pos[0], next_pos[1]]
                if self.gridworld.policy[x][y] != tmp_idx:
                    policy_stable = False
                    self.gridworld.policy[x][y] = tmp_idx
    return policy_stable
```

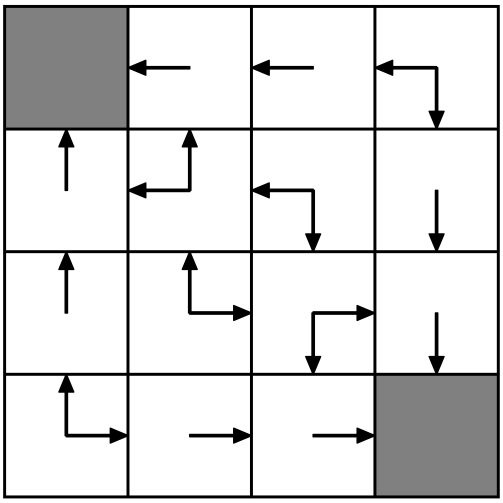
After that, we use the visualization methods in `GridWorld.py`. Here is the result.

-18.16	0.0	-29.2	-44.03	-51.52	-54.64
-32.32	-30.15	-39.57	-47.38	-51.89	-53.76
-44.65	-44.7	-47.55	-50.02	-50.92	-50.75
-52.93	-52.47	-51.91	-50.23	-47.02	-43.58
-57.67	-56.34	-53.4	-47.98	-39.35	-28.98
-59.74	-57.82	-53.38	-44.93	-29.42	0.0



Besides, **we can use 4×4 matrix example as our reference**, in order to prove the correctness of policy iteration. The result is given below:

0.0	-13.99	-19.99	-21.99
-13.99	-17.99	-19.99	-19.99
-19.99	-19.99	-17.99	-13.99
-21.99	-19.99	-13.99	0.0



Both value matrix and optimal policy matrix are the same as those given in ppt. Therefore, the correctness of policy iteration has been verifed.

Value Iteration

The value iteration method is implemented according to the following pseudocode.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

Most codes are similar to policy iteration's codes. But there exists some difference in `calc_value` function. In value iteration, we directly choose the action which has the largest value, rather than adding them all.

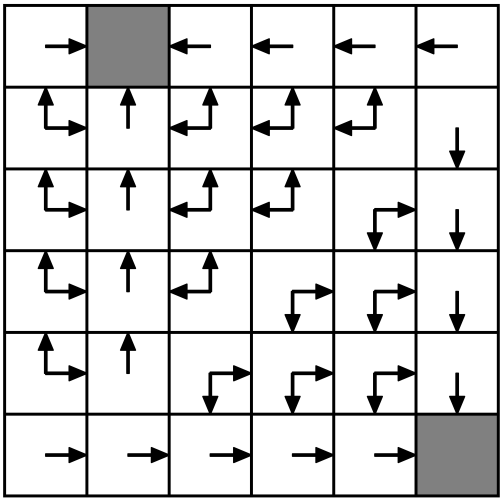
```
def calc_value(self, x, y):
    v = -float('inf')
    n = self.gridworld.n
    m = self.gridworld.m
    for action in self.gridworld.action:
        next_pos = np.array([x, y]) + np.array(action)
```

```
        if next_pos[0] < 0 or next_pos[0] >= n or next_pos[1] < 0 or next_pos[1] >= m:
            continue
        else:
            v = max(v, (self.gridworld.reward +
                        self.gridworld.discount_factor * self.gridworld.value[
                next_pos[0], next_pos[1]]) * self.gridworld.prob)

    return v
```

The results are given below. Due to the computational precision of the value matrix, the optimal action between policy-based method and value-based method is slightly different.

-0.25	0.0	-0.25	-0.31	-0.33	-0.33
-0.31	-0.25	-0.31	-0.33	-0.33	-0.33
-0.33	-0.31	-0.33	-0.33	-0.33	-0.33
-0.33	-0.33	-0.33	-0.33	-0.33	-0.31
-0.33	-0.33	-0.33	-0.33	-0.31	-0.25
-0.33	-0.33	-0.33	-0.31	-0.25	0.0



Comparison between two methods

For both methods, we control the small threshold θ to **0.001**, and print the number of iterations of each method.

For policy iteration method, the total iteration number is **346** for policy evaluation, and only twice for policy improvement.

For value iteration method, the total iteration number is **5**.

We can see that the iteration number of value iteration is **much smaller** than that of policy iteration. Both methods are guaranteed to converge; however, policy iteration is **more complex** and **requires more iterations to converge** in GridWorld.

In practice, **value iteration performs much better than policy iteration, and finds the optimal state function in much fewer steps.**

Conclusion

In summary, we have implemented both policy iteration method and value iteration method. Also, we have made a comparison of their performance. From my perspective, this assignment is truly meaningful, which deeply strengthen my understanding of MDP and dynamic programming methods. I hope that I can further strengthen my ability in the following assignments.