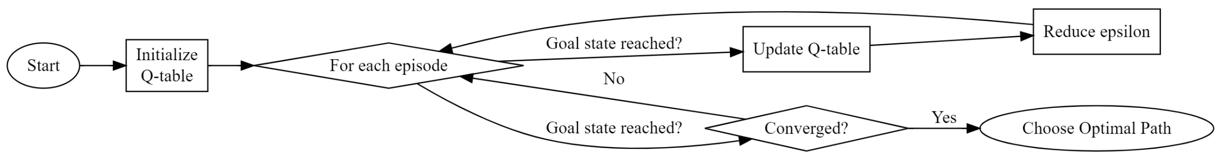
DRL HW01_GridWrld

▼ 1. (15%) Gnerate Q-learning algorithm that solve the grid world flowchart (prompt, dot file, and result 截圖)



💡 prompt：Gnerate Q-learning steps that solve the grid world flowchart

Sure, here are the steps for solving the grid world flowchart using Q-learning:

1. Define the state space, action space, and reward function of the grid world. In this case, the state space consists of the locations on the grid, the action space consists of moving up, down, left, or right, and the reward function gives a positive reward of 100 for reaching the goal state and a negative reward of -1 for each time step taken.
2. Initialize the Q-table with zeros or random values.
3. Choose a learning rate (alpha), discount factor (gamma), and exploration rate (epsilon). Alpha determines the weight given to new information, gamma determines the importance of future rewards, and epsilon determines the probability of taking a random action.
4. For each episode, initialize the state to a random location on the grid.
5. Repeat until the goal state is reached:
   a. Use the Q-table to choose an action, either by selecting the highest Q-value for the current state or by taking a random action with probability epsilon.
   b. Execute the action and observe the resulting state and reward.
   c. Update the Q-value for the current state and action using the Q-learning update rule:
   Q(s, a) = (1 - alpha) * Q(s, a) + alpha * (r + gamma * max(Q(s', a')))
   where s is the current state, a is the chosen action, r is the observed reward, s' is the resulting state, and max(Q(s', a')) is the maximum Q-value for the next state.
   d. Set the current state to the resulting state.
6. Reduce the value of epsilon over time to gradually shift from exploration to exploitation.
7. Repeat the process for a sufficient number of episodes until the Q-table has converged.
8. Once the Q-table has converged, use it to choose the optimal path from any starting location to the goal state by selecting the action with the highest Q-value for each state encountered until the goal is reached.

💡 prompt：give me above steps to flowchart with dot file

Here is an example of how the Q-learning steps for solving the grid world flowchart could be represented in a flowchart using the DOT language:

```
digraph Q_learning_flowchart {
    rankdir=LR;

    // Define nodes
    start [shape=ellipse, label="Start"];
```

```
    initialize [shape=rectangle, label="Initialize\nQ-table"];
    episode [shape=diamond, label="For each episode"];
    update_Q [shape=rectangle, label="Update Q-table"];
    reduce_epsilon [shape=rectangle, label="Reduce epsilon"];
    converged [shape=diamond, label="Converged?"];
    optimal_path [shape=ellipse, label="Choose Optimal Path"];

    // Define edges
    start -> initialize;
    initialize -> episode;
    episode -> update_Q [label="Goal state reached?"];
    update_Q -> reduce_epsilon;
    reduce_epsilon -> episode;
    episode -> converged [label="Goal state reached?"];
    converged -> optimal_path [label="Yes"];
    converged -> episode [label="No"];
}
```

This code creates a directed graph (digraph) with nodes representing the different steps of the Q-learning algorithm and edges representing the flow of control between these steps. The `rankdir=LR` statement sets the orientation of the graph from left to right.

The `start` node represents the beginning of the algorithm, and the `initialize` node initializes the Q-table. The `episode` node represents the loop that runs for each episode, and the `update_Q` node updates the Q-table based on the observed reward and the next state. The `reduce_epsilon` node gradually reduces the exploration rate over time. The `converged` node checks whether the Q-table has converged, and the `optimal_path` node selects the optimal path using the converged Q-table.

Note that the labels for each node can be modified to include more detailed information about each step if desired.

▼ 2. (15%) Generate Q-learnig pseudo algorithm in latex format (prompt, latex and result)

Algorithm 1 q-learning Algorithm

Initialize Q-table Q(s, a) arbitrarily for all s in S and a in A(s)
repeat
    Initialize state s
    repeat
        Choose action a using epsilon-greedy policy based on Q(s, a)
        Take action a and observe reward r and next state s'
        Update Q(s, a) using Q-learning update rule:
        $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$
        Set s = s'
    until goal state is reached
    Gradually reduce epsilon
    if Q-table has converged **then**
        break
    end if
until convergence criterion is met
Choose optimal path using converged Q-table

💡 prompt：Gnerate above steps Q-learnig pseudo algorithm that solve the grid world

Certainly! Here is a pseudo-code implementation of the Q-learning algorithm for solving the grid world:

```
Initialize Q-table Q(s, a) arbitrarily for all s in S and a in A(s)

Repeat for each episode:
    Initialize state s
    Repeat until goal state is reached:
```

```
        Choose action a using epsilon-greedy policy based on Q(s, a)
        Take action a and observe reward r and next state s'
        Update Q(s, a) using Q-learning update rule:
            Q(s, a) = (1 - alpha) * Q(s, a) + alpha * (r + gamma * max(Q(s', a')))
        Set s = s'
    Gradually reduce epsilon
    If Q-table has converged, break out of loop

Choose optimal path using converged Q-table
```

In this algorithm, `S` is the state space, `A(s)` is the set of actions available in state `s`, `alpha` is the learning rate, `gamma` is the discount factor, and `epsilon` is the exploration rate. The `max` function returns the maximum Q-value for the next state `s'` over all possible actions `a'`.

Note that the implementation of the `epsilon-greedy` policy is not shown here, but typically involves choosing a random action with probability `epsilon` and the action with the highest Q-value for the current state with probability `1 - epsilon`.

> 💡 prompt：give me above pseudo-code with latex format

Sure, here's the same Q-learning algorithm in pseudo-code format using LaTeX:

```
\begin{algorithm}
\begin{algorithmic}
    \caption{ q-learning Algorithm }
    \State Initialize Q-table Q(s, a) arbitrarily for all s in S and a in A(s)
    \Repeat
        \State Initialize state s
        \Repeat
            \State Choose action a using epsilon-greedy policy based on Q(s, a)
            \State Take action a and observe reward r and next state s'
            \State Update Q(s, a) using Q-learning update rule:
            \State $Q(s, a) \gets (1 - \alpha) Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a'))$
            \State Set s = s'
        \Until goal state is reached
        \State Gradually reduce epsilon
        \If{Q-table has converged}
            \State \textbf{break}
        \EndIf
    \Until convergence criterion is met
    \State Choose optimal path using converged Q-table
\end{algorithmic}
\end{algorithm}
```

This code uses the `algorithmic` environment from the `algorithmicx` package to define the pseudo-code, and uses a combination of `State` and `\State` commands to define the different steps of the algorithm. The `Until` and `If` tatements are written using capitalized keywords, and the `break` statement is written in boldface. The `max` function is defined using a subscript and braces, and the `gets` command is used to denote assignment.

▼ 3. (20%) Geneate the environment of the n x n grid world in flask web

▼ STEP1 Flask web(1.app.py 2.index.html take n from user 3. squre.html)

> 💡 Prompt: write a flask web that takes n as user input n=3 to 10, and generate n x n square

Sure! Here's an example Flask web application that takes user input `n` between 3 and 10, and generates an `n` x `n` square using HTML and CSS:

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        n = int(request.form['n'])
        if n < 3 or n > 10:
            return render_template('index.html', error='n must be between 3 and 10')
        return render_template('square.html', n=n)
    else:
        return render_template('index.html')

if __name__ == '__main__':
    app.run(debug=True)
```

This code sets up a Flask application with two routes: one for the homepage (`'/'`) and one for the square page (`'/square'`). The homepage uses a form to get user input for `n`, and validates that `n` is between 3 and 10. If the input is valid, the homepage redirects to the square page and passes `n` as a parameter. The square page uses HTML and CSS to generate an `n` x `n` square.

Here's the HTML and CSS for the square page (`square.html`):

```
<!DOCTYPE html>
<html>
<head>
  <title>Square</title>
  <style>
    .row {
      display: flex;
      flex-direction: row;
    }
    .cell {
      width: 50px;
      height: 50px;
      border: 1px solid black;
      box-sizing: border-box;
      text-align: center;
      font-size: 20px;
      font-weight: bold;
      line-height: 50px;
    }
  </style>
</head>
<body>
  {% for i in range(n) %}
    <div class="row">
      {% for j in range(n) %}
        <div class="cell">{{ i*n + j + 1 }}</div>
      {% endfor %}
    </div>
  {% endfor %}
</body>
</html>
```

This code uses a loop to generate an `n` x `n` square. Each row is a `div` with the class `row`, and each cell is a `div` with the class `cell`. The cell content is the index of the cell in the grid (`i*n + j + 1`). The CSS styles the cells with a fixed width and height, a border, and centers the content.

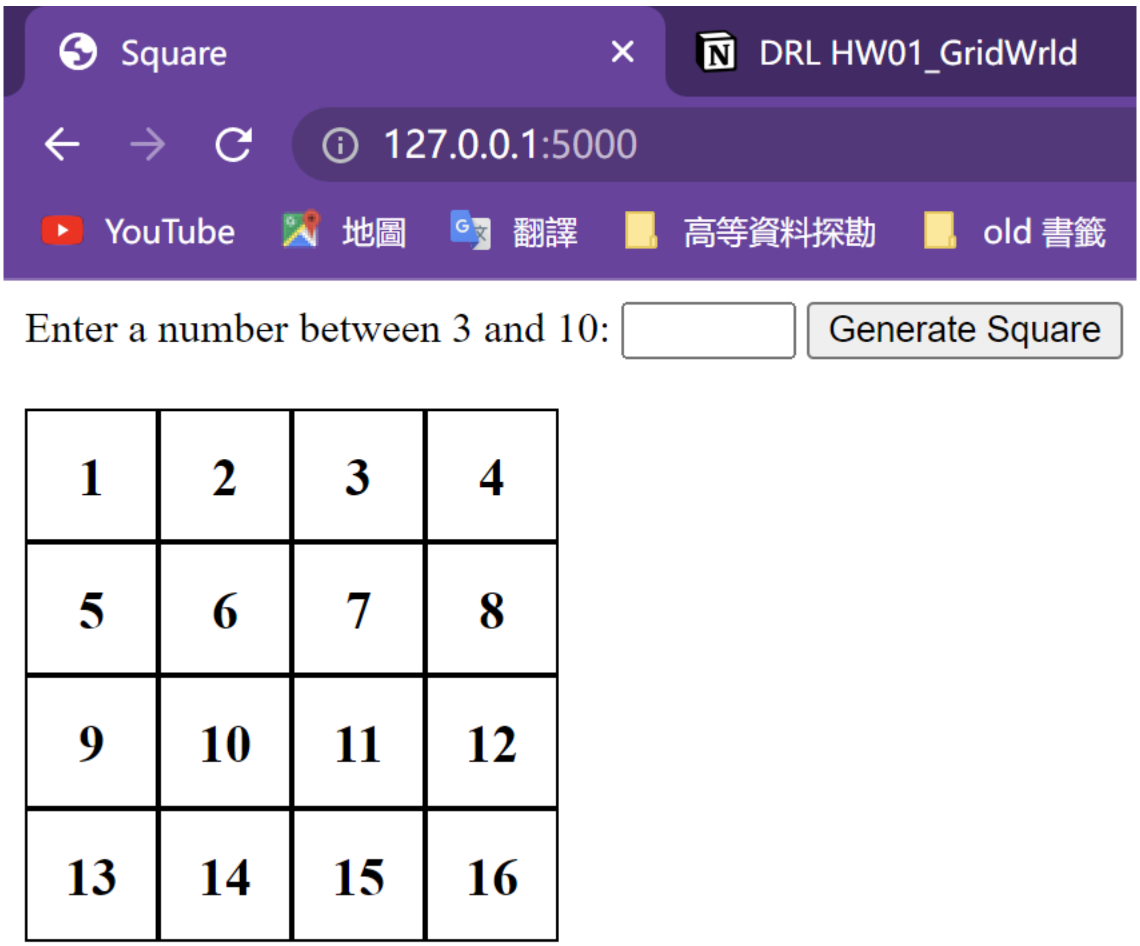
Here's the HTML for the homepage (`index.html`):

```
<!DOCTYPE html>
<html>
<head>
  <title>Square Generator</title>
</head>
<body>
  {% if error %}
    <p>{{ error }}</p>
  {% endif %}
```

```
    <form action="/" method="post">
      <label for="n">Enter a number between 3 and 10:</label>
      <input type="number" name="n" id="n" min="3" max="10" required>
      <input type="submit" value="Generate Square">
    </form>
  </body>
</html>
```

This code displays a form with an input field for `n`. If there is an error (i.e., `n` is outside the range of 3 to 10), the error message is displayed. When the user submits the form, the application redirects to the square page with the user input as a parameter.

I hope this helps! Let me know if you have any questions.



▼ STEP2. Impove square.html

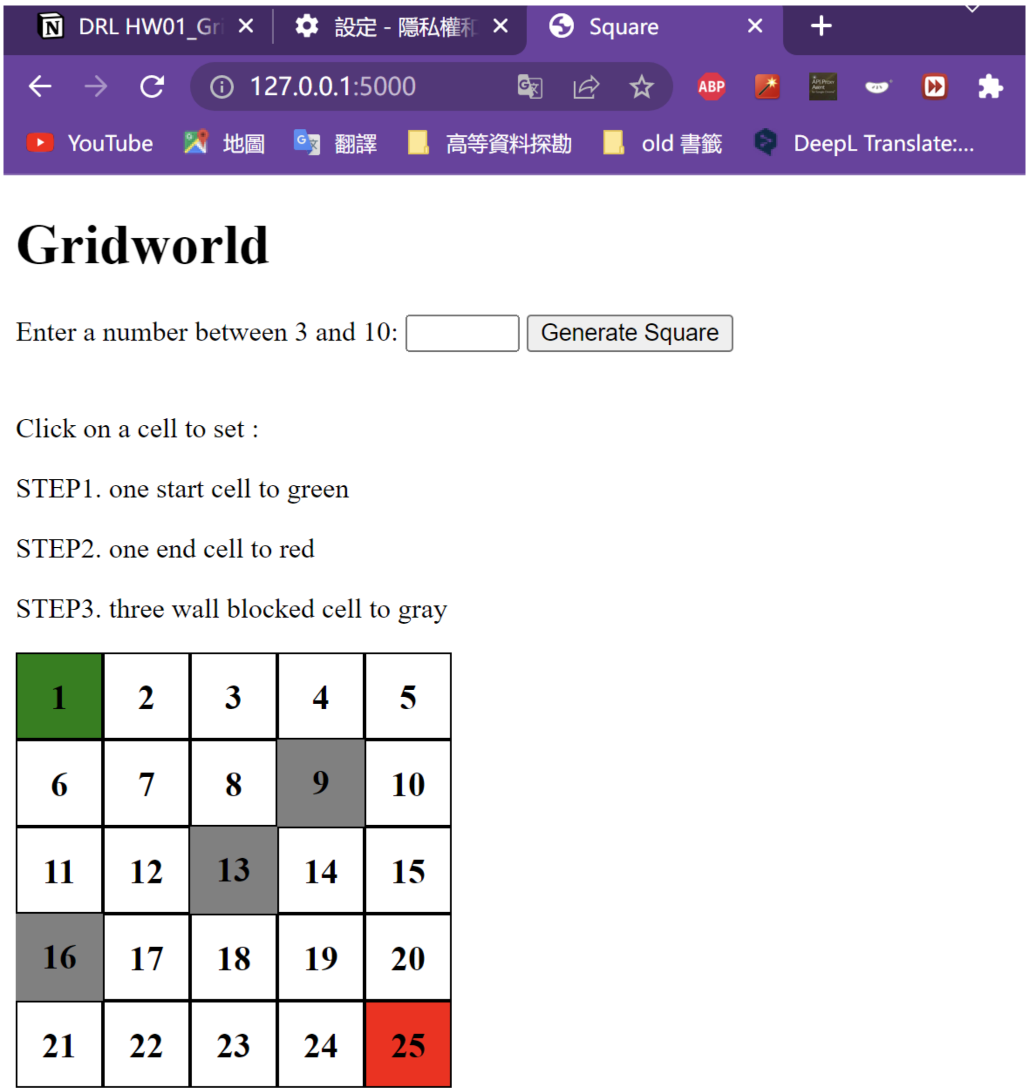
💡 prompt: rewrite the square.html, make a gridworld that use jquery and flex. allow user to setup start grid and end grid by clicking the cell, make them green and red respectively. Make 3 blocks in grey that cannot pass.

Sure, here's an updated version of the `square.html` file that creates a grid world using jQuery and flexbox. The user can set the start and end grids by clicking on cells, which will turn green and red respectively. There are also three blocks that cannot be passed, which are shown in gray:

```
<!DOCTYPE html>
<html>
<head>
  <title>Grid World</title>
  <style>
    .grid {
      display: flex;
      flex-wrap: wrap;
      width: 500px;
      height: 500px;
      margin: 0 auto;
      border: 1px solid black;
      box-sizing: border-box;
    }
    .cell {
      flex: 1 0 calc(10% - 2px);
      height: calc(10% - 2px);
      margin: 1px;
      box-sizing: border-box;
      background-color: white;
      border: 1px solid gray;
    }
    .cell.start {
      background-color: green;
    }
    .cell.end {
      background-color: red;
    }
    .cell.blocked {
      background-color: gray;
      border: none;
    }
  </style>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script>
    $(document).ready(function() {
      // Create grid world
      var n = 10;
      var grid = [];
      for (var i = 0; i < n; i++) {
        grid[i] = [];
        for (var j = 0; j < n; j++) {
          grid[i][j] = 0;
          if ((i == 2 && j == 2) || (i == 3 && j == 2) || (i == 4 && j == 2)) {
            // Set blocked cells
            $('#cell_' + i + '_' + j).addClass('blocked');
          }
          $('#cell_' + i + '_' + j).click(function() {
            var id = $(this).attr('id');
            var i = parseInt(id.split('_')[1]);
            var j = parseInt(id.split('_')[2]);
            if ($('#cell_' + i + '_' + j).hasClass('start')) {
              // Clear start cell
              $('#cell_' + i + '_' + j).removeClass('start');
            } else if ($('#cell_' + i + '_' + j).hasClass('end')) {
              // Clear end cell
              $('#cell_' + i + '_' + j).removeClass('end');
            } else {
              // Set start or end cell
              if ($('.start').length == 0) {
                $('#cell_' + i + '_' + j).addClass('start');
              } else if ($('.end').length == 0) {
                $('#cell_' + i + '_' + j).addClass('end');
              }
            }
          });
        }
      }
    });
  </script>
</head>
<body>
  <div class="grid">
    {% for i in range(n) %}
```

```
        {% for j in range(n) %}
          <div class="cell" id="cell_{{ i }}_{{ j }}"></div>
        {% endfor %}
      {% endfor %}
    </div>
  </body>
  </html>
```

The HTML and CSS are similar to before, but with a few changes. The grid world is now a flex container (`display: flex;`), with each cell having a `flex-basis` of `10%.`



▼ STEP3. add q-learning

💡 prompt: give me app.py that implement the q-learning to solve the gridworld problem shtat shows in the index.html and square.html. Separate the app.py into three parts.

▼ 4.(20%) Generate the python code

套用 https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/tree/master/contents/2_Q_Learning_maze 去修改

▼ envirnment

```
"""
Reinforcement learning maze example.

Green rectangle:        explorer.
Black rectangles:       hells       [reward = -1].
Red rectangle:          paradise    [reward = +1].
All other states:       ground      [reward = 0].

This script is the environment part of this example. The RL is in RL_brain.py.

View more on my tutorial page: https://morvanzhou.github.io/tutorials/
"""


import numpy as np
import time
import sys
import tkinter as tk

# if sys.version_info.major == 2:
#     import Tkinter as tk
# else:
#     import tkinter as tk


UNIT = 40   # pixels
# MAZE_H = 4  # grid height
# MAZE_W = 4  # grid width


class Maze(tk.Tk, object):
    def __init__(self,n=3,startX=0,startY=0,endX=2,endY=2,hellX_1=0,hellY_1=1,hellX_2=0,hellY_2=2,hellX_3=1,hellY_3=2):
        super(Maze, self).__init__()
        self.action_space = ['u', 'd', 'l', 'r']
        self.n_actions = len(self.action_space)
        self.title('maze')
        self.MAZE_H=n
        self.MAZE_W=n
        self.start_X=startX
        self.start_Y=startY
        self.end_X=endX
        self.end_Y=endY
        self.hell_x1=hellX_1
        self.hell_y1=hellY_1
        self.hell_x2=hellX_2
        self.hell_y2=hellY_2
        self.hell_x3=hellX_3
        self.hell_y3=hellY_3
        self.geometry('{0}x{1}'.format(self.MAZE_W * UNIT, self.MAZE_H * UNIT))
        self._build_maze()

    def _build_maze(self):
        self.canvas = tk.Canvas(self, bg='white',
                           height=self.MAZE_H * UNIT,
                           width=self.MAZE_W * UNIT)

        # create grids
        for c in range(0, self.MAZE_W * UNIT, UNIT):
            x0, y0, x1, y1 = c, 0, c, self.MAZE_H * UNIT
            self.canvas.create_line(x0, y0, x1, y1)
        for r in range(0, self.MAZE_H * UNIT, UNIT):
            x0, y0, x1, y1 = 0, r, self.MAZE_W * UNIT, r
            self.canvas.create_line(x0, y0, x1, y1)
```

```
        # create origin
        origin = np.array([20, 20])

        # hell
        hell1_center = origin + np.array([UNIT * self.hell_y1, UNIT * self.hell_x1])
        self.hell1 = self.canvas.create_rectangle(
            hell1_center[0] - 15, hell1_center[1] - 15,
            hell1_center[0] + 15, hell1_center[1] + 15,
            fill='black')
        # hell
        hell2_center = origin + np.array([UNIT * self.hell_y2, UNIT * self.hell_x2])
        self.hell2 = self.canvas.create_rectangle(
            hell2_center[0] - 15, hell2_center[1] - 15,
            hell2_center[0] + 15, hell2_center[1] + 15,
            fill='black')

        # hell
        hell3_center = origin + np.array([UNIT * self.hell_y3, UNIT * self.hell_x3])
        self.hell3 = self.canvas.create_rectangle(
            hell3_center[0] - 15, hell3_center[1] - 15,
            hell3_center[0] + 15, hell3_center[1] + 15,
            fill='black')

        # create red rect as end Position
        end_center = origin + np.array([UNIT * self.end_Y, UNIT * self.end_X])
        self.oval = self.canvas.create_rectangle(
            end_center[0] - 15, end_center[1] - 15,
            end_center[0] + 15, end_center[1] + 15,
            fill='red')

        # create green rect as start Position
        start_center = origin + np.array([UNIT * self.start_Y, UNIT * self.start_X])
        self.rect = self.canvas.create_rectangle(
            start_center[0] - 15, start_center[1] - 15,
            start_center[0] + 15, start_center[1] + 15,
            fill='green')

        # pack all
        self.canvas.pack()

    def reset(self):
        self.update()
        time.sleep(0.5)
        self.canvas.delete(self.rect)
        origin = np.array([20, 20])
        # create green rect as start Position
        start_center = origin + np.array([UNIT * self.start_Y, UNIT * self.start_X])
        self.rect = self.canvas.create_rectangle(
            start_center[0] - 15, start_center[1] - 15,
            start_center[0] + 15, start_center[1] + 15,
            fill='green')
        # return observation
        return self.canvas.coords(self.rect)

    def step(self, action):
        s = self.canvas.coords(self.rect)
        base_action = np.array([0, 0])
        if action == 0:   # up
            if s[1] > UNIT:
                base_action[1] -= UNIT
        elif action == 1:   # down
            if s[1] < (self.MAZE_H - 1) * UNIT:
                base_action[1] += UNIT
        elif action == 2:   # right
            if s[0] < (self.MAZE_W - 1) * UNIT:
                base_action[0] += UNIT
        elif action == 3:   # left
            if s[0] > UNIT:
                base_action[0] -= UNIT

        self.canvas.move(self.rect, base_action[0], base_action[1])  # move agent

        s_ = self.canvas.coords(self.rect)  # next state

        # reward function
        if s_ == self.canvas.coords(self.oval):
            reward = 1
            done = True
            s_ = 'terminal'
        elif s_ in [self.canvas.coords(self.hell1), self.canvas.coords(self.hell2),self.canvas.coords(self.hell3)]:
```

```
            reward = -1
            done = True
            s_ = 'terminal'
        else:
            reward = 0
            done = False

        return s_, reward, done

    def render(self):
        time.sleep(0.1)
        self.update()


def update():
    for t in range(10):
        s = env.reset()
        while True:
            env.render()
            a = 1
            s, r, done = env.step(a)
            if done:
                break

if __name__ == '__main__':
    env = Maze()
    env.after(100, update)
    env.mainloop()
```

▼ RL-qleaning

```
"""
Reinforcement learning maze example.

Red rectangle:          explorer.
Black rectangles:       hells       [reward = -1].
Yellow bin circle:      paradise    [reward = +1].
All other states:       ground      [reward = 0].

This script is the main part which controls the update method of this example.
The RL is in RL_brain.py.

View more on my tutorial page: https://morvanzhou.github.io/tutorials/
"""

from maze_env import Maze
from RL_brain import QLearningTable
import argparse
import sys

def update():
    for episode in range(100):
        # initial observation
        observation = env.reset()

        while True:
            # fresh env
            env.render()

            # RL choose action based on observation
            action = RL.choose_action(str(observation))

            # RL take action and get next observation and reward
            observation_, reward, done = env.step(action)

            # RL learn from this transition
            RL.learn(str(observation), action, reward, str(observation_))

            # swap observation
            observation = observation_

            # break while loop when end of this episode
            if done:
                break

    # end of game
    print('game over')
```

```
    env.destroy()

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description='Set gridword inital parameter')
    parser.add_argument('--n',
                        type=int,
                        help='set numbers of grid',
                        required=True)
    parser.add_argument('--startX',
                        type=int,
                        help='set startX',
                        )
    parser.add_argument('--startY',
                        type=int,
                        help='set startY',
                        )
    parser.add_argument('--endX',
                        type=int,
                        help='set endX',
                        )
    parser.add_argument('--endY',
                        type=int,
                        help='set endY',
                        )
    parser.add_argument('--blockX_1',
                        type=int,
                        help='set blockX_1',
                        )
    parser.add_argument('--blockY_1',
                        type=int,
                        help='set blockY_1',
                        )
    parser.add_argument('--blockX_2',
                        type=int,
                        help='set blockX_2',
                        )
    parser.add_argument('--blockY_2',
                        type=int,
                        help='set blockY_2',
                        )
    parser.add_argument('--blockX_3',
                        type=int,
                        help='set blockX_3',
                        )
    parser.add_argument('--blockY_3',
                        type=int,
                        help='set blockY_3',
                        )
    args = parser.parse_args()
    env = Maze(args.n,args.startX,args.startY,args.endX,args.endY,args.blockX_1,args.blockY_1,args.blockX_2,args.blockY_2,args.block
    RL = QLearningTable(actions=list(range(env.n_actions)))
    print(list(range(env.n_actions)))
    # print(RL)

    env.after(100, update)
    env.mainloop()
```

▼ run game

```
"""
This part of code is the Q learning brain, which is a brain of the agent.
All decisions are made in here.

View more on my tutorial page: https://morvanzhou.github.io/tutorials/
"""

import numpy as np
import pandas as pd


class QLearningTable:
    def __init__(self, actions, learning_rate=0.01, reward_decay=0.9, e_greedy=0.9):
        self.actions = actions  # a list
        self.lr = learning_rate
        self.gamma = reward_decay
        self.epsilon = e_greedy
```

```
        self.q_table = pd.DataFrame(columns=self.actions, dtype=np.float64)

    def choose_action(self, observation):
        self.check_state_exist(observation)
        # action selection
        if np.random.uniform() < self.epsilon:
            # choose best action
            state_action = self.q_table.loc[observation, :]
            # some actions may have the same value, randomly choose on in these actions
            action = np.random.choice(state_action[state_action == np.max(state_action)].index)
        else:
            # choose random action
            action = np.random.choice(self.actions)
        return action

    def learn(self, s, a, r, s_):
        self.check_state_exist(s_)
        q_predict = self.q_table.loc[s, a]
        if s_ != 'terminal':
            q_target = r + self.gamma * self.q_table.loc[s_, :].max()  # next state is not terminal
        else:
            q_target = r  # next state is terminal
        self.q_table.loc[s, a] += self.lr * (q_target - q_predict)  # update

    def check_state_exist(self, state):
        if state not in self.q_table.index:
            # append new state to q table
            self.q_table = self.q_table.append(
                pd.Series(
                    [0]*len(self.actions),
                    index=self.q_table.columns,
                    name=state,
                )
            )
```

▼ 5.(30%) Put everything together

▼ update flask app.py

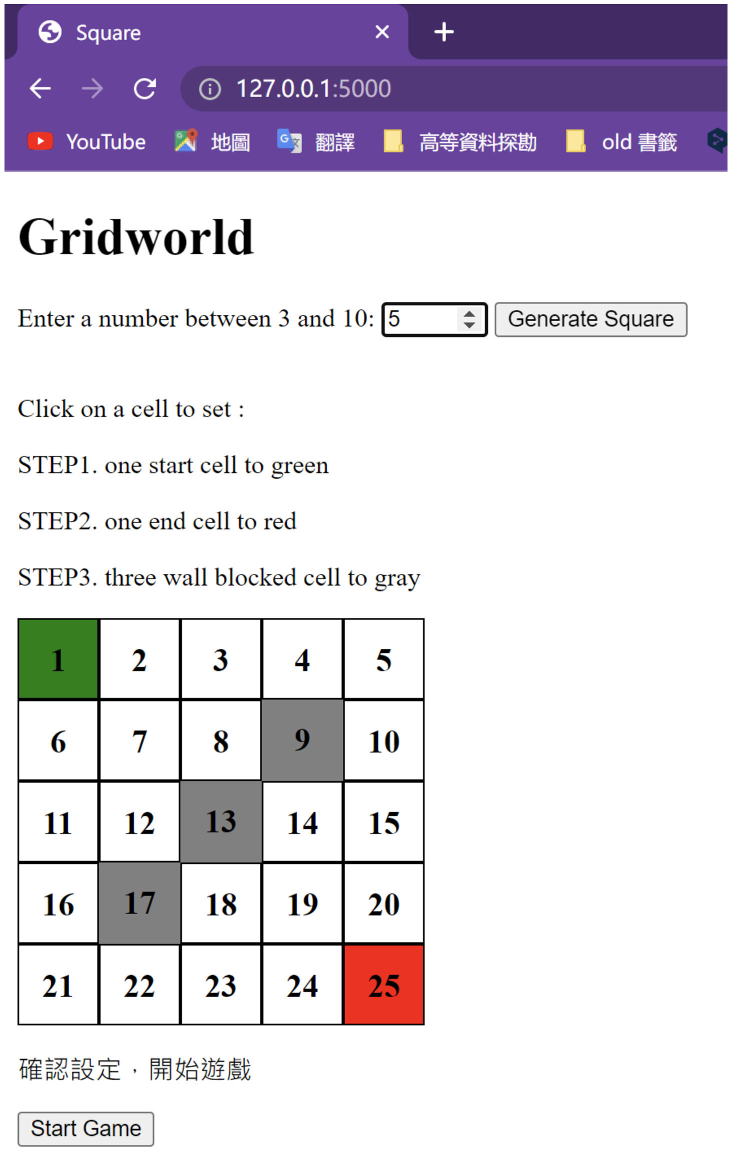
```
@app.route('/startGame', methods=['GET', 'POST'])
def startGame():
    # Initialize environment
    if request.method == 'POST':
        n = int(request.form['n'])
        startPos = str(request.form['startPos'])
        startX = startPos.split(",", 1)[0]
        startY = startPos.split(",", 1)[1]
        endPos = str(request.form['endPos'])
        endX = endPos.split(",", 1)[0]
        endY = endPos.split(",", 1)[1]
        blockPos = str(request.form['blockPos'])
        # blockX=[blockPos[0],blockPos[4],blockPos[8]]
        # blockY=[blockPos[0],blockPos[4],blockPos[8]]
        blockX_1 = blockPos[0]
        blockY_1 = blockPos[2]
        blockX_2 = blockPos[4]
        blockY_2 = blockPos[6]
        blockX_3 = blockPos[8]
        blockY_3 = blockPos[10]


    env = GridWorld(n=n)
    p = subprocess.run(
        [
            'python', 'run_this.py',
            '--n',str(n),
            '--startX',startX,
            '--startY',startY,
            '--endX',endX,
            '--endY',endY,
            '--blockX_1',blockX_1,
            '--blockY_1',blockY_1,
            '--blockX_2',blockX_2,
            '--blockY_2',blockY_2,
            '--blockX_3',blockX_3,
```

```
            '--blockY_3',blockY_3,

        ]
    )
    return render_template('square.html', n=n)
```

▼ update flask square.html




```
<!DOCTYPE html>
<html>
<head>
  <title>Square</title>
  <style>
      .grid {
      display: flex;
      flex-wrap: wrap;
      width: 500px;
      height: 500px;
      margin: 0 auto;
      border: 1px solid black;
      box-sizing: border-box;
```

```
    }
    .row {
      display: flex;
      flex-direction: row;
    }
    .cell {
      width: 50px;
      height: 50px;
      border: 1px solid black;
      box-sizing: border-box;
      text-align: center;
      font-size: 20px;
      font-weight: bold;
      line-height: 50px;
    }
      .cell.start {
      background-color: green;
    }
    .cell.end {
      background-color: red;
    }
    .cell.blocked {
      background-color: gray;
      border: none;
    }
  </style>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script>
    $(document).ready(function() {

    var start = null;
    var end = null;
    var blocked = [];

    $('.cell').click(function() {

    var id = $(this).attr('id');
    var row = parseInt(id.split('_')[1]);
    var col = parseInt(id.split('_')[2]);

    if ($(this).hasClass('blocked')) {
    $(this).removeClass('blocked');
    blocked.splice(blocked.indexOf([row, col]), 1);
    } else if ($(this).hasClass('start')) {
    $(this).removeClass('start');
    start = null;
    }else if ($(this).hasClass('end')) {
    $(this).removeClass('end');
    end = null;
    }else if ($('.start').length == 0) {
    $(this).addClass('start');
    start = [row, col];
    $('#startPos').val(start);
    } else if ($('.end').length == 0) {
    $(this).addClass('end');
    end = [row, col];
    $('#endPos').val(end);
    } else if($('.blocked').length < 3){
    $(this).addClass('blocked');
    blocked.push([row, col]);
    $('#blockPos').val(blocked);
    }
  });

  });

</head>
<body>
  <h1>Gridworld</h1>

  <form action="/" method="post">
    <label for="n">Enter a number between 3 and 10:</label>
    <input type="number" name="n" id="n" min="3" max="10" required>
    <input type="submit" value="Generate Square">
  </form>
  <br>
  <p>Click on a cell to set : </p>
  <p>STEP1. one start cell to green</p>
  <p>STEP2. one end cell to red</p>
  <p>STEP3. three wall blocked cell to gray</p>
```

```
  {% for i in range(n) %}
    <div class="row">
      {% for j in range(n) %}
        <div class="cell" id="cell_{{ i }}_{{ j }}">{{ i*n + j + 1 }}</div>
      {% endfor %}
    </div>
  {% endfor %}

  <form action="/startGame" method="post">
    <p>確認設定，開始遊戲</p>
    <input type="number" name="n" id="n" min="3" max="10" value="{{n}}" hidden >
    <input type="text" name="startPos" id="startPos" hidden>
    <input type="text" name="endPos" id="endPos"  hidden>
    <input type="text" name="blockPos" id="blockPos" hidden>
    <input type="submit" value="Start Game">
  </form>

</body>
</html>
```

結果展示

https://youtu.be/gaG7Jk4-8jA