Machine Learning - Final Exam

Tintrim Dwi Ary Widhianingsih
20185116

December 20, 2018

Problem 1 Reinforcement Learning using Monte Carlo.

In this section, we try to run the Reinforcement Learning (RL) program via Monte Carlo using python. There are two program files in this method: `environtment.py` and `mc_agent.py`. The codes are available in `https://github.com/rlcode/reinforcement-learning/tree/master/1-grid-world/3-monte-carlo`. This code is built for solving a simple grid world game, where the agent should find the path to go to the goal with avoiding the obstacles that are fixed before. In the game configuration, the number of obstacles that are used are 2 and the goal is located behind both obstacles.

For the analysis, firstly we breakdown and analyze the program itself. In this part, we start from `environment.py` which is used to build the environment on our canvas. The first part inside this program file, as shown in 1, is the step to import some modules (`time`, `numpy`, and `tkinter`) and assign some fix parameters (the number of pixels, the height, and the width of our grid world canvas). The canvas is set to be 5×5 grid with 100 pixels inside each grid.

```
1  import time
2  import numpy as np
3  import tkinter as tk
4  from PIL import ImageTk, Image
5
6  np.random.seed(1)
7  PhotoImage = ImageTk.PhotoImage
8  UNIT = 100  # pixels
9  HEIGHT = 5  # grid height
10 WIDTH = 5  # grid width
```

Code 1: Environment of RL using Monte Carlo – Import Modules and Assign Fix Variables

The next code is belong to `Env` class. The first function defined in this file is for initializing some values and configurations. We can see in 2, the action space is defined as "u" for up, "d" for down, "l" for left, and "r" for right. So that the number of possible actions that agent can choose are 4. This value is defined as `n_actions`. Line 7 configures the shape of our canvas.

```
class Env(tk.Tk):
    def __init__(self):
        super(Env, self).__init__()
        self.action_space = ['u', 'd', 'l', 'r']
        self.n_actions = len(self.action_space)
        self.title('monte carlo')
        self.geometry('{0}x{1}'.format(HEIGHT * UNIT, HEIGHT * UNIT))
        self.shapes = self.load_images()
        self.canvas = self._build_canvas()
        self.texts = []
```

Code 2: Environment of RL using Monte Carlo – Initialization

Code 3 builds the display of our grid world canvas. As we can see in the line 6–11 creates the grids and line 14–17 adds and locates the object (rectangle as the agent, circle as the goal, and triangles as the obstacles) in the canvas.

```
def _build_canvas(self):
    canvas = tk.Canvas(self, bg='white',
                       height=HEIGHT * UNIT,
                       width=WIDTH * UNIT)
    # create grids
    for c in range(0, WIDTH * UNIT, UNIT):  # 0~400 by 80
        x0, y0, x1, y1 = c, 0, c, HEIGHT * UNIT
        canvas.create_line(x0, y0, x1, y1)
    for r in range(0, HEIGHT * UNIT, UNIT):  # 0~400 by 80
        x0, y0, x1, y1 = 0, r, HEIGHT * UNIT, r
        canvas.create_line(x0, y0, x1, y1)

    # add img to canvas
    self.rectangle = canvas.create_image(50, 50, image=self.shapes[0])
    self.triangle1 = canvas.create_image(250, 150, image=self.shapes[1])
    self.triangle2 = canvas.create_image(150, 250, image=self.shapes[1])
    self.circle = canvas.create_image(250, 250, image=self.shapes[2])

    # pack all
    canvas.pack()

    return canvas
```

Code 3: Environment of RL using Monte Carlo – Build the Canvas

Code 4 is for importing the images and resizing it to be 65×65 of the size.

```
def load_images(self):
    rectangle = PhotoImage(
        Image.open("img/rectangle.png").resize((65, 65)))
    triangle = PhotoImage(
        Image.open("img/triangle.png").resize((65, 65)))
    circle = PhotoImage(
        Image.open("img/circle.png").resize((65, 65)))

    return rectangle, triangle, circle
```

Code 4: Environment of RL using Monte Carlo – Load the Images

The following code is used for locating our agent, where x and y are the x and y axis respectively. The formula inside the function is for getting the exact grid of our agent, e.g. if our agent is located in (250,150), it means that our agent is in the grid (3,2). In the `_build_canvas` function, shown in Code 3, we can see that the rectangle is firstly located in (50,50). It means that it is situated in grid (1,1), and so do the triangles and circle.

```
@staticmethod
def coords_to_state(coords):
    x = int((coords[0] - 50) / 100)
    y = int((coords[1] - 50) / 100)
    return [x, y]
```

Code 5: Environment of RL using Monte Carlo – Get the Coordinate

Code 6 is for reseting our agent to be in the initial grid.

```
def reset(self):
    self.update()
    time.sleep(0.5)
    x, y = self.canvas.coords(self.rectangle)
    self.canvas.move(self.rectangle, UNIT / 2 - x, UNIT / 2 - y)

    # return observation
    return self.coords_to_state(self.canvas.coords(self.rectangle))
```

Code 6: Environment of RL using Monte Carlo – Reset the work

The next code sets the configuration of the movement of our agent. In line 6–17, we can see the rule for moving the rectangle to up (0), down (1), left (2) and right (3). Line 19 if for moving the rectangle based on the value that we get from the previous line. For line 27–36, we fix the reward in every grid that agent will get. If the agent reach the grid with circle, the reward that will be got is 100, for the grid with triangle, the reward is -100, otherwise is 0.

```
def step(self, action):
    state = self.canvas.coords(self.rectangle)
    base_action = np.array([0, 0])
    self.render()

    if action == 0:  # up
        if state[1] > UNIT:
            base_action[1] -= UNIT
    elif action == 1:  # down
        if state[1] < (HEIGHT - 1) * UNIT:
            base_action[1] += UNIT
    elif action == 2:  # left
        if state[0] > UNIT:
            base_action[0] -= UNIT
    elif action == 3:  # right
        if state[0] < (WIDTH - 1) * UNIT:
            base_action[0] += UNIT
```

```
18
19      # move agent
20      self.canvas.move(self.rectangle, base_action[0], base_action[1])
21
22      # move rectangle to top level of canvas
23      self.canvas.tag_raise(self.rectangle)
24      next_state = self.canvas.coords(self.rectangle)
25
26      # reward function
27      if next_state == self.canvas.coords(self.circle):
28          reward = 100
29          done = True
30      elif next_state in [self.canvas.coords(self.triangle1),
31                          self.canvas.coords(self.triangle2)]:
32          reward = -100
33          done = True
34      else:
35          reward = 0
36          done = False
37
38      next_state = self.coords_to_state(next_state)
39
40      return next_state, reward, done
```

Code 7: Environment of RL using Monte Carlo – Get the Next State and Reward

The following function is just for stop the program for a while, which is just 0.03 seconds, then update the work. The containing code in `update()` can be looked at Code 12 for detail.

```
1 def render(self):
2     time.sleep(0.03)
3     self.update()
```

Code 8: Environment of RL using Monte Carlo – Render and Update

Next code is gathered in `mc_agent.py` file. To run this file, we should import `numpy` and `random` module. We also import `defaultdict` function from `collections` and the previous file that we have made to provide the environment and movement rule inside our game.

```
1 import numpy as np
2 import random
3 from collections import defaultdict
4 from environment import Env
```

Code 9: RL using Monte Carlo – Import Modules

The following code is included in `MCAgent` class. Code 10 defines some initialization of our needed variables. The height and width of our canvas is initialized again in this part. The learning rate that we apply for updating our value of value function is 0.01. We use 0.9 for the discount factor. We use 0.1 for ε as the threshold in ε-greedy policy.

```
# Monte Carlo Agent which learns every episodes from the sample
class MCAgent:
    def __init__(self, actions):
        self.width = 5
        self.height = 5
        self.actions = actions
        self.learning_rate = 0.01
        self.discount_factor = 0.9
        self.epsilon = 0.1
        self.samples = []
        self.value_table = defaultdict(float)
```

Code 10: RL using Monte Carlo – Initialization

Code 11 is used to append the sample infomations and save it to an array. The return values in this code are the state, reward, and conditionality of the ending of an episode.

```
# append sample to memory(state, reward, done)
def save_sample(self, state, reward, done):
    self.samples.append([state, reward, done])
```

Code 11: RL using Monte Carlo – Append Sample

In every episode, the agent updates the Q-function of the visited states using the following formula.

$$V_{new}(t) \leftarrow V_{old}(t) + \alpha(G_{old}(t) - V_{old}(t)) \tag{1}$$

where for finite steps T, the real value function $G(t)$ can be assigned as

$$G(t) = R(t+1) + \gamma R(t+2) + \ldots + \gamma^{T-t+1} R(T) \tag{2}$$

Eq.1 is coded in line 11–12 in Code 12, while Eq.2 is expressed in line 9. The calculation shown in Code 12 is done reversely from the goal to the initial grid of the agent's path.

```
# for every episode, agent updates q function of visited states
def update(self):
    G_t = 0
    visit_state = []
    for reward in reversed(self.samples):
        state = str(reward[0])
        if state not in visit_state:
            visit_state.append(state)
            G_t = self.discount_factor * (reward[1] + G_t)
            value = self.value_table[state]
            self.value_table[state] = (value +
                                       self.learning_rate * (G_t - value))
```

Code 12: RL using Monte Carlo – Update Q-Function

The action is played using the ε-greedy policy, where in the line 4–10 expressed that if the random number is less than the $\varepsilon = 0.1$, the agent will take a random action, otherwise, the agent will choose the action that will maximize the value of the value function.

```
# get action for the state according to the q function table
# agent pick action of epsilon-greedy policy
def get_action(self, state):
    if np.random.rand() < self.epsilon:
        # take random action
        action = np.random.choice(self.actions)
    else:
        # take action according to the q function table
        next_state = self.possible_next_state(state)
        action = self.arg_max(next_state)
    return int(action)
```

Code 13: RL using Monte Carlo – Get Action

The following code is used to get the maximum value of the next state. If the number of the maximum value is more than 1, the agent will take a random choice between those actions.

```
# compute arg_max if multiple candidates exit, pick one randomly
@staticmethod
def arg_max(next_state):
    max_index_list = []
    max_value = next_state[0]
    for index, value in enumerate(next_state):
        if value > max_value:
            max_index_list.clear()
            max_value = value
            max_index_list.append(index)
        elif value == max_value:
            max_index_list.append(index)
    return random.choice(max_index_list)
```

Code 14: RL using Monte Carlo – Get the Maximum Value of the Next State

Code 15 gets the possible action for our agent. The conditional argument in line 6–21 express the action when the agent is in the border grid, for example if the agent is in the most top row of the grid, there is no action for going up, if the agent is in the most left column, there agent will not go to the left side, and soon.

```
# get the possible next states
def possible_next_state(self, state):
    col, row = state
    next_state = [0.0] * 4

    if row != 0:
        next_state[0] = self.value_table[str([col, row - 1])]
    else:
        next_state[0] = self.value_table[str(state)]
    if row != self.height - 1:
        next_state[1] = self.value_table[str([col, row + 1])]
    else:
        next_state[1] = self.value_table[str(state)]
    if col != 0:
        next_state[2] = self.value_table[str([col - 1, row])]
```

```
16      else:
17          next_state[2] = self.value_table[str(state)]
18      if col != self.width - 1:
19          next_state[3] = self.value_table[str([col + 1, row])]
20      else:
21          next_state[3] = self.value_table[str(state)]
22
23      return next_state
```

Code 15: RL using Monte Carlo – Get the Possible Next State

The following code express the main loop of the whole function. In the original code, the number of episodes are 1000. We can change this number by changing the input in line 6. In every loop or episode, after the agent finishes one episode, the state will be reset, so that the agent will come back to the intial state, which is (0,0).

```
1  # main loop
2  if __name__ == "__main__":
3      env = Env()
4      agent = MCAgent(actions=list(range(env.n_actions)))
5
6      for episode in range(1000):
7          state = env.reset()
8          action = agent.get_action(state)
9
10         while True:
11             env.render()
12
13             # forward to next state. reward is number and done is boolean
14             next_state, reward, done = env.step(action)
15             agent.save_sample(next_state, reward, done)
16
17             # get next action
18             action = agent.get_action(next_state)
19
20             # at the end of each episode, update the q function table
21             if done:
22                 print("episode : ", episode)
23                 agent.update()
24                 agent.samples.clear()
25                 break
```

Code 16: RL using Monte Carlo – Main Loop

The display of the canvas when we run the program is like in the Fig.1a. As we set in the code, we have a rectangle as the agent that initially located in grid (0,0), two triangles as the obstacles that are situated in grid (2,1) and (1,2), and a circle as a goal that located in the middle grid, which is (2,2). The value that we have set or will be updated is not shown in the display. So, for the easier seeing, we print some of the return values as the informations of our sighting. We will discuss about those values in the next part.

The original position of all object is shown in the Fig.1a. When the agent successfully reach the goal, the display is like in the Fig.1b. If the agent find the obtacles, the display will be like Fig.1c and 1d.

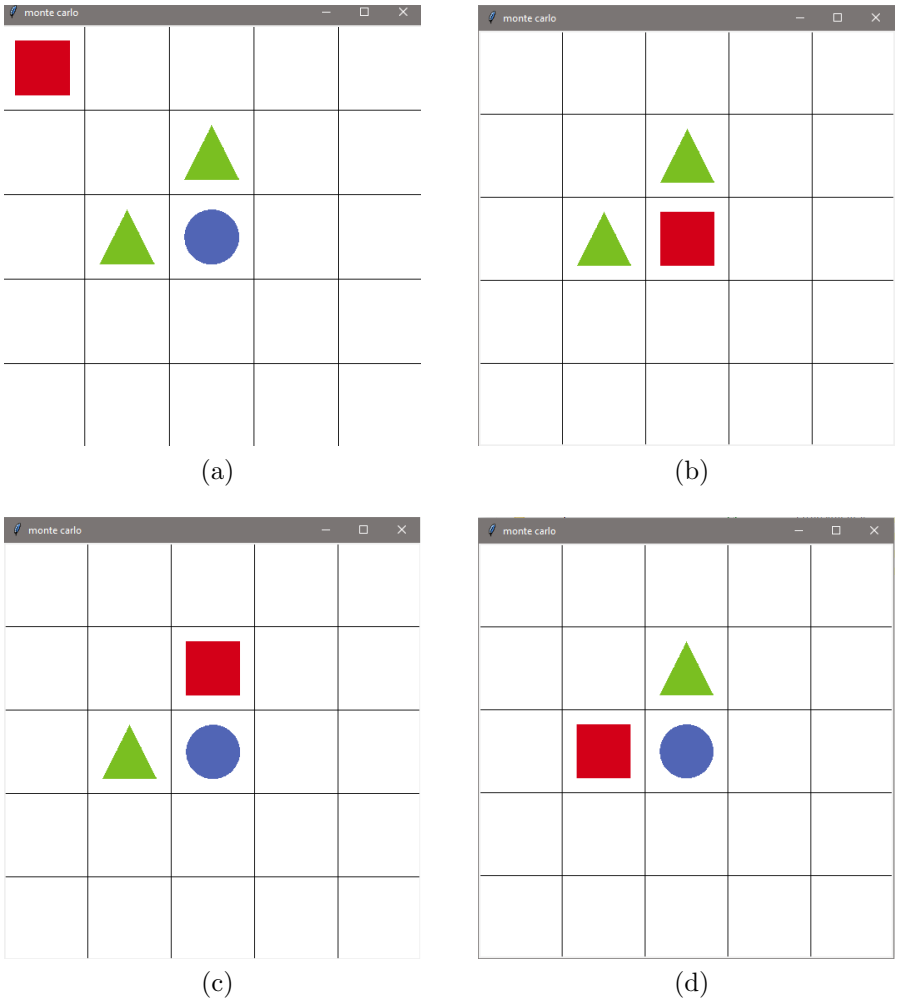


Figure 1: Display Canvas Monte Carlo: (a) Initial (b) Reaching the Goal (c) Reaching Obstacle 1 (d) Reaching Obstacle 2

In the original code, the number of episode is 1000. Each episode will finish if the agent reaches the goal or the obstacle (either obstacle 1 or 2). In the printed value in Fig.2, we can also see that the episode will be terminated if the agent get reward 100 or -100.

Fig.3 shows the value from the code. We just show a part of printed value in episode 1 (episode 0 in the figure is same as episode 1). In the first step, we can see that the initial action value is 0. The random value to compare with ε in ε-greedy policy is 0.41702, which is greater than $\varepsilon = 0.1$ that has been set before. In this step, the agent will do exploitation, choosing the maximal value of all possible actions. However, the action value now is still 0 for all actions, so using `arg_max()` in the Code 14, we will get 0, 1, 2, 3 as the choice of our action. Since we have more than one choices, we will randomly choose the action based on the candidate that we have got. In the example the agent chooses 1, which is go to the right. Now, our agent moves from state (0,0) to (0,1).

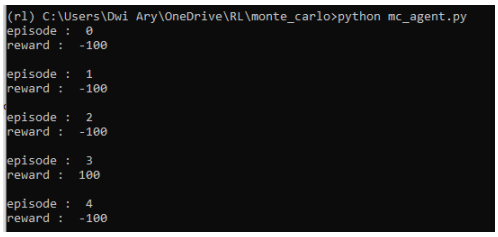


Figure 2: RL using Monte Carlo – Reward of the Episode Increment

If our random value for deciding whether do exploration or exploitation is lower than $\varepsilon = 0.1$, we will directly choose the action randomly, or do exploration, even though actually we have the maximum value if we do exploitation. This possibility is shown in the third part in Fig.3.

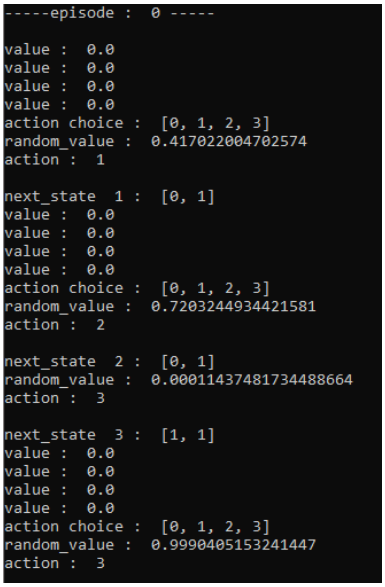
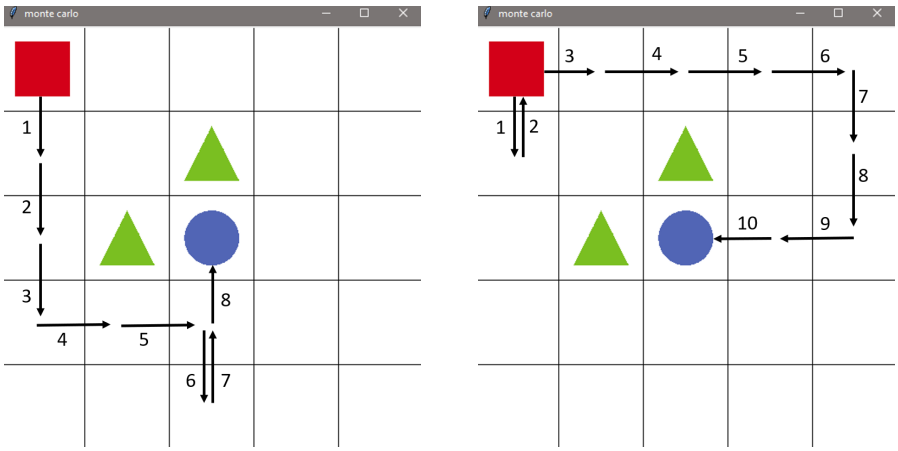


Figure 3: RL using Monte Carlo – Result of Episode 1

In every step, the agent always generates random value. The good thing of when the agent get less random value than our ε, which means that our agent do exploration, is when we start our training process, the agent do not have any idea about the environment yet, so it can explore it. However, if our agent do exploration when it has already got the "good path", or the path that can bring the agent to the goal, the agent will look like confusing because the agent will try the new action from the current state but then it comes back to the "good path". For example in Fig.4, in the first figure, the "good path" is step 1, 2, 3, 4, 5, and 8. From the step 5, the agent should directly go to step 8, but in the sixth step the random value that agent get is less than $\varepsilon = 0.1$, so the agent will do exploration. The exploration in this step is just randomly choosen one of the 4 possible policy, and the agent choose step 6. After that the agent get the random value greater than 0.1. Since the

agent do exploitation, the agent comes back to the current path. We also can see the same occurence when the agent seems confusing because of the exploration in the second figure.

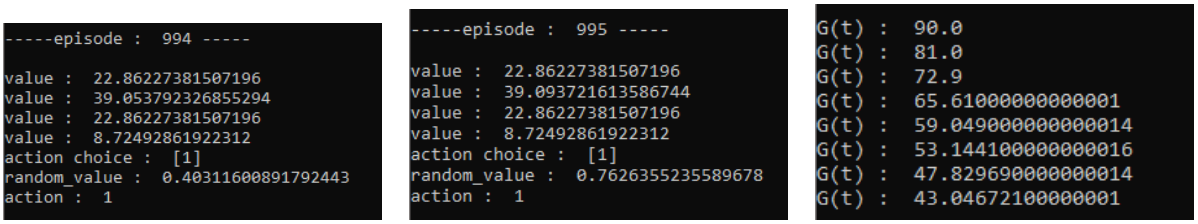


(a) (b)

Figure 4: RL using Monte Carlo – The Path which Agent Went Through in: (a) Episode 102 (b) Episode 997

The updating action value in this method is based on Eq.1. For example, we can calculate the updated value for step 996. The first value printed in the Fig.5a and 5b is respectively for action 0 (left), 1 (right), 2 (up), and 3 (down). If we see at those figure, the value that is changed is just the second value, because the chosen action in episode 995 is 1. Suppose we have $V_{old}(t) = 39.05379$, $G_{old}(t)$ for the first step is 43.04672, $\alpha = 0.01$, so can calculate:

$$V_{new}(t) \leftarrow 39.05379 + 0.01(43.04672 - 39.05379) = 39.09372$$



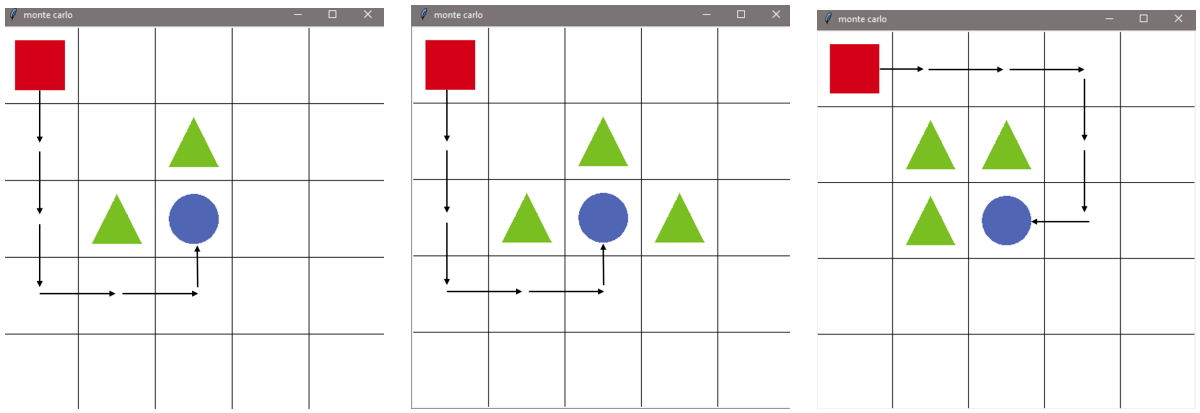
(a) (b) (c)

Figure 5: RL using Monte Carlo – Printed Values: (a) in Episode 995 (b) in Episode 996 (c) of $G(t)$ in Episode 995

Now we have $V_{new}(t) = 39.09372$, which is the second value in episode 996. $G(t)$ value in Fig.5c is decresing and positive, this is because of the reward that agent get in this episode 995 is 100 and based on Eq.2, the calculation of $G(t)$ for each grid is reversed from

the last grid to the first grid of the path. If the reward that agent get is -100, the value of $G(t)$ will increase from -100 to 0. This value will decrease the value of $V_{new}(s)$.

The following figure is the display after we add one more rectangle in the canvas and the convergence path that is mostly passed by the agent.



(a) (b) (c)

Figure 6: RL using Monte Carlo – The Convergence Path

Problem 2 Reinforcement Learning using SARSA.

SARSA is one the control methods in Reinforcement Learning (RL). SARSA stands for State-Action-Reward-State-Action. A SARSA agent interacts with the environment and updates the policy based on actions taken, hence this is known as an on-policy learning algorithm.

Basically the code for building the environment in SARSA is same with Monte Carlo. In SARSA, the addition function is just for showing the value of the value function. Code 17 is used to configure the font and Code 18 is used to appear that value.

```
def text_value(self, row, col, contents, action, font='Helvetica', size=10,
               style='normal', anchor="nw"):
    if action == 0:
        origin_x, origin_y = 7, 42
    elif action == 1:
        origin_x, origin_y = 85, 42
    elif action == 2:
        origin_x, origin_y = 42, 5
    else:
        origin_x, origin_y = 42, 77
    x, y = origin_y + (UNIT * col), origin_x + (UNIT * row)
    font = (font, str(size), style)
    text = self.canvas.create_text(x, y, fill="black", text=contents,
                                   font=font, anchor=anchor)
    return self.texts.append(text)
```

Code 17: Environment of RL using SARSA – Text Value

```
def print_value_all(self, q_table):
    for i in self.texts:
        self.canvas.delete(i)
    self.texts.clear()
    for x in range(HEIGHT):
        for y in range(WIDTH):
            for action in range(0, 4):
                state = [x, y]
                if str(state) in q_table.keys():
                    temp = q_table[str(state)][action]
                    self.text_value(y, x, round(temp, 2), action)
```

Code 18: Environment of RL using SARSA – Print the Value

The initialization part defines the learning rate ($\alpha = 0.01$), discount factor ($\gamma = 0.9$), $\varepsilon = 0.1$, and the intial Q-function values, which are 0.

```
class SARSAgent:
    def __init__(self, actions):
        self.actions = actions
        self.learning_rate = 0.01
        self.discount_factor = 0.9
        self.epsilon = 0.1
        self.q_table = defaultdict(lambda: [0.0, 0.0, 0.0, 0.0])
```

Code 19: RL using SARSA – Initialization

The following code is used for updating Q-function value. Argument in line 5–6 is following Eq.3.

$$Q_{new}(S_t, A_t) \leftarrow Q_{old}(S_t, A_t) + \alpha(R + \gamma Q_{old}(S_{t+1}, A_{t+1}) - Q_{old}(S_t, A_t)) \tag{3}$$

```
# with sample <s, a, r, s', a'>, learns new q function
def learn(self, state, action, reward, next_state, next_action):
    current_q = self.q_table[state][action]
    next_state_q = self.q_table[next_state][next_action]
    new_q = (current_q + self.learning_rate *
            (reward + self.discount_factor * next_state_q - current_q))
    self.q_table[state][action] = new_q
```

Code 20: RL using SARSA – Learn the Q-Function

The action is played using the ε-greedy policy, where in the line 4–10 expressed that if the random number is less than the $\varepsilon = 0.1$, the agent will take a random action, otherwise, the agent will choose the action that will maximize the value of the value function.

```
# get action for the state according to the q function table
# agent pick action of epsilon-greedy policy
def get_action(self, state):
    if np.random.rand() < self.epsilon:
```

```
5           # take random action
6           action = np.random.choice(self.actions)
7       else:
8           # take action according to the q function table
9           state_action = self.q_table[state]
10          action = self.arg_max(state_action)
11      return action
```

Code 21: RL using SARSA – Get Action

The following code is used to get the maximum value of the next state. If the number of the maximum value is more than 1, the agent will take a random choice between those actions.

```
1  @staticmethod
2  def arg_max(state_action):
3      max_index_list = []
4      max_value = state_action[0]
5      for index, value in enumerate(state_action):
6          if value > max_value:
7              max_index_list.clear()
8              max_value = value
9              max_index_list.append(index)
10         elif value == max_value:
11             max_index_list.append(index)
12     return random.choice(max_index_list)
```

Code 22: RL using SARSA – Get the Maximum Value of the Next State

The following code express the main loop of the whole function. In the original code, the number of episodes are 1000. We can change this number by changing the input in line 5. In one episode, the agent will take the action based on the decision that is get from Code 21. After that the agent will learn the new Q-function based on the action that will be acted and the next action. The action from the line 10 in Code 23 is the action that agent will do, after doing this action the agent will get return value of S and A from environment and also get reward R. The action in line 17 is the possible next action after doing the action from line 10, after this action the agent willl get return value of S' and A' from environment.

```
1  if __name__ == "__main__":
2      env = Env()
3      agent = SARSAgent(actions=list(range(env.n_actions)))
4
5      for episode in range(1000):
6          # reset environment and initialize state
7
8          state = Env.reset()
9          # get action of state from agent
10         action = agent.get_action(str(state))
11
12         while True:
13             env.render()
```

```
14
15              # take action and proceed one step in the environment
16              next_state, reward, done = env.step(action)
17              next_action = agent.get_action(str(next_state))
18
19              # with sample <s,a,r,s',a'>, agent learns new q function
20              agent.learn(str(state), action, reward, str(next_state), next_action)
21
22              state = next_state
23              action = next_action
24
25              # print q function of all states at screen
26              env.print_value_all(agent.q_table)
27
28              # if episode ends, then break
29              if done:
30                  break
```

Code 23: RL using SARSA – Main Loop

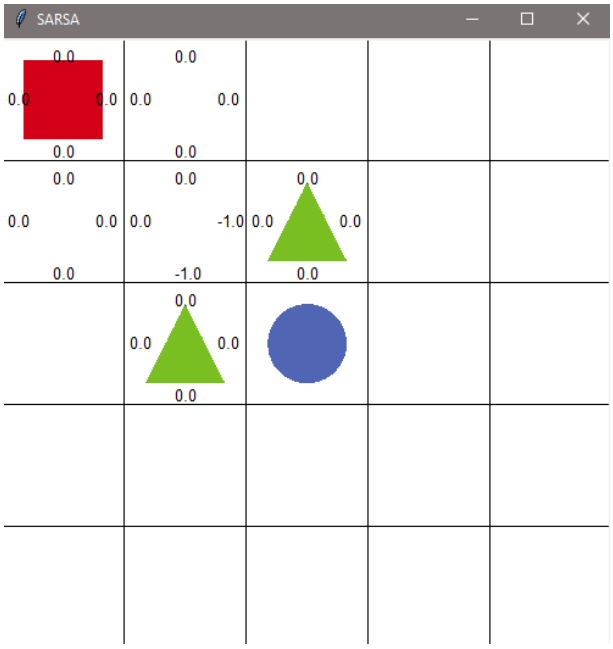


Figure 7: RL using SARSA – Display after 2 Episodes

The display when we run the program for SARSA is like in Fig.7. The very first display actually is like in Fig.1a. When the agent walk to another grid, the value for all possible actions is shown. For example in the Fig.7, the agent have had 2 episodes which finished by reaching the obstacle. In the starting point, the agent will show the value of the current grid then move based on ε-greedy policy. Then, for example the agent go to grid (0,1), the agent will calculate the value of $Q_{new}(S_t, A_t)$ for all possible actions. Then it will choose one of the policy based on those values using ε-greedy policy.

The `current Q`, `next state Q`, and `updated Q` that are printed in the Fig.8 are belong to the previous state. From that figure, suppose that for the first state we have $Q_{old}(S_t, A_t) = 6.64823 \times 10^{-7}$ and $Q_{old}(S_{t+1}, A_{t+1}) = 3.67254 \times 10^{-5}$. We also have had $\gamma = 0.9$ and $\alpha = 0.01$. Then the reward for the state without triangle or circle is always 0. So based on Eq.3 we can calculate the updated Q as:

$Q_{new}(S_t, A_t) \leftarrow 6.64823 \times 10^{-7} + 0.01[0 + 0.9(3.67254 \times 10^{-5} - 6.64823 \times 10^{-7}] = 9.88709 \times 10^{(} - 7)$

Before we update the Q value, first we should decide what is the action and the next action. To decide it, SARSA uses ε-greedy policy for both action. If we look at Code 23, we can see that in the line 22 and 23, the state and action for the next step is drawn from the next state and next action that have been got from the previous step. So for each step, the agent will just generate random value for getting the next state and next action. So the state in the third part of the screenshot in Fig.8 is actually the next state of the first part in that screenshot.

```
 episode :  24
random policy :  0.5460708041429516
max value :  8.731309364243653e-10
value :  8.731309364243653e-10
value :  6.648228791491856e-07
value :  1.4897755031776662e-09
value :  0.0
possible action :  [1]
action :  1

state :  [0, 1]
reward :  0
random policy :  0.5264259339055213
max value :  0.0
value :  0.0
value :  3.6725446322866643e-05
value :  0.0
value :  -0.009000000000000001
possible action :  [1]
current Q    :  6.648228791491856e-07
next state Q :  3.6725446322866643e-05
updated Q    :  9.887036672634935e-07
action :  1

STATUS :  False
state :  [0, 2]
reward :  0
random policy :  0.1354279030721699
max value :  0.0
value :  0.0
value :  0.0015633225767935214
value :  0.0
value :  -1.0
possible action :  [1]
current Q    :  3.6725446322866643e-05
next state Q :  0.0015633225767935214
updated Q    :  5.0428095050779665e-05
action :  1
```

Figure 8: RL using SARSA – Updating Q Value

Because the agent in SARSA always care about the next action, it is possible that the agent wil stuck in some grids and does not move for long time. It will affect the training time to reaching the goal.

If in the last step agent reach the obstacle, the Q value for going to it will be decrease and decrease. In the opposite, the agent with update the Q value positively if it reaches

the goal. We can see in Fig.9, the nearer the grid to the goal, the bigger the updated value to go there. We can also see that for the obstacles, the nearer the grid to the obstacles, the lower the Q value. It will make the agent avoid to go through that part.

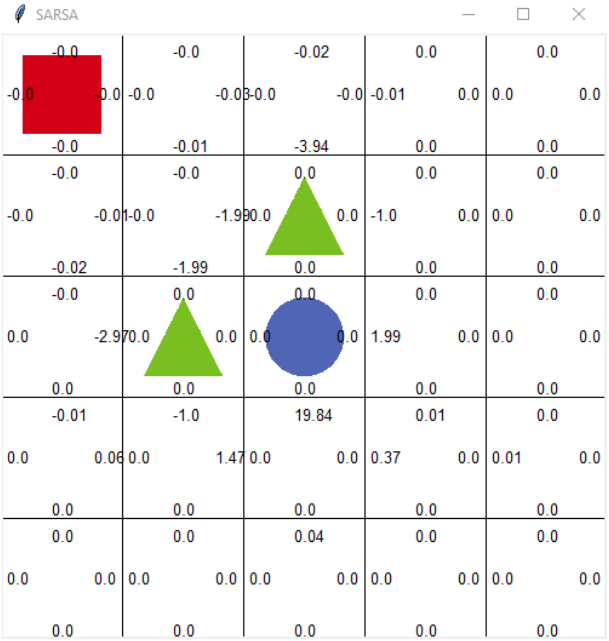
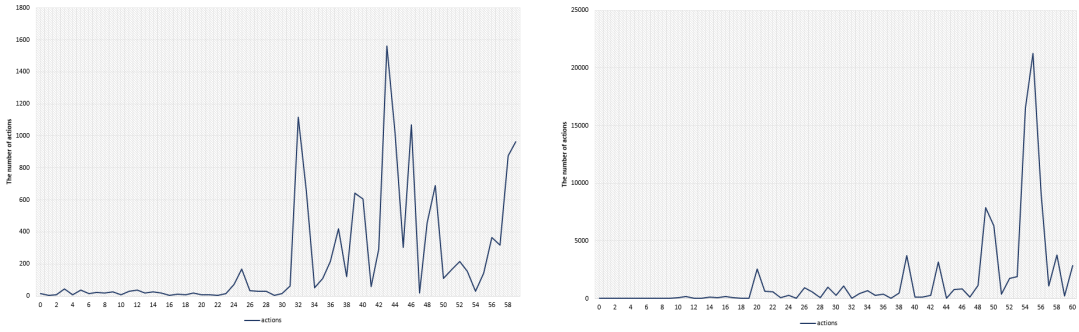


Figure 9: RL using SARSA – Display after Updating Q Value

The tendency that we will get from our agent when training our agent in the grid world using SARSA is if our agent more often reaches the triangles, the value of the Q function will be lower and lower. For example in the state (0,2), if the value to go to the right and down is being lower and lower, when the generated random value guides the agent to do exploitation, the agent will choose to go up or left. Going up means that the agent stuck in that state, and go to the left means that the agent comes back to the previous state. This action can be done so many times until the agent get the opportunity to explore or until the value to go right is higher than the others.

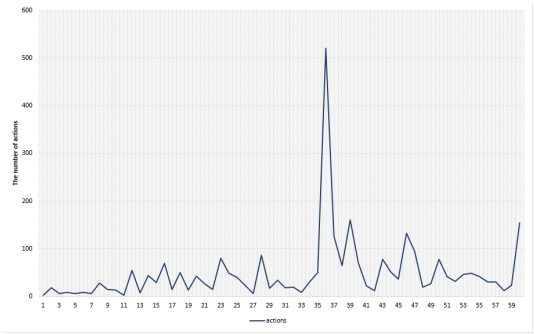
The scenario to add one more triangle in the canvas is same with the scenario in monte carlo agent. For the comparison, we do the experiment until 60 episodes then counts the number of the actions in each episode. The result of our experiment can be seen in Fig.10. Those plots have the same tendency, even though the scale of the graph is not same. In the beginning SARSA can learn the grid word fast, but together with the increasing number of the episodes, the agent learns slower and slower (it is shown from the increasing fluctuation of the graph). This can be guessed that due to in the beginning the agent reaches the triangle more than the circle, so when the value of the action to go to that triagles is always being lower and lower. In the other hand, it can make the agent more afraid to pass that grid or to do that action again in the following episode.

From the three scenarios that we have done, we found that when we run the second scenario, it takes the most time than the others.



(a)

(b)



(c)

Figure 10: RL using SARSA – The Graph of The Number of Actions for: (a) Grid World with 2 Obstacles, (b) Grid World with 3 Obstacles (Added One is in State (2,3)), and (c) Grid World with 3 Obstacles (Added One is in State (1,1))

Problem 3 Reinforcement Learning using Q-Learning.

Q-learning is one of the method in Reinforcement Learning. This method is belong to the off-policy control algorithm. In this method, the learned action value funtion, Q, directly approximates Q^*, the optimal action value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The Q-learning algorithm is shown in procedural form in Algorithm 1.

Basically the argument in the Q-learning code is same with in the previous explained code, `environment.py` in monte carlo and SARSA. The code for the agent also mostly same with the code in the SARSA, just the main loop and code for learning the Q-function are different. In Q-learning, there is no contraint on how the next action is selected, only that we have this "optimisitic" view that all hence forth action selections from every state should be optimal, thus we pick the action a' that maximizes $Q_{S_{t+1},a'}$. This means that with Q-learning we can give it data generated by any behaviour policy (expert, random, even bad policies) and it should learn the optimal Q-value given enough data samples.

The following code is for updating the Q-function in every step of our agent, based on the equation in Algorithm 1 line 7.

Algorithm 1 Q-Learning Algorithm

1: Initialize $Q(S, A)$ arbitrarily
2: **repeat** for each episode
3: Initialize S
4: **repeat** for each step of episode
5: Choose α from S using policy derived from Q (e.g., ε-greedy policy)
6: Take action α, observe r, S'
7: $Q_{new}(S_t, A_t) \leftarrow Q_{old}(S_t, A_t) + \alpha(R + \gamma \max_{a' \in A_{t+1}} Q_{old}(S'_{t+1}, a') - Q_{old}(S_t, A_t))$
8: **until** s is terminal
9: **until**

```
1 # update q function with sample <s, a, r, s'>
2 def learn(self, state, action, reward, next_state):
3     current_q = self.q_table[state][action]
4     # using Bellman Optimality Equation to update q function
5     new_q = reward + self.discount_factor * max(self.q_table[next_state])
6     self.q_table[state][action] += self.learning_rate * (new_q - current_q)
```

Code 24: RL using Q Learning – Learn the Q-Function

The following code express the main loop of the whole function. In the original code, the number of episodes are 1000. We can change this number by changing the input in line 5. In one episode, the agent will take the action based on the decision that is get from Code 21.

```
1  if __name__ == "__main__":
2      env = Env()
3      agent = QLearningAgent(actions=list(range(env.n_actions)))
4
5      for episode in range(1000):
6          state = env.reset()
7
8          while True:
9              env.render()
10
11             # take action and proceed one step in the environment
12             action = agent.get_action(str(state))
13             next_state, reward, done = env.step(action)
14
15             # with sample <s,a,r,s'>, agent learns new q function
16             agent.learn(str(state), action, reward, str(next_state))
17
18             state = next_state
19             env.print_value_all(agent.q_table)
20
21             # if episode ends, then break
22             if done:
23                 break
```

Code 25: RL using Q Learning – Main Loop

The display of the grid world canvas when we train it using Q learning basically same with SARSA. In every step of the agent, it will show the value of all possible actions and also update it. The updating rule in Q learning follows the equation in line 7 of Algorithm 1. For example, using the value that we get in Fig.11, we have $Q_{old}(S_t, A_t) = 8.416175$, $max_{a' \in A_{t+1}} Q_{old}(S'_{t+1}, a') = 17.72013$. The learning rate that is defined in the program is 0.01, the discount factor is 0.9, and the reward for this state is 0. Using 5 digits behind the decimal, we can calculate the $Q_{new}(S_t, A_t)$ as:

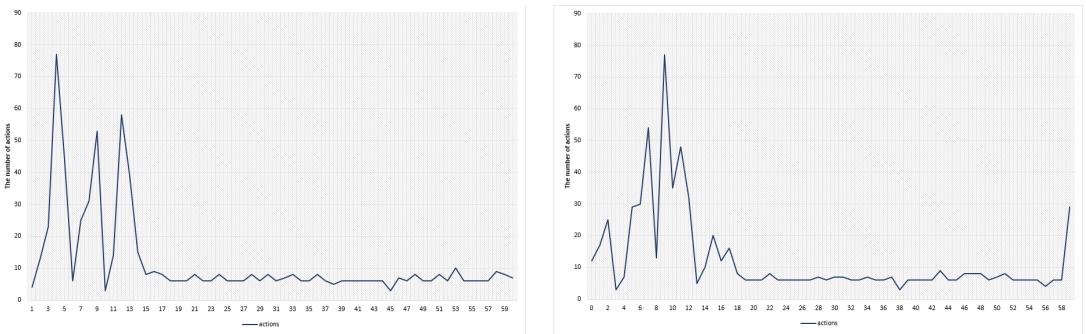
$$Q_{new}(S_t, A_t) \leftarrow 8.416175 + 0.01(0 + 0.9(17.72013) - 8.416175) = 8.49150$$

where $max_{a' \in A_{t+1}} Q_{old}(S'_{t+1}, a') = 17.72013$ actually is got from the maximum value of all values for the next possible action. We can see it in the second part in the following figure, among the value that are shown there, the value for action 1 is the biggest.

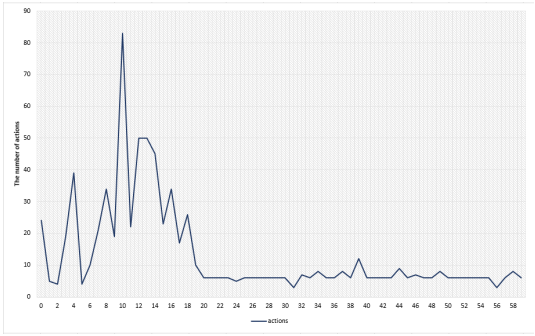
```
episode :  361
state :  [0, 0]
value :  0.07081479723500728
value :  8.41617518083031
value :  0.2503612631441551
value :  0.003498199846749725
action :  1
current Q :  8.41617518083031
new Q :  8.491494562484215
max next Q :  17.720125940245406
STATUS :  False

state :  [0, 1]
value :  0.1474080676338108
value :  17.720125940245406
value :  0.2616613468665955
value :  0.0017219839929197075
action :  1
current Q :  17.720125940245406
new Q :  17.841120730542
max next Q :  33.13289441100524
STATUS :  False

state :  [0, 2]
value :  0.34777343431150404
value :  33.13289441100524
value :  1.0169901864804753
value :  -8.64827525163591
action :  1
current Q :  33.13289441100524
new Q :  33.28517269724132
max next Q :  53.73413670512613
STATUS :  False
```

Figure 11: RL using Q Learning – Updating Q Value

We do the same experiments with the previous methods. The result of our experiment using Q learning is shown in Fig.12. The result of our experiment is totally different with SARSA. In SARSA, the graph has tendency to increase together with the number of the episode, which means that the model runs slower and slower. However, using this method, we can see that the graph is shown in the opposite way. In the beginning, the model train slowly then it is being faster and faster. The maximum value in the beginning is not as great as the number of actions in SARSA, that sometimes reaches 1000 or 2000 actions.



(a) (b)



(c)

Figure 12: RL using Q Learning – The Graph of The Number of Actions for: (a) Grid World with 2 Obstacles, (b) Grid World with 3 Obstacles (Added One is in State (2,3)), and (c) Grid World with 3 Obstacles (Added One is in State (1,1))

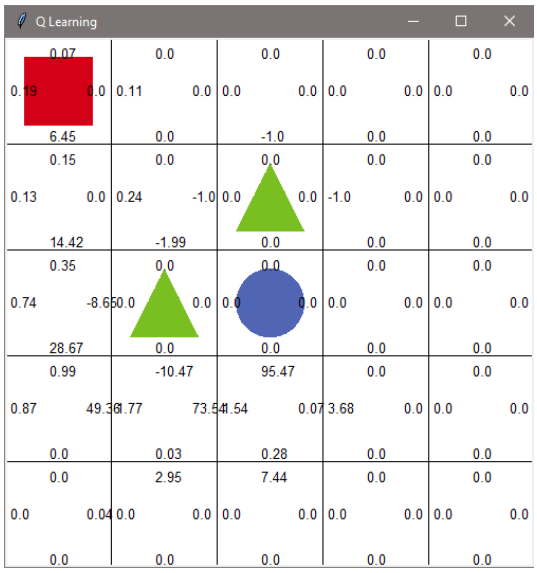


Figure 13: RL using Q Learning – Display After Updating Q Value