Table of Contents

LAB13 tutorial for Machine Learning Markov Decision Process

> The document description are designed by JIa Yanhong in 2022. Nov. 22th

---

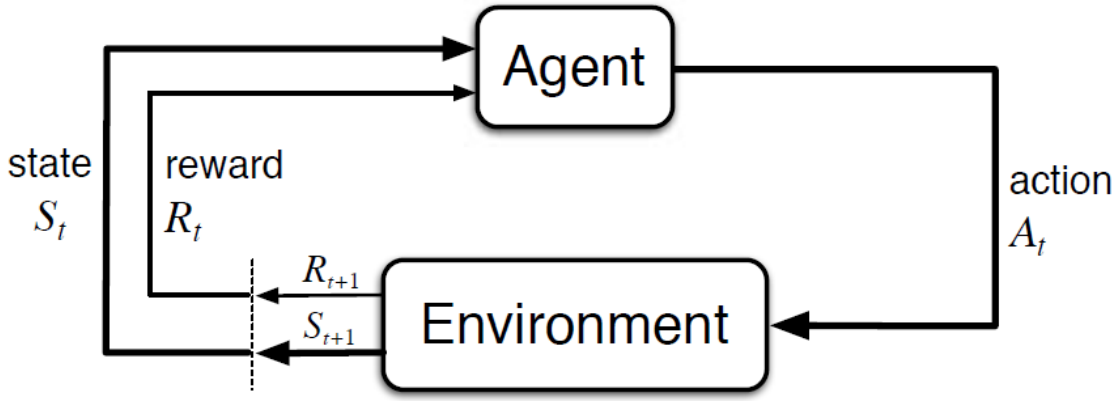
1 Objective

- Understand the theory of Markov Decision Process
- Implement the Value Iteration or Policy Iteration in python
- Complete the LAB assignment and submit it to BB or sakai.

---

2 Introduction

In RL, we build an agent that can **make smart decisions**. For instance, an agent that **learns to play a video game.** Or a trading agent that **learns to maximize its benefits** by making smart decisions on **what stocks to buy and when to sell.**



But, to make intelligent decisions, our agent will learn from the environment by **interacting with it through trial and error** and receiving rewards (positive or negative) **as unique feedback.**

Its goal **is to maximize its expected cumulative reward** (because of the reward hypothesis).

The agent's decision-making process is called the policy π: given a state, a policy will output an action or a probability distribution over actions. That is, given an observation of the environment, a policy will provide an action (or multiple probabilities for each action) that the agent should take.

Our goal is to find an optimal policy π, a policy that leads to the best expected cumulative reward.

So our root question for this lab is how we formulate any problem in RL mathematically ,This is where the Markov Decision Process(MDP) comes in.

Markov Decision Process (MDP) is a mathematical framework to describe an environment in reinforcement learning.

3 Markov Decision Process

Markov Decision Process (MDP) consists of a tuple of 5 elements $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$

- $\mathcal{S}$: set of states → At each time step, state of the environment is an element $s \in \mathcal{S}$.
- $\mathcal{A}$: set of actions → At each time step, agent takes an action $a \in \mathcal{A}$.
- $\mathcal{P}$: state transition model (matrix)

$$\mathcal{P}^a_{x\to x'} = P[S_{t+1}|S_t = s, A_t = a]$$

Probability of transition to next state x' after taking action a in current state x.

- $\mathcal{R}$: reward model (matrix)

$$\mathcal{R}^a_s = E[R_{t+1}|S_t = s, A_t = a]$$

Reward obtained after taking action a in current state x. (to be more general, $\mathcal{R}^a_{x\to x'} = E[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s']$)

- γ: discount factor → Control the importance of future rewards.

The goal is to find an optimal policy π maximizing a cumulative function of the random rewards.

3.1 Mathematical representation of policy

Policy is the agent's behavior, it is a mapping from states to actions. Given a state, a policy will output an action or a probability distribution over actions. A policy is defined as follows :

- Deterministic policy:

$$a = \pi(s)$$

- Stochastic policy:

$$\pi(a|s) = P(A_t = a|S_t = s)$$

3.2 The Discounted Returns

The total discounted reward for a state from time-step t:

$$G_t(s) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{\infty}^{k=0} \gamma^k R_{t+k+1}$$

Rewards are temporary. Even after picking an action that gives a decent reward, we might be missing on a greater total reward in the long-run. This long-term total reward is the Return. However, in practice, we consider discounted Returns.

3.3 Value Function

A value function is the long-term value of a state or an action. In other words, it's the expected Return over a state or an action. This is something that we are actually interested in optimizing.

3.3.1 The state value function:

$$V_\pi(s) = E_\pi(G_t(s)|S_t = s) = E_\pi(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots |S_t = s) = E_\pi(R_{t+1} + \gamma V_\pi(s_{t+1})|S_t = s)$$

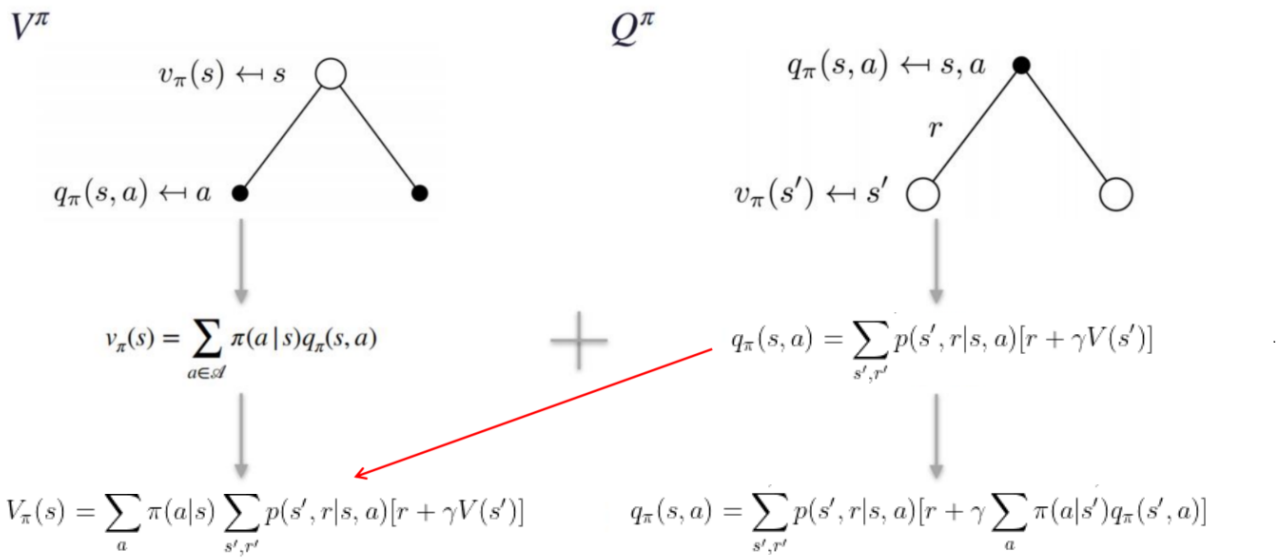
3.3.2 The Action Value Function

$$q_\pi(s,a) = E_\pi(G_t(s)|S_t = s, A_t = a) = E_\pi(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots |S_t = s, A_t = a) = E_\pi(R_{t+1} + \gamma$$

3.3.3 The relationship between the two value function

$$V_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) = \sum_a \pi(a|s) \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$$

$$q_\pi(s,a) = \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$$



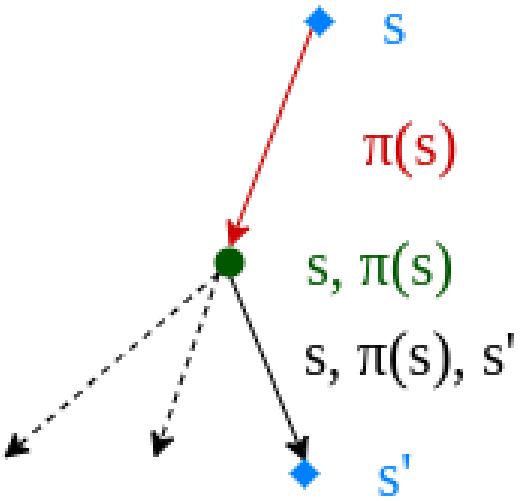
Given an MDP environment, we can use **dynamic programming algorithms** to compute optimal policies, which lead to the highest possible sum of future rewards at each state.

Dynamic programming algorithms work on the assumption that we have a perfect model of the environment's MDP. So, we're able to use a one-step look-ahead approach and compute rewards for all possible actions.

In this tutorial, we'll discuss how to find an optimal policy π for a given MDP. More specifically, **we'll learn about two dynamic programming algorithms: policy iteration and value iteration .**

4 Policy Iteration

In policy iteration, we start by choosing an arbitrary policy π. Then, we iteratively evaluate and improve the policy until convergence:



We evaluate a policy $\pi(s)$ by calculating the state value function $V(s)$:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r'} p(s',r|s,a)[r + \gamma V(s')]$$

Then, we calculate the improved policy by using one-step look-ahead to replace the initial policy $\pi(s)$:

$$\pi(s) = arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$$

Here, r is the reward generated by taking the action a, γ is a discount factor for future rewards and p is the transition probability.

In the beginning, we don't care about the initial policy π_0 being optimal or not. During the execution, we concentrate on improving it on every iteration by repeating policy evaluation and policy improvement steps. Using this algorithm we produce a chain of policies, where each policy is an improvement over the previous one: $\pi_0 \xrightarrow{E} v_{\pi 0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi 1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$

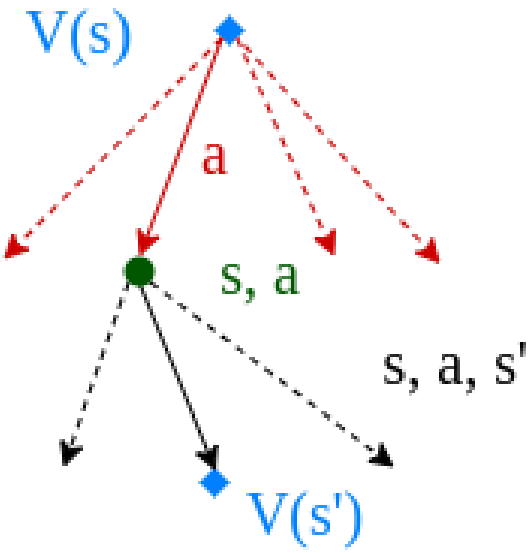
Algorithm 1: Policy Iteration Algorithm

Data: θ: a small number
Result: V: a value function s.t. $V \approx v_*$, π: a deterministic policy s.t. $\pi \approx \pi_*$
Function *PolicyIteration* **is**
  `/* Initialization */`
  Initialize $V(s)$ arbitrarily;
  Randomly initialize policy $\pi(s)$;
  `/* Policy Evaluation */`
  $\Delta \leftarrow 0$;
  while $\Delta < \theta$ **do**
    for *each* $s \in S$ **do**
      $v \leftarrow V(s)$;
      $V(s) \leftarrow \sum_{s',r'} p(s', r|s, \pi(s))[r + \gamma V(s')]$;
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
    end
  end
  `/* Policy Improvement */`
  policy-stable $\leftarrow$ *true*;
  for *each* $s \in S$ **do**
    old-action $\leftarrow \pi(s)$;
    $\pi(s) \leftarrow arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
    if *old-action* != $\pi(s)$ **then**
      policy-stable $\leftarrow$ *false*;
    end
  end
  if *policy − stable* **then**
    return $V \approx v_*$ and $\pi \approx \pi_*$;
  else
    go to Policy Evaluation;
  end
end

Since a finite MDP has a finite number of policies, the defined process is finite. In the end, converging an optimal policy π_* and an optimal value function v_* is guaranteed.

5 Value Iteration

In value iteration, we compute the optimal state value function by iteratively updating the estimate $\mathbf{v(s)}$:



The optimal state value function $V_*(s)$ is the maximum value function over all policies:

$$v_*(s) = \max_\pi v_\pi(s)$$

The optimal state value function $q_*(s, a)$ is the maximum action value function over all policies:

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

An optimal policy can be found by maximising over $q_*(s, a)$:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\ q_*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

We start with a random value function ![V(s)](E:\2022FALL\Machine-Learning-LAB\Lab13.Markov Decision Process\images\quicklatex.com-b04898eb2ba64775d4726571566b0f80_l3-166910049252419.svg). At each step, we update it:

$$V(s) = \max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$$

Hence, we look-ahead one step and go over all possible actions at each iteration to find the maximum:

Algorithm 2: Value Iteration Algorithm

Data: θ: a small number
Result: π: a deterministic policy s.t. $\pi \approx \pi_*$
Function *ValueIteration* **is**
  /* Initialization */
  Initialize $V(s)$ arbitrarily, except $V(terminal)$;
  $V(terminal) \leftarrow 0$;
  /* Loop until convergence */
  $\Delta \leftarrow 0$;
  while $\Delta < \theta$ **do**
    for *each* $s \in S$ **do**
      $v \leftarrow V(s)$;
      $V(s) \leftarrow \max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
    end
  end
  /* Return optimal policy */
  return π s.t. $\pi(s) = arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
end

The update step is very similar to the update step in the policy iteration algorithm. The only difference is that we take the maximum over all possible actions in the value iteration algorithm.

Instead of evaluating and then improving, the value iteration algorithm updates the state value function in a single step. This is possible by calculating all possible rewards by looking ahead.

The value iteration algorithm is guaranteed to converge to the optimal values.

6 Policy Iteration vs. Value Iteration

Policy iteration and value iteration are both dynamic programming algorithms that find an optimal policy π_* in a reinforcement learning environment. They both employ variations of Bellman updates and exploit one-step look-ahead:

Policy Iteration	Value Iteration
Starts with a random policy	Starts with a random value function
Algorithm is more complex	Algorithm is simpler
Guaranteed to converge	Guaranteed to converge
Cheaper to compute	More expensive to compute
Requires few iterations to converge	Requires more iterations to converge
Faster	Slower

In policy iteration, we start with a fixed policy. Conversely, in value iteration, we begin by selecting the value function. Then, in both algorithms, we iteratively improve until we reach convergence.

The policy iteration algorithm updates the policy. The value iteration algorithm iterates over the value function instead. Still, both algorithms implicitly update the policy and state value function in each iteration.

In each iteration, the policy iteration function goes through two phases. One phase evaluates the policy, and the other one improves it. The value iteration function covers these two phases by taking a maximum over the utility function for all possible actions.

The value iteration algorithm is straightforward. It combines two phases of the policy iteration into a single update operation. However, the value iteration function runs through all possible actions at once to find the maximum action value. Subsequently, the value iteration algorithm is computationally heavier.

Both algorithms are guaranteed to converge to an optimal policy in the end. Yet, the policy iteration algorithm converges within fewer iterations. As a result, the policy iteration is reported to conclude faster than the value iteration algorithm.

7 LAB Assignment

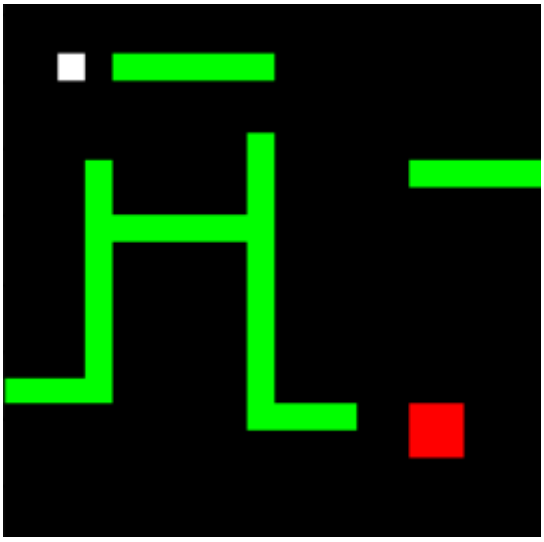
Please finish the **Exercise** and answer **Questions**.

7.1 Exercise (100 Points)

In this lab, we are about to find a shortest path with collision avoidance using MDP. We will model the problem as a MDP problem and use **value iteration** or **policy iteration** algorithm to solve it.

7.1.1 Environment Data

- Environment: `map_matrix.npy` has environment data. You need to use `numpy` to load it.
  - White block: an agent, for example, a robot
  - Red block: destination
  - Green block: obstacle



- $\mathscr{R}$ Reward: reward is implemented in code and it only concerns the next state:
  - wall: -1
  - destination: 0
  - else: -0.1
- $\mathscr{P}$ State transformation: next state is deterministic when taking an action under a certain state.
- π Initial policy: each direction (up, right, bottom, left) has equal probability.

7.1.2 Display

We have several methods for you to display policy, state value and path on map

```
def display_policy()
def display_v()
def display_path()
```

7.1.3 Implementation code

7.1.3.1 Initialize $\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}$

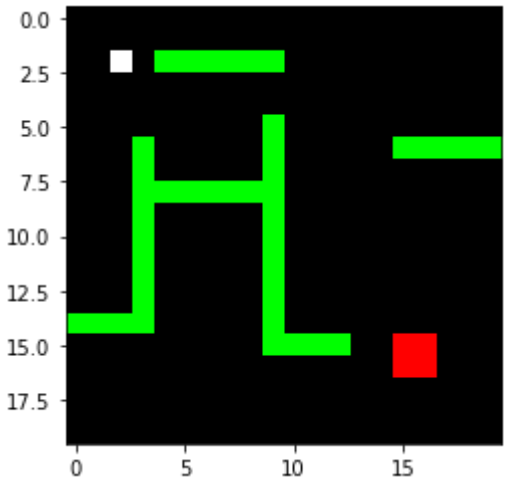
In [1]:

```
import numpy as np
import pandas as pd
import time
import matplotlib.pyplot as plt
```

In [2]:

```
map_matrix = np.load("map_matrix.npy")
plt.imshow(map_matrix)
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers).
```



In [3]:

```
MAX_Y = map_matrix.shape[0]
MAX_X = map_matrix.shape[1]
noS = MAX_Y * MAX_X
noA = 4
```

In [4]:

```
# A, defining action variables
UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3

# S, a sequence of number rather than coordinates
S = range(noS)

# reward
reward_blank = -0.1  #for every step -1 in other states
reward_wall = -1  # for wall
reward_goal = 0  # for destination
wall = []
goal = []
start = None
for y in range(MAX_Y):
    for x in range(MAX_X):
        if list(map_matrix[y][x]) == [0., 30., 0.]:
            wall.append(y * MAX_X + x)
        elif list(map_matrix[y][x]) == [255., 0., 0.]:
            goal.append(y * MAX_X + x)
        elif list(map_matrix[y][x]) == [255., 255., 255.]:
            start = (y, x)
print("--> State of walls:", wall)
print("--> State of destination:", goal)

# P[s][a] = (state transition probability, next state, reward, done)
P = dict()
grid = np.arange(noS).reshape(map_matrix.shape[0:2])
it = np.nditer(grid, flags=['multi_index'])

while not it.finished:
    s = it.iterindex
    y, x = it.multi_index
    P[s] = dict()
    #if (terminal_state(s)):
    if (s in goal):  #s in terminal_state:
        P[s][UP] = (1.0, s, reward_goal, True)
        P[s][RIGHT] = (1.0, s, reward_goal, True)
        P[s][DOWN] = (1.0, s, reward_goal, True)
        P[s][LEFT] = (1.0, s, reward_goal, True)
    else:
        ns_up = s if y == 0 else s - MAX_X
        ns_right = s if x == (MAX_X - 1) else s + 1
        ns_down = s if y == (MAX_Y - 1) else s + MAX_X
        ns_left = s if x == 0 else s - 1

        if s in goal:
            rw = reward_goal
        elif s in wall:
            rw = reward_wall
        else:
            rw = reward_blank

        P[s][UP] = (1.0, ns_up, rw, ns_up in goal)
        P[s][RIGHT] = (1.0, ns_right, rw, ns_right in goal)
        P[s][DOWN] = (1.0, ns_down, rw, ns_down in goal)
        P[s][LEFT] = (1.0, ns_left, rw, ns_left in goal)
```

```
    it.iternext()

print('--> Number of states in grid: ' + str(noS))
print('--> Number of action options in each state:' + str(noA))
print("--> Transition probabilities matrix(prob, next_state, reward, is_done): ")
print(P[0])
```

```
--> State of walls: [44, 45, 46, 47, 48, 49, 109, 123, 129, 135, 136, 137, 138, 139,
143, 149, 163, 164, 165, 166, 167, 168, 169, 183, 189, 203, 209, 223, 229, 243, 249,
263, 269, 280, 281, 282, 283, 289, 309, 310, 311, 312]
--> State of destination: [315, 316, 335, 336]
--> Number of states in grid: 400
--> Number of action options in each state:4
--> Transition probabilities matrix(prob, next_state, reward, is_done):
{0: (1.0, 0, -0.1, False), 1: (1.0, 1, -0.1, False), 2: (1.0, 20, -0.1, False), 3:
(1.0, 0, -0.1, False)}
```

7.1.3.2 Optimization Approach

You need to implement one of them

- Value Iteration

In [5]:

```
def value_iteration(P,theta=0.0001, discount_factor=1.0):
    """
    Value iteration

    :param theta: threshold to stop iteration
    :param discount_factor: the same as discount factor in formula
    :return: a policy and a state-value matrix V
    """



    V = np.zeros(noS)
    step = 0
    while True:
        # TODO: finish value iteration
        break

    # TODO: find the best policy according to V
    # Create a deterministic policy using the optimal value function
    # policy[s, a] means the probability of taking action a under state s, in this case, they a
    policy = np.zeros([noS, noA])
    pass

    return policy, V
```

- Policy Iteration

In [6]:

```
def policy_eval(policy_matrix, P_matrix, discount_factor=1.0, theta=0.00001):
    # Start with a random (all 0) value function
    V = np.zeros(noS)
    while True:
        delta = 0
        # TODO: finish evaluation part
        pass

        if delta < theta:
            break
    return np.array(V)


def policy_improvement(P_matrix, policy_eval_fn=policy_eval, discount_factor=1.0):
    def one_step_lookahead(state, V_matrix):
        A = np.zeros(noA)

        # TODO: calculate next states' values after taking different actions
        pass

        return A

    # Start with a random policy
    policy_now = np.ones([noS, noA]) / noA

    while True:
        # Evaluate the current policy
        V = policy_eval_fn(policy_now, P_matrix, discount_factor)

        # Will be set to false if we make any changes to the policy
        policy_stable = True

        # For each state...
        for s in range(noS):
            # TODO: The best action we would take under the current policy
            pass

            # TODO: Find the best action by one-step lookahead
            # Ties are resolved arbitarily
            pass

            # TODO: Greedily update the policy
            pass

        # If the policy is stable we've found an optimal policy. Return it
        if policy_stable:
            return policy_now, V
```

In [7]:

```
iteration = policy_improvement
policy, v = iteration(P)
```

7.1.4 Display

In [8]:

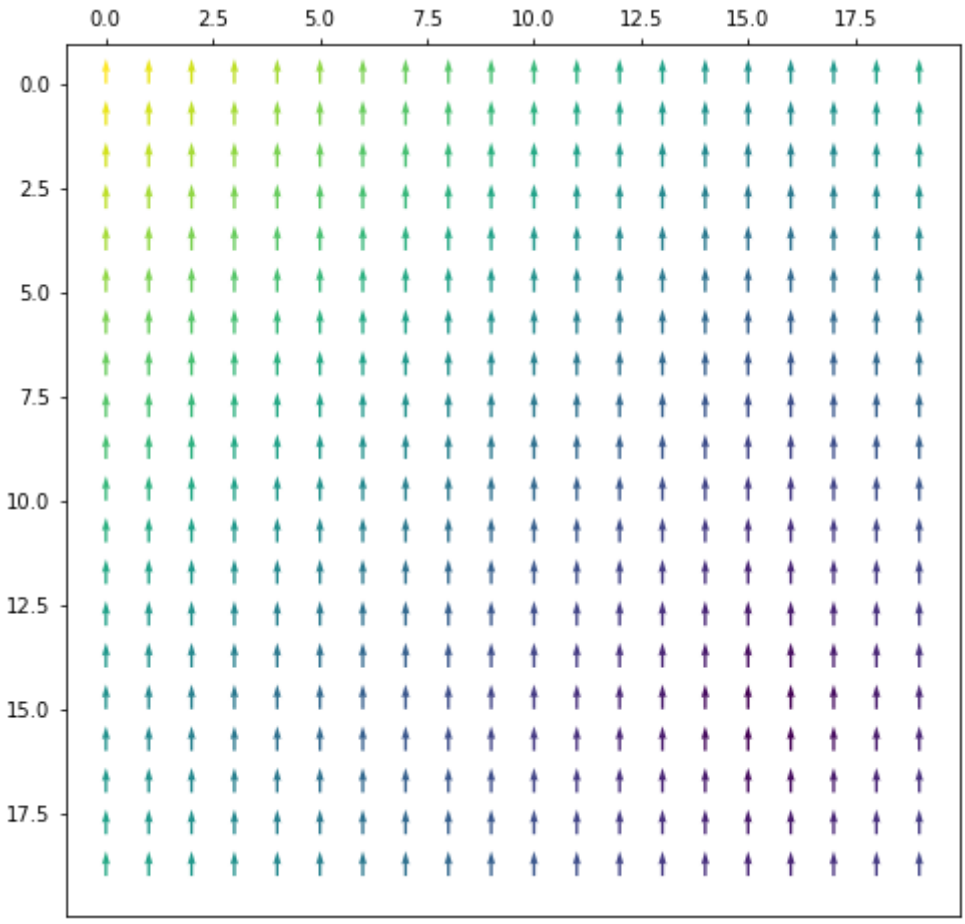
```
# display modules
def display_policy(policy_map: np.ndarray, des=((15, 16), (15, 15), (16, 15), (16, 16))):
    """
    Display policy in a vector field
    :param des: destination, for dying quiver
    :param policy_map: your policy map where each position is a number standing for an action
    :return:
    """

    def phrase_action(action: int):
        if action == 0:
            return 0, 1
        if action == 1:
            return 1, 0
        if action == 2:
            return 0, -1
        if action == 3:
            return -1, 0
        return 0, 0

    des = np.asarray(np.copy(des))

    X, Y = policy_map.shape
    grid_X = np.arange(X).reshape(-1, 1).repeat(Y, axis=1)
    grid_Y = np.arange(Y).reshape(1, -1).repeat(X, axis=0)
    direct_X, direct_Y = [], []
    C = []
    for x_row in list(zip(grid_X, grid_Y)):
        for p in tuple(zip(x_row[0], x_row[1])):
            direct_x, direct_y = phrase_action(policy_map[p[0], p[1]])
            direct_X.append(direct_x)
            direct_Y.append(direct_y)
            if des is not None:
                C.append(np.sum(np.abs(p - des)))
    fig = plt.figure(figsize=(8, 8))
    ax = plt.gca()
    ax.xaxis.set_ticks_position('top')  #将X坐标轴移到上面
    ax.invert_yaxis()  #反转Y坐标轴
    plt.quiver(grid_X, grid_Y, direct_X, direct_Y, 0 if len(C) == 0 else C)
    plt.show()


def display_v(V: np.ndarray, map_m: np.ndarray):
    """
    display state value in heat map
    :param map_m: 2D map matrix corresponding to v
    :param V: state value, 1D
    :return:
    """

    heatmap = V.reshape(map_m.shape[0:2])
    plt.imshow(heatmap, cmap=plt.cm.autumn)
    plt.colorbar()
    plt.show()


def display_path(policy_map, map_m, departure, des, max_iter=100):
    """

```

```
    :param policy_map: policy
    :param map_m: map
    :param departure: start point
    :param des: destination
    :return:
    """
    map_m = np.copy(map_m)

    now_y, now_x = departure
    iter_n = 0
    while now_y * MAX_X + now_x not in des:
        if iter_n > max_iter:
            break
        map_matrix[now_y, now_x, :] = np.array([151, 255, 255])
        if UP == policy_map[now_y, now_x]:
            now_y = now_y - 1
        elif RIGHT == policy_map[now_y, now_x]:
            now_x = now_x + 1
        elif DOWN == policy_map[now_y, now_x]:
            now_y = now_y + 1
        elif LEFT == policy_map[now_y, now_x]:
            now_x = now_x - 1
        now_y = (now_y + MAX_Y) % MAX_Y
        now_x = (now_x + MAX_X) % MAX_X
        map_m[now_y, now_x, :] = 120
        plt.imshow(map_m)
        plt.show()
        display.clear_output(wait=True)

        iter_n += 1
```

In [9]:

```
print("Grid Policy (0=up, 1=right, 2=down, 3=left):")
grid_policy = np.reshape(np.argmax(policy, axis=1), map_matrix.shape[0:2])
display_policy(grid_policy.T)
```

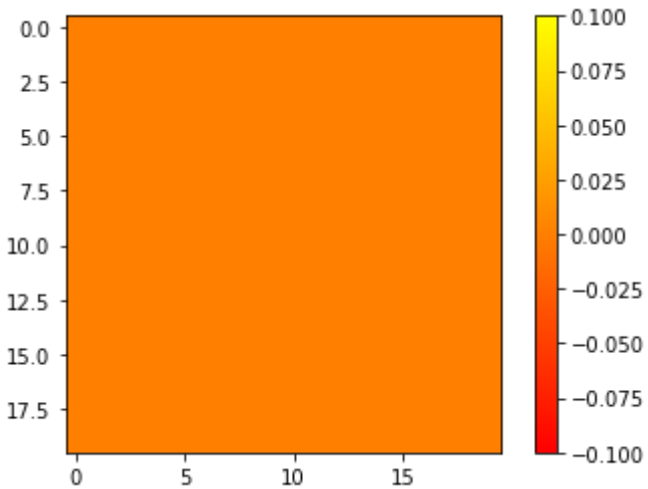
```
Grid Policy (0=up, 1=right, 2=down, 3=left):
```



In [10]:

```
print("State value:")
display_v(v, map_matrix)
```

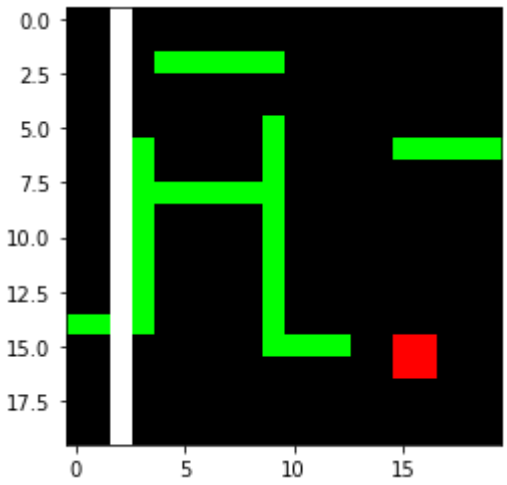
```
State value:
```



In [11]:

```
%matplotlib inline
display_path(grid_policy, map_matrix, start, goal)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats o
r [0..255] for integers).
```



In [11]:

```

```