Reinforcement Learning: Assignment 2

17 July 2024

Github: https://github.com/zyxsjdy/Solve-the-Gridworld-Problem-with-Reinforcement-Learning

Part 1

0. Environment Setup

Consider a simple 5×5 gridworld problem:

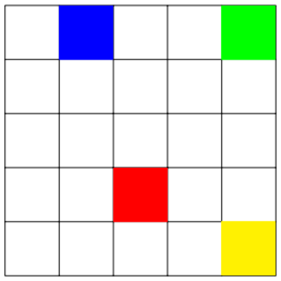
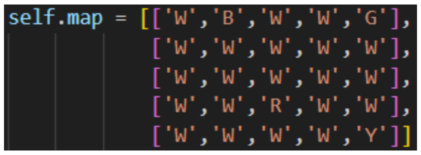


Fig. 1: Gridworld.

Each of the 25 cells of the gridworld represents a possible state of the world. The gridworld is implemented in the code as

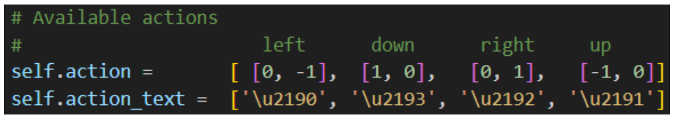


```
self.map = [['W','B','W','W','G'],
            ['W','W','W','W','W'],
            ['W','W','W','W','W'],
            ['W','W','R','W','W'],
            ['W','W','W','W','Y']]
```

Fig. 2: Gridworld map in code.

where $W, B, G, R,$ and Y represent the white, blue, green, red, and yellow grid, respectively. For example, $self.map[0][4] = \text{'}G\text{'}$, $self.map[3][2] = \text{'}R\text{'}$.

An agent in the gridworld environment can take a step up, down, left, or right, which is expressed as



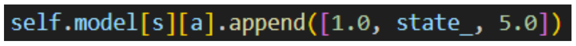
```
# Available actions
#                    left       down       right      up
self.action =      [ [0, -1],   [1, 0],    [0, 1],    [-1, 0]]
self.action_text = ['\u2190', '\u2193', '\u2192', '\u2191']
```

Fig. 3: Actions in code.

where the $self.action$ represents the actual action that is taken. For example, $self.action[0] = [0, -1]$ means keeping the index of the current row intact while reducing the index of the current column by 1. The second variable $self.action\_text$ is the Unicode text used for visualization, e.g. $\backslash u2190$ represents ←.

After the map and action settings are introduced, the model of this gridworld problem is described as follows. Based on the current state and action (s, a), the model returns a sequence (p, s', r) where p denotes the probability of transiting from state s to state s' under action a.

If the current state is in the blue grid, i.e. $self.map[row][col] = 'B'$, the next state will be in the red grid no matter which action is taken, and a reward of 5.0 is given. The model for the blue grid is implemented in code as

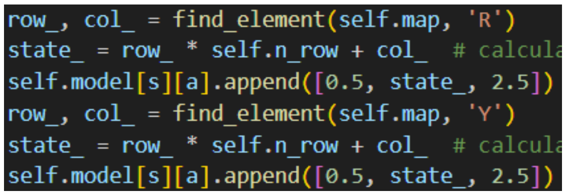


```
self.model[s][a].append([1.0, state_, 5.0])
```

Fig. 4: Model for the blue grid in code.

where s is the state in blue grid, and for any action a, $state\_$ is the next state in the red grid.

If the current state is in the green grid, i.e. $self.map[row][col] = 'G'$, the next state will be either in the red grid or the yellow grid with equal probability no matter which action is taken, and a reward of 2.5 is given. The model for the green grid is implemented as

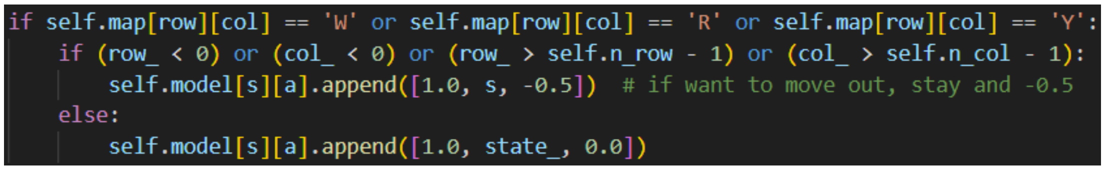


```
row_, col_ = find_element(self.map, 'R')
state_ = row_ * self.n_row + col_  # calcula
self.model[s][a].append([0.5, state_, 2.5])
row_, col_ = find_element(self.map, 'Y')
state_ = row_ * self.n_row + col_  # calcula
self.model[s][a].append([0.5, state_, 2.5])
```

Fig. 5: Model for the green grid in code.

where the probabilities of jumping to the red or the yellow grid are the same, 0.5.

For the other states: if the agent attempts to step off the grid after taking any actions as shown in Fig. 3, i.e. the row index or the column index is less than 0 or larger than 4, the next state will be the same as the current state, and a reward of -0.5 is given. Otherwise, the agent will move normally with a reward of 0.0.



```
if self.map[row][col] == 'W' or self.map[row][col] == 'R' or self.map[row][col] == 'Y':
    if (row_ < 0) or (col_ < 0) or (row_ > self.n_row - 1) or (col_ > self.n_col - 1):
        self.model[s][a].append([1.0, s, -0.5])  # if want to move out, stay and -0.5
    else:
        self.model[s][a].append([1.0, state_, 0.0])
```

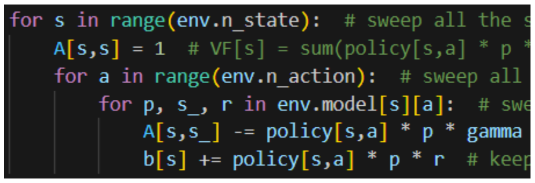
Fig. 6: Model for the other grids in code.

Intuitively, an agent with a good policy should try to find the states with a high value and exploit the rewards available at those states.

1. Estimate the value function

1.1 Solving the system of Bellman equations explicitly

Based on the Bellman equations shown on the right panel of Fig. 7, we set up a linear equation as shown in the left panel of Fig. 7.



```
for s in range(env.n_state):  # sweep all the s
    A[s,s] = 1  # VF[s] = sum(policy[s,a] * p *
    for a in range(env.n_action):  # sweep all
        for p, s_, r in env.model[s][a]:  # swe
            A[s,s_] -= policy[s,a] * p * gamma
            b[s] += policy[s,a] * p * r  # keep
```

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S},
\end{aligned}
$$

Fig. 7: Building the equation set according to the Bellman equations.

where A is initialized as a zero matrix. For each state s, $A[s,s]$ is set as 1, which represents the coefficient of $v(s)$, and $A[s,s']$ shown in the fifth line on the left panel in Fig.7 represents the coefficient of $v(s')$ after moving to the left side of the equation so a the minus sign is used. $b[s]$ shown in the last line on the left panel in Fig.7 represents the constant in the Bellman equation for state s. After this step, the linear equation is transferred to the matrix multiplication $Ax = b$, where A is a 25×25 matrix representing the coefficients for all the states in the Bellman equation, x is a 25×1 matrix representing the 25 unknown $v(s)$, b is also a 25×1 matrix representing the constant in the Bellman equation. x can be simply solved by multiplying the inverse of matrix A with b.

1.2 Iterative policy evaluation

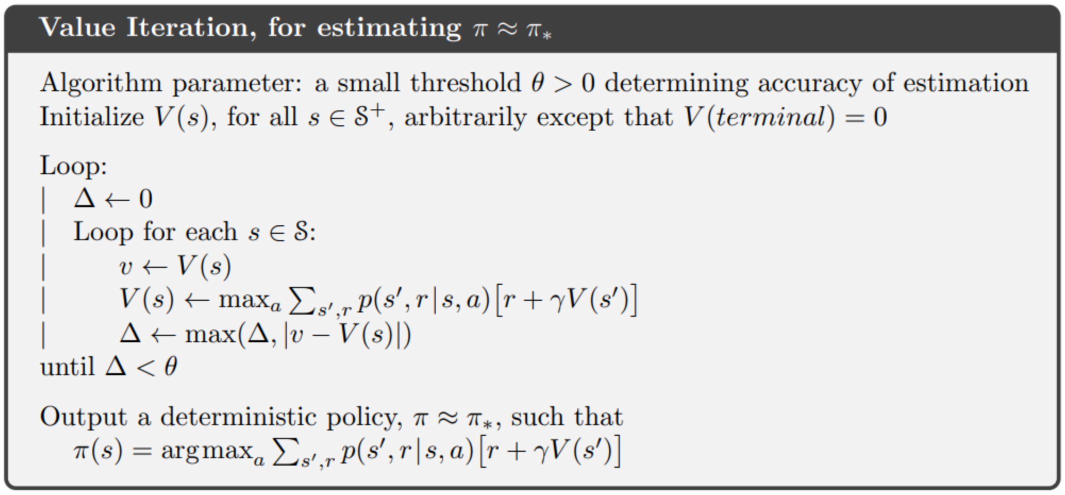
This algorithm can be implemented according to the pseudocode given in the book.

```
Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input $\pi$, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$ arbitrarily, for $s \in \mathcal{S}$, and $V(terminal)$ to 0

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

Fig. 8: Iterative policy evaluation.

1.3 Value iteration

Similarly, this algorithm can be implemented according to the pseudocode given in the book.

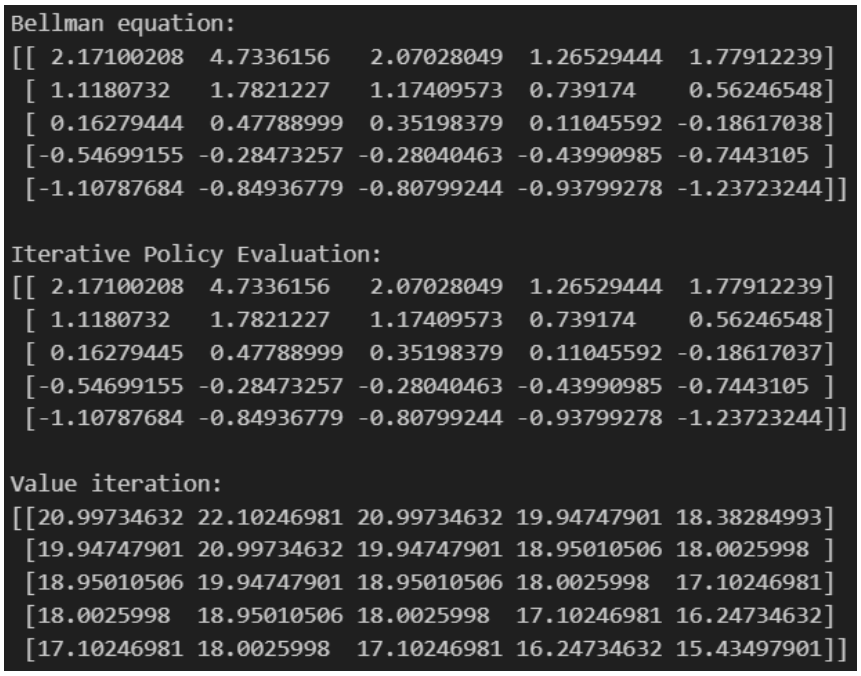


```
Value Iteration, for estimating $\pi \approx \pi_*$

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

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
```

Fig. 9: Value iteration.

1.4 Results

With a discount factor $\gamma = 0.95$ and a policy with equiprobable moves, the results are shown as follows.



```
Bellman equation:
[[ 2.17100208  4.7336156   2.07028049  1.26529444  1.77912239]
 [ 1.1180732   1.7821227   1.17409573  0.739174    0.56246548]
 [ 0.16279444  0.47788999  0.35198379  0.11045592 -0.18617038]
 [-0.54699155 -0.28473257 -0.28040463 -0.43990985 -0.7443105 ]
 [-1.10787684 -0.84936779 -0.80799244 -0.93799278 -1.23723244]]

Iterative Policy Evaluation:
[[ 2.17100208  4.7336156   2.07028049  1.26529444  1.77912239]
 [ 1.1180732   1.7821227   1.17409573  0.739174    0.56246548]
 [ 0.16279445  0.47788999  0.35198379  0.11045592 -0.18617037]
 [-0.54699155 -0.28473257 -0.28040463 -0.43990985 -0.7443105 ]
 [-1.10787684 -0.84936779 -0.80799244 -0.93799278 -1.23723244]]

Value iteration:
[[20.99734632 22.10246981 20.99734632 19.94747901 18.38284993]
 [19.94747901 20.99734632 19.94747901 18.95010506 18.0025998 ]
 [18.95010506 19.94747901 18.95010506 18.0025998  17.10246981]
 [18.0025998  18.95010506 18.0025998  17.10246981 16.24734632]
 [17.10246981 18.0025998  17.10246981 16.24734632 15.43497901]]
```

Fig. 10: Results of the three approaches.

The first and second approaches almost show the same results (only small differences in the last few digits, which is caused by the estimation of the Iterative policy evaluation), while the third approach gives a significantly different result.

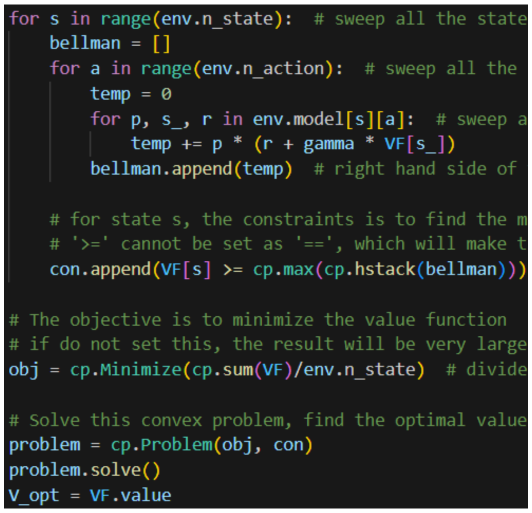
For all the approaches, the blue grid has the highest value. This is undoubted since it gives a reward of 5 for any action, which is much higher than acting in the green grid (giving a reward of 2.5) and in the other grids (giving a reward of 0 or -0.5).

For the first and second approaches, the value of the green grid is the fifth largest, just less than the blue grid and the three grids around the blue grid. However, for the third approach, the value of the green grid is less than the values of a lot of grids. Even 2 grids around the green grid tend not to move to the green grid (the grid on the left of the green grid has a larger value than that of the green grid; the grid below the green grid tends to move to the left since that grid has a larger value compared to that of the green grid $18.95 \ldots > 18.38 \ldots$). This may be caused by the extremely high value in the blue grid (5.0), in which case the agent tends to move to the blue grid; and also when the agent arrives at the green grid, there is a 0.5 probability of jumping to the yellow grid, which is at the edge grid (easier to move outside of the grid, giving a -0.5 reward) that is far from the blue grid and the green, so the agent tends not to move to the green grid.

2. Find the optimal policy

2.1 Explicitly solving the Bellman optimality equation

Based on the Bellman optimality equations, we set up a linear programming problem,



```
for s in range(env.n_state):  # sweep all the state
    bellman = []
    for a in range(env.n_action):  # sweep all the
        temp = 0
        for p, s_, r in env.model[s][a]:  # sweep a
            temp += p * (r + gamma * VF[s_])
        bellman.append(temp)  # right hand side of

    # for state s, the constraints is to find the m
    # '>=' cannot be set as '==', which will make t
    con.append(VF[s] >= cp.max(cp.hstack(bellman)))

# The objective is to minimize the value function
# if do not set this, the result will be very large
obj = cp.Minimize(cp.sum(VF)/env.n_state)  # divide

# Solve this convex problem, find the optimal value
problem = cp.Problem(obj, con)
problem.solve()
V_opt = VF.value
```

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_*(s')\big]. \end{aligned}$$

Fig. 11: Building the problem according to the Bellman optimality equations.

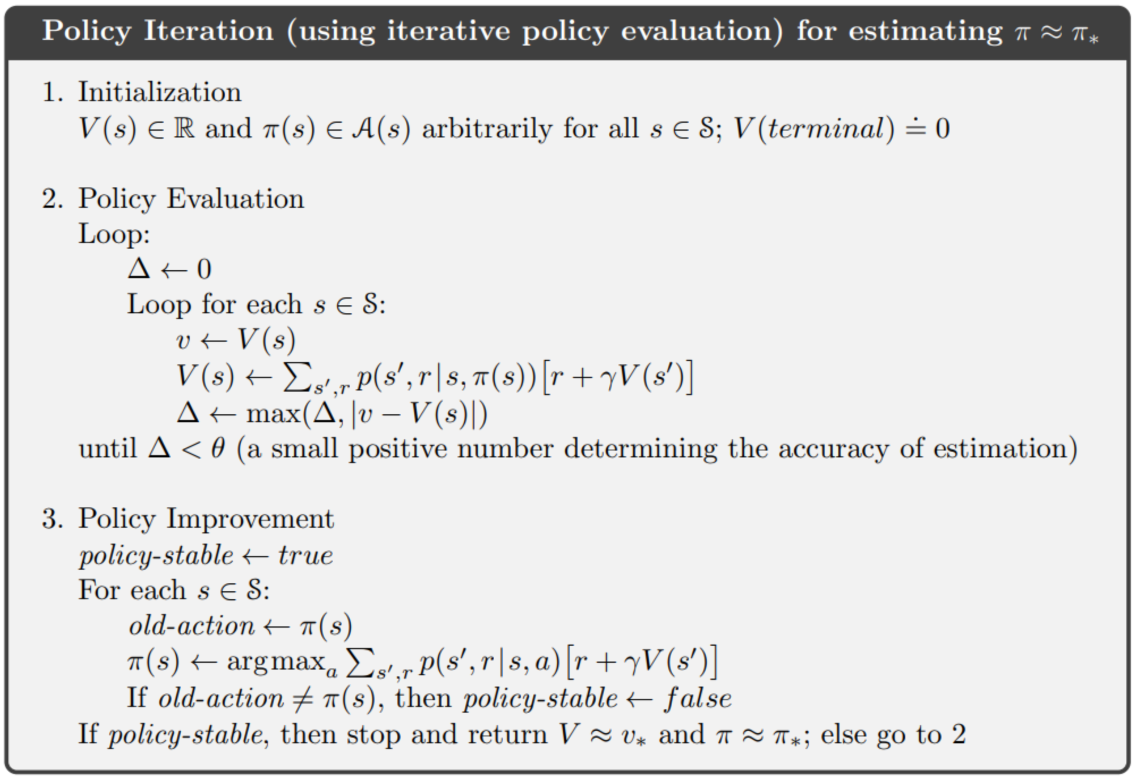
We use library *cvxpy* to solve this linear programming problem. For each action, one optimal action value function q is appended, and for each state, one constraint $V \geq$

$max(q)$ is appended. There are 4 q functions for 4 actions, and 25 constraints for 25 states. The objective is to minimize the sum of the value function.

There are two points needed to be highlighted. First, the idea of using $\geq$ instead of $==$ is to satisfy the *DCP* rules of the problem setting. Second, the idea of setting the objective as minimizing the sum of the value function is because of the multiple solutions of this linear programming problem. If the objective were not set, the value of each grid could reach 50 or even 100.

2.2 Using policy iteration with iterative policy evaluation

This algorithm can be implemented according to the pseudocode given in the book.



```
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 $s \in \mathcal{S}$; $V(terminal) \doteq 0$

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

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

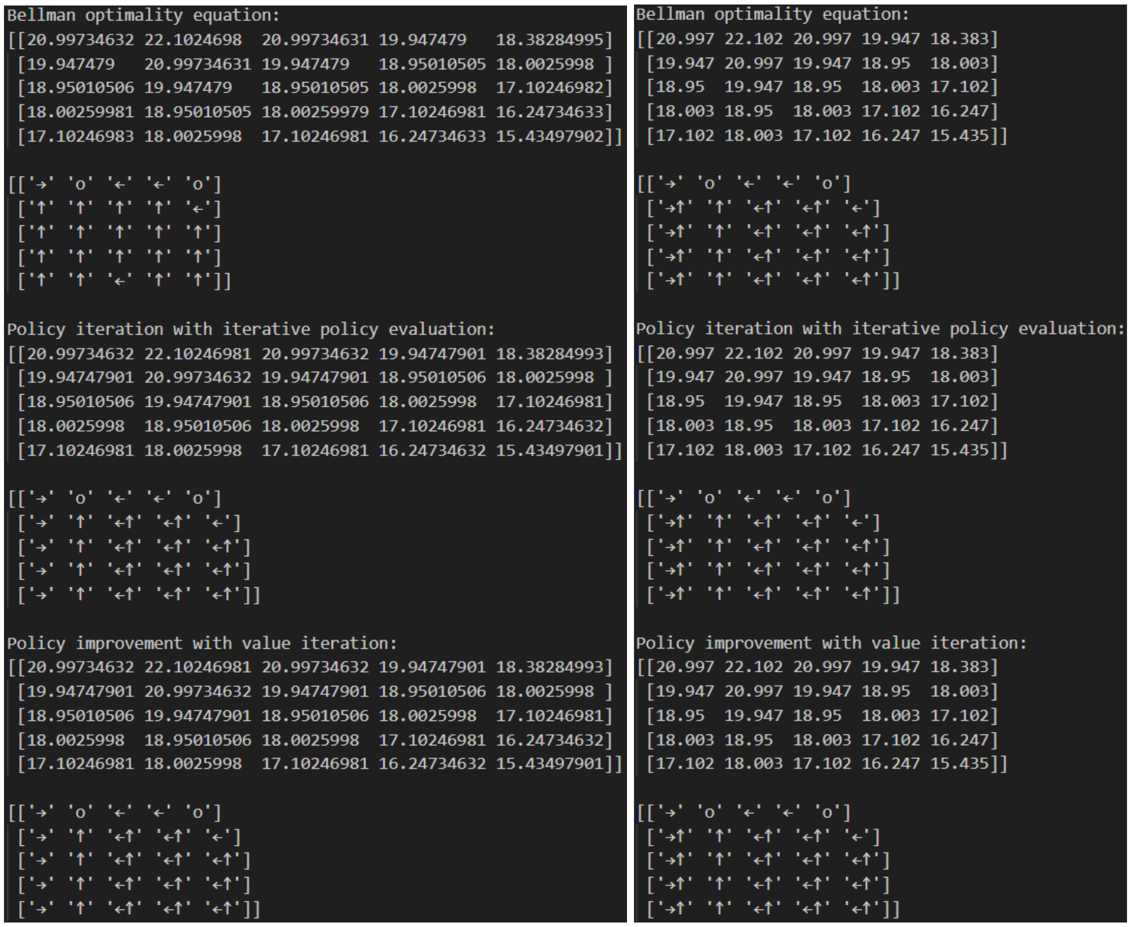
Fig. 12: Policy iteration.

2.3 Policy improvement with value iteration

This algorithm can be implemented by replacing the policy evaluation part in Fig. 12 with the loop of the value iteration in Fig. 9.

2.4 Results

With the same discount factor $\gamma = 0.95$, the results are shown as follows.



```
Bellman optimality equation:
[[20.99734632 22.1024698  20.99734631 19.947479   18.38284995]
 [19.947479   20.99734631 19.947479   18.95010505 18.0025998 ]
 [18.95010506 19.947479   18.95010505 18.0025998  17.10246982]
 [18.00259981 18.95010505 18.00259979 17.10246981 16.24734633]
 [17.10246983 18.0025998  17.10246981 16.24734633 15.43497902]]

[['→' 'o' '←' '←' 'o']
 ['↑' '↑' '↑' '↑' '←']
 ['↑' '↑' '↑' '↑' '↑']
 ['↑' '↑' '↑' '↑' '↑']
 ['↑' '↑' '←' '↑' '↑']]

Policy iteration with iterative policy evaluation:
[[20.99734632 22.10246981 20.99734632 19.94747901 18.38284993]
 [19.94747901 20.99734632 19.94747901 18.95010506 18.0025998 ]
 [18.95010506 19.94747901 18.95010506 18.0025998  17.10246981]
 [18.0025998  18.95010506 18.0025998  17.10246981 16.24734632]
 [17.10246981 18.0025998  17.10246981 16.24734632 15.43497901]]

[['→' 'o' '←' '←' 'o']
 ['→' '↑' '←↑' '←↑' '←']
 ['→' '↑' '←↑' '←↑' '←↑']
 ['→' '↑' '←↑' '←↑' '←↑']
 ['→' '↑' '←↑' '←↑' '←↑']]

Policy improvement with value iteration:
[[20.99734632 22.10246981 20.99734632 19.94747901 18.38284993]
 [19.94747901 20.99734632 19.94747901 18.95010506 18.0025998 ]
 [18.95010506 19.94747901 18.95010506 18.0025998  17.10246981]
 [18.0025998  18.95010506 18.0025998  17.10246981 16.24734632]
 [17.10246981 18.0025998  17.10246981 16.24734632 15.43497901]]

[['→' 'o' '←' '←' 'o']
 ['→' '↑' '←↑' '←↑' '←']
 ['→' '↑' '←↑' '←↑' '←↑']
 ['→' '↑' '←↑' '←↑' '←↑']
 ['→' '↑' '←↑' '←↑' '←↑']]
```

```
Bellman optimality equation:
[[20.997 22.102 20.997 19.947 18.383]
 [19.947 20.997 19.947 18.95  18.003]
 [18.95  19.947 18.95  18.003 17.102]
 [18.003 18.95  18.003 17.102 16.247]
 [17.102 18.003 17.102 16.247 15.435]]

[['→' 'o' '←' '←' 'o']
 ['→↑' '↑' '←↑' '←↑' '←']
 ['→↑' '↑' '←↑' '←↑' '←↑']
 ['→↑' '↑' '←↑' '←↑' '←↑']
 ['→↑' '↑' '←↑' '←↑' '←↑']]

Policy iteration with iterative policy evaluation:
[[20.997 22.102 20.997 19.947 18.383]
 [19.947 20.997 19.947 18.95  18.003]
 [18.95  19.947 18.95  18.003 17.102]
 [18.003 18.95  18.003 17.102 16.247]
 [17.102 18.003 17.102 16.247 15.435]]

[['→' 'o' '←' '←' 'o']
 ['→↑' '↑' '←↑' '←↑' '←']
 ['→↑' '↑' '←↑' '←↑' '←↑']
 ['→↑' '↑' '←↑' '←↑' '←↑']
 ['→↑' '↑' '←↑' '←↑' '←↑']]

Policy improvement with value iteration:
[[20.997 22.102 20.997 19.947 18.383]
 [19.947 20.997 19.947 18.95  18.003]
 [18.95  19.947 18.95  18.003 17.102]
 [18.003 18.95  18.003 17.102 16.247]
 [17.102 18.003 17.102 16.247 15.435]]

[['→' 'o' '←' '←' 'o']
 ['→↑' '↑' '←↑' '←↑' '←']
 ['→↑' '↑' '←↑' '←↑' '←↑']
 ['→↑' '↑' '←↑' '←↑' '←↑']
 ['→↑' '↑' '←↑' '←↑' '←↑']]
```

Fig. 13: Results of the three approaches.

According to the left panel of Fig. 13 (‘*o*’ means moving to four directions are the same), it is clear that the three approaches almost give the same value functions, with only minor differences in the last few digits, which might be caused by the estimation problem or precision error. This problem causes the differences and incorrectness in the derived optimal policy.

After rounding the value function, the correct result is shown on the right panel of Fig. 13. The three approaches give the same value functions (after rounding) and the same optimal policies. The value functions also show the same result as the result given by value iteration shown in Fig. 10 in section 1.4. These results also prove our guess about why the value of the green grid is quite low. The optimal policies show that the agent does not want to move to the green grid, its target is always the blue grid.

Part 2

0. Environment Setup

The 5×5 gridworld problem is changed to:

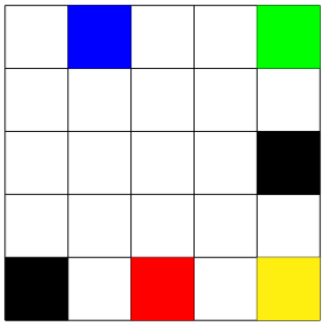
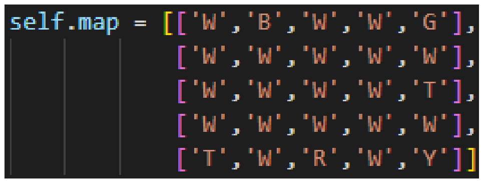


Fig. 14: New Gridworld.

The new gridworld is implemented in the code as

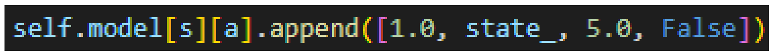


```
self.map = [['W','B','W','W','G'],
            ['W','W','W','W','W'],
            ['W','W','W','W','T'],
            ['W','W','W','W','W'],
            ['T','W','R','W','Y']]
```

Fig. 15: New Gridworld map in code.

where 'T' is the black grid representing the terminal state.

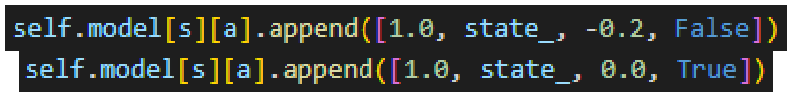
In addition to the sequence (p, s', r) returned by the model mentioned in Part 1, a bool value is added to the sequence: (p, s', r, t), where t is $'True'$ or $'False'$ representing whether the next state s' is a terminal state. For example,



```
self.model[s][a].append([1.0, state_, 5.0, False])
```

Fig. 16: Example of the new model for the new gridworld.

Furthermore, the movement of the agent from a white grid to another white grid yields a reward of -0.2, and the movement of the agent from a white grid to the terminal grid yields a reward of 0.0:



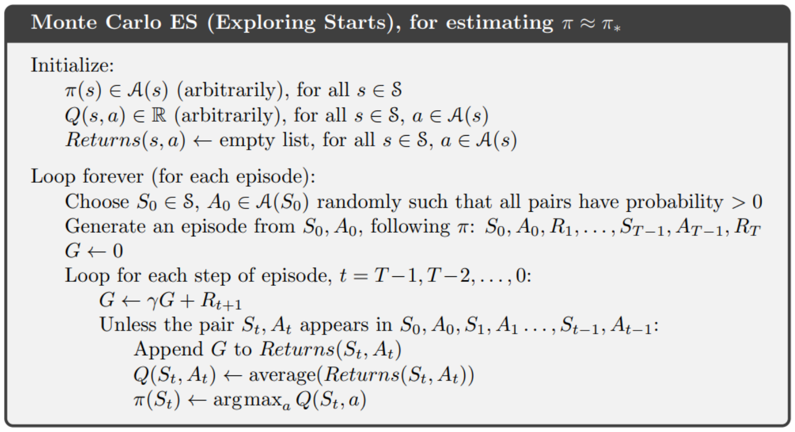
```
self.model[s][a].append([1.0, state_, -0.2, False])
self.model[s][a].append([1.0, state_, 0.0, True])
```

Fig. 17: Example of the new model for the new gridworld.

1. Monte Carlo Method

1.1 Exploring starts

This algorithm can be implemented according to the pseudocode given in the book.

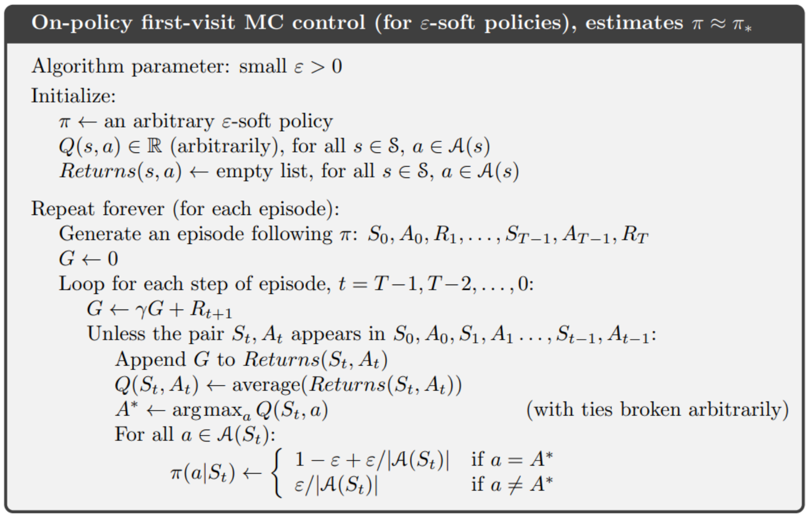


```
Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
    $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
    Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability $> 0$
    Generate an episode from $S_0, A_0$, following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
            $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
```

Fig. 18: Monte Carlo with Exploring Starts.

1.2 ε-soft approach

Similarly, this algorithm can be implemented according to the pseudocode given in the book.

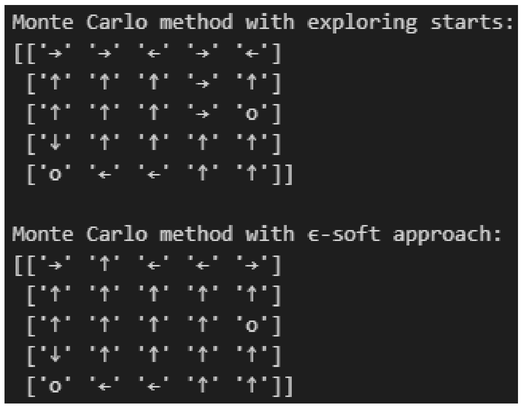


```
On-policy first-visit MC control (for $\varepsilon$-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
    $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
            $A^* \leftarrow \arg\max_a Q(S_t, a)$                    (with ties broken arbitrarily)
            For all $a \in \mathcal{A}(S_t)$:
                $\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
```

Fig. 19: Monte Carlo with ε-soft approach.

1.3 Results

Starting with a policy of equiprobable moves with the same discount factor $\gamma = 0.95$, the results are shown as follows.

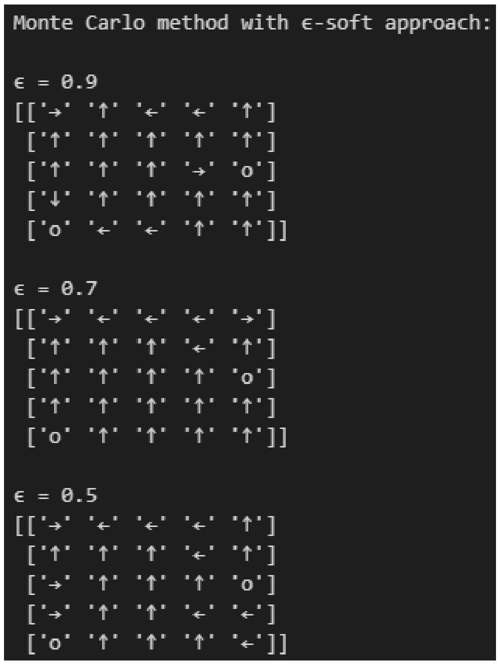


```
Monte Carlo method with exploring starts:
[['→' '→' '←' '→' '←']
 ['↑' '↑' '↑' '→' '↑']
 ['↑' '↑' '↑' '→' 'o']
 ['↓' '↑' '↑' '↑' '↑']
 ['o' '←' '←' '↑' '↑']]

Monte Carlo method with ϵ-soft approach:
[['→' '↑' '←' '←' '→']
 ['↑' '↑' '↑' '↑' '↑']
 ['↑' '↑' '↑' '↑' 'o']
 ['↓' '↑' '↑' '↑' '↑']
 ['o' '←' '←' '↑' '↑']]
```

Fig. 20: Monte Carlo Results (for ε-soft approach, ε = 0.85 and starts from grid [4,3]).

Basically, the two approaches give a similar result. Some small differences are shown: for the first approach, the agent moves to the right from grid [0,3] to reach the green grid, while it moves to the left to reach the blue grid for the second approach; and from grid [2,3], the agent moves right to terminate the episode for the first approach, while the agent just moves up for the second approach.

For the Monte Carlo with ε-soft approach, we also analyse the impact of the value of ε to the result, from 0.9 to 0.1, which is shown in the following figure.



```
Monte Carlo method with ϵ-soft approach:

ϵ = 0.9
[['→' '↑' '←' '←' '↑']
 ['↑' '↑' '↑' '↑' '↑']
 ['↑' '↑' '↑' '→' 'o']
 ['↓' '↑' '↑' '↑' '↑']
 ['o' '←' '←' '↑' '↑']]

ϵ = 0.7
[['→' '←' '←' '←' '→']
 ['↑' '↑' '↑' '←' '↑']
 ['↑' '↑' '↑' '↑' 'o']
 ['↑' '↑' '↑' '↑' '↑']
 ['o' '↑' '↑' '↑' '↑']]

ϵ = 0.5
[['→' '←' '←' '←' '↑']
 ['↑' '↑' '↑' '←' '↑']
 ['→' '↑' '↑' '↑' 'o']
 ['→' '↑' '↑' '←' '←']
 ['o' '↑' '↑' '↑' '←']]
```

Fig. 21: Monte Carlo with ε-soft approach ($\varepsilon = 0.9, 0.7, 0.5$).

The maximum episodes for $\varepsilon = 0.9, 0.7, 0.5$ are set as $5000, 1000$, and 500. For $\varepsilon = 0.9$, the result shows that the agent is more likely to move to the terminal state compared to $\varepsilon = 0.85$ as shown in Fig. 20 (the difference is shown at grid $[2,3]$). For $\varepsilon = 0.7$, the agent is less likely to move to the terminal and keep moving to the blue grid. For $\varepsilon = 0.5$, the agent almost does not move to the terminal grid (no arrow is pointing to the terminal grid). This causes the code to finish running after a long time; so the agent may move to the terminal and move to the next episode only with a very small probability of exploration.

```
✓ 22m 17.1s
```

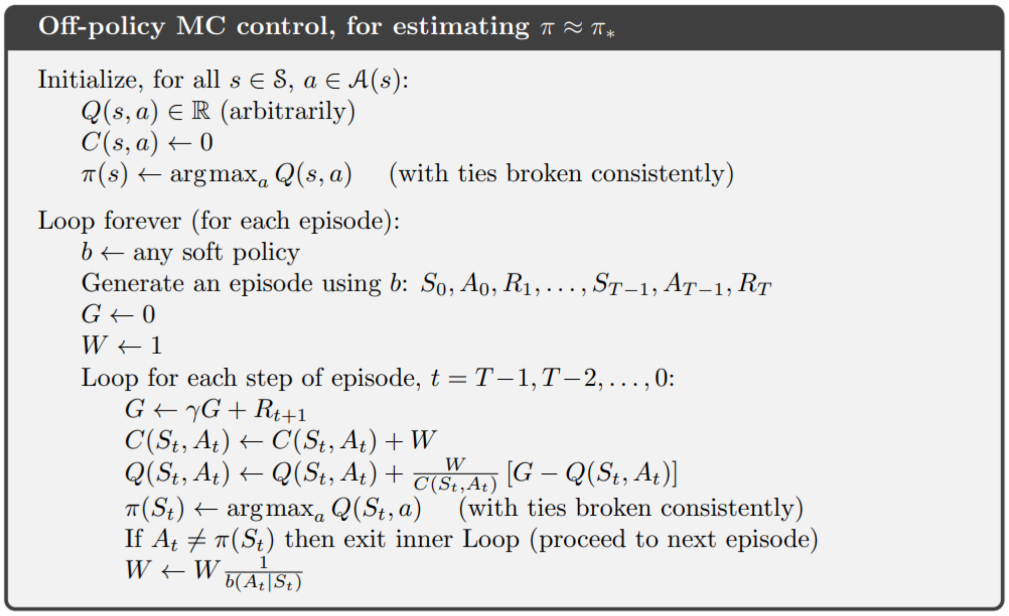
Fig. 22: The amount of time needed to get the result in Fig. 21.

The main reason behind this phenomenon is that with a small ε, the agent tends not to discover the world, i.e., exploration. It is more likely to follow the existing best choice which leads the agent only towards the blue grid. This is just a suboptimal result of the gridworld problem. Only when the ε is larger, which encourages the agent to explore, the agent can find a better solution which is closer to the optimal policy, or maybe, find the optimal policy.

2. Monte Carlo Method with a behaviour policy

2.1 Algorithm

This algorithm can be implemented according to the pseudocode given in the book.

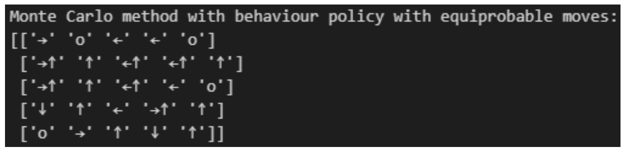


```
Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$
    $\pi(s) \leftarrow \operatorname{argmax}_a Q(s,a)$    (with ties broken consistently)

Loop forever (for each episode):
    $b \leftarrow$ any soft policy
    Generate an episode using $b$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
        $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$    (with ties broken consistently)
        If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
```

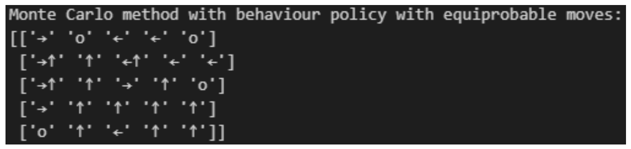
Fig. 23: Off-policy Monte Carlo control.

2.2 Results

With the same discount factor $\gamma = 0.95$ and a behaviour policy with equiprobable moves, the results are shown as follows.



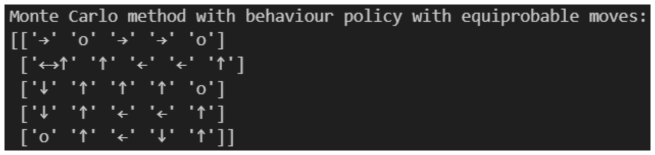
```
Monte Carlo method with behaviour policy with equiprobable moves:
[['→' 'o' '←' '←' 'o']
 ['→↑' '↑' '←↑' '←↑' '↑']
 ['→↑' '↑' '←↑' '←' 'o']
 ['↓' '↑' '←' '→↑' '↑']
 ['o' '→' '↑' '↓' '↑']]
```



```
Monte Carlo method with behaviour policy with equiprobable moves:
[['→' 'o' '←' '←' 'o']
 ['→↑' '↑' '←↑' '←' '←']
 ['→↑' '↑' '→' '↑' 'o']
 ['→' '↑' '↑' '↑' '↑']
 ['o' '↑' '←' '↑' '↑']]
```

Fig. 24: Two example results of the off-policy Monte Carlo control with exploring starts.

For off-policy Monte Carlo control, it needs a long time to converge comparing with the previously mentioned methods. By setting the maximum episodes as 200000 and with exploring starts, we get the result shown in Fig. 24. Due to the large size of the gridworld, it is difficult for the algorithm to go through all the state-action pairs, especially those far from the terminal grid. This problem can be shown in the grid [4,3] in the upper panel and the grid [2,2] in the lower panel of Fig. 24.



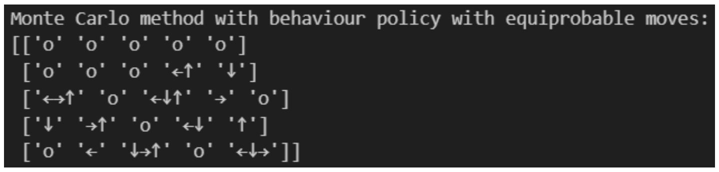
```
Monte Carlo method with behaviour policy with equiprobable moves:
[['→' 'o' '→' '→' 'o']
 ['↔↑' '↑' '←' '←' '↑']
 ['↓' '↑' '↑' '↑' 'o']
 ['↓' '↑' '←' '←' '↑']
 ['o' '↑' '←' '↓' '↑']]
```

Fig. 25: Example result of the off-policy Monte Carlo control with fix start ($s_0 = 23$).

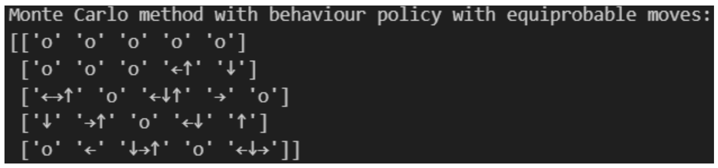
From Fig. 25, we can see that the off-policy Monte Carlo control does not work well with a fix starting point, the agent acts improperly at some grid, e.g., grid [1,0] and [4,3]. This is caused by the fact that the off-policy Monte Carlo control is difficult to go through all the state-action pairs. It sometimes converges very early when it finds a suboptimal solution and will never update some state-action pairs because the optimal policy it found will not take the same action as the behaviour policy (always exits the inner loop). We think the difference between exploring starts and fix start is that for exploring starts, all the states have similar probability to be at the beginning of the trajectory. Meanwhile, for the fix start, the grids around the fix start will always be at the beginning of the trajectory, and it is difficult for the algorithm to go through. The algorithm always converges before going through those state-action pairs.

Another point needs to be mentioned is that the Q function is initialized as -0.21, comparing with the -0.2 reward of moving normally (white to white). This allows the algorithm to start to learn the policy easier. If just initialize the Q function as 0, for example, if the agent decided to move up from a white grid to a white grid and get a reward -0.2, the Q function will be updated, and the policy will be updated based on the Q function. Then, the updated policy will never take the action taken by the behaviour policy, since -0.2 is less than 0. By setting the initial Q function as -0.21, the algorithm will learn more from those normal action, and try to find the optimal solution.

Last but not least, instead of setting the reward of moving to the black gird as -0.2, the same as moving from a white grid to a white grid, we set it as 0. The idea behind this comes from the result we get by setting it as -0.2, as shown in the following figure.



```
Monte Carlo method with behaviour policy with equiprobable moves:
[['o' 'o' 'o' 'o' 'o']
 ['o' 'o' 'o' '←↑' '↓']
 ['↔↑' 'o' '←↓↑' '→' 'o']
 ['↓' '→↑' 'o' '←↓' '↑']
 ['o' '←' '↓→↑' 'o' '←↓→']]
```



```
Monte Carlo method with behaviour policy with equiprobable moves:
[['o' 'o' 'o' 'o' 'o']
 ['o' 'o' 'o' '←↑' '↓']
 ['↔↑' 'o' '←↓↑' '→' 'o']
 ['↓' '→↑' 'o' '←↓' '↑']
 ['o' '←' '↓→↑' 'o' '←↓→']]
```

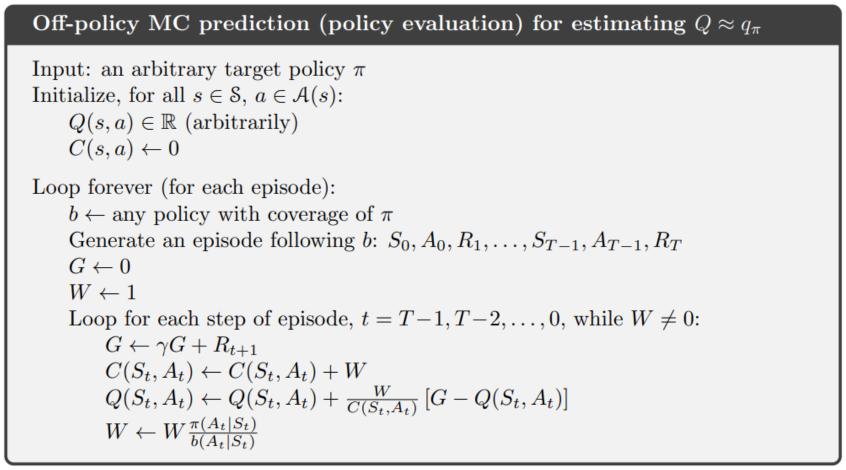
Fig. 26: Example result of the off-policy Monte Carlo control (moving to the terminal yields a reward -0.2). Upper panel: with exploring starts. Lower pannel: with a fix start ($s_0 = 23$).

From Fig. 26 we can see that there is no difference between using exploring starts and a fix start. By setting the terminal to yield a reward -0.2, the algorithm will almost always exit the inner loop and never do further updates. This proves that we should set moving to the terminal grid to yield a reward 0.0, or at least larger than -0.2, so that the updated optimal policy will take the same action as the behaviour policy to move the agent to the terminal and then the inner loop will continue to the previous steps.

3. Monte Carlo Method with policy iteration

3.1 Algorithm

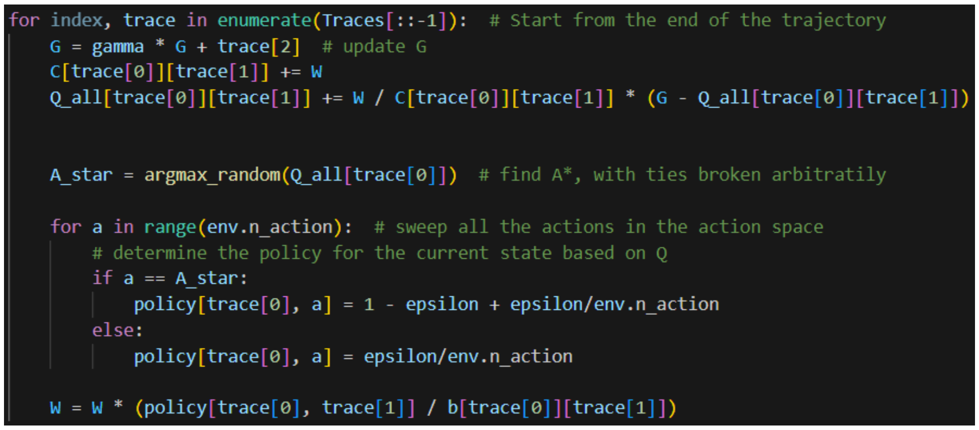
According to the pseudocode for policy evaluation, the algorithm can be implemented by combining it with the ε-soft update approach in Fig. 19.



Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

```
Input: an arbitrary target policy $\pi$
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s, a) \in \mathbb{R}$ (arbitrarily)
    $C(s, a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of $\pi$
    Generate an episode following $b$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
```

Fig. 27: Off-policy Monte Carlo Prediction.

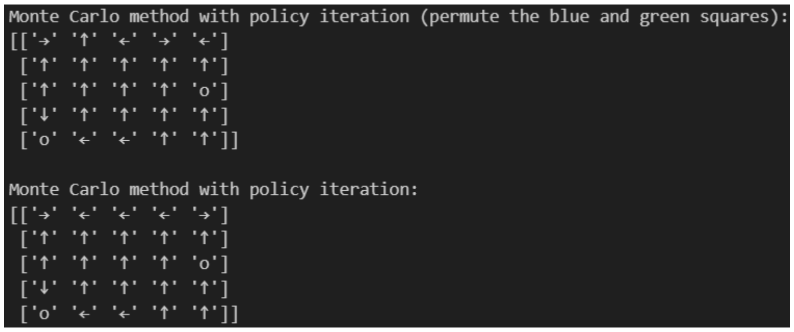


```python
for index, trace in enumerate(Traces[::-1]):  # Start from the end of the trajectory
    G = gamma * G + trace[2]  # update G
    C[trace[0]][trace[1]] += W
    Q_all[trace[0]][trace[1]] += W / C[trace[0]][trace[1]] * (G - Q_all[trace[0]][trace[1]])


    A_star = argmax_random(Q_all[trace[0]])  # find A*, with ties broken arbitratily

    for a in range(env.n_action):  # sweep all the actions in the action space
        # determine the policy for the current state based on Q
        if a == A_star:
            policy[trace[0], a] = 1 - epsilon + epsilon/env.n_action
        else:
            policy[trace[0], a] = epsilon/env.n_action

    W = W * (policy[trace[0], trace[1]] / b[trace[0]][trace[1]])
```

Fig. 28: Adding the ε-soft approach to update the policy between the update of the Q function and the importance weight W.

3.2 Results

With the same discount factor $\gamma = 0.95$ and a behaviour policy with equiprobable moves, the results are shown as follows.



```
Monte Carlo method with policy iteration (permute the blue and green squares):
[['→' '↑' '←' '→' '←']
 ['↑' '↑' '↑' '↑' '↑']
 ['↑' '↑' '↑' '↑' 'o']
 ['↓' '↑' '↑' '↑' '↑']
 ['o' '←' '←' '↑' '↑']]

Monte Carlo method with policy iteration:
[['→' '←' '←' '←' '→']
 ['↑' '↑' '↑' '↑' '↑']
 ['↑' '↑' '↑' '↑' 'o']
 ['↓' '↑' '↑' '↑' '↑']
 ['o' '←' '←' '↑' '↑']]
```

Fig. 29: Result of the combination of the off-policy Monte Carlo Prediction and the ε-soft update approach ($\varepsilon = 0.85$).

From Fig. 29, we can see that with and without permuting the blue and green squares, there are three differences. The first and second differences are shown in grid [0,1] and grid [0,4], the blue and green grids. These two differences can be ignored since taking any action in these two grids is actually the same. The last difference is shown in grid [0,3]. Permuting the blue and green squares, the agent tends to move to the right from this grid, while it tends to move to the left for the case without permuting.

We think the reason of this phenomenon is that the agent tends to move to the blue grid compared to moving to the green grid. If we do not permute the blue and green squares, the agent will move to the left, the blue grid, at grid [0,3]. However, by permuting the grid, there is 0.1 probability at each step that the blue and green squares will exchange their positions. This probability will be accumulated during the movement of the agent in the gridworld. Therefore, there would be a relatively high chance that the green grid becomes the blue grid when the agent moves from grid [0,3] to grid [0,4]. This could be the reason why the agent moves to the right from grid [0,3] in the case of permuting the blue and green squares.