Iterative SARSA: The Modified SARSA Algorithm for Finding the Optimal Path

Submitted by

17BCE0813	**PRAJVAL MOHAN**
17BCE0880	**LAKSHYA SHARMA**
17BCE2210	**SIMRAN KOUL**
17BCE2213	**PRANAV NARAYAN**

Under the guidance of

PROF. NARAYANAMOORTHI M

In partial fulfillment for the award of the degree of

B.Tech

in

COMPUTER SCIENCE AND ENGINEERING

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

ABSTRACT

This paper presents a thorough comparative analysis of various reinforcement learning algorithms used by autonomous mobile robots for optimal path finding and, we propose a new algorithm called Iterative SARSA for the same. The main objective of the paper is to differentiate between the Q-learning and SARSA, and modify the latter. These algorithms use either the on-policy or off-policy methods of reinforcement learning. For the on-policy method, we have used the SARSA algorithm and for the off-policy method, the Q-learning algorithm has been used. These algorithms also have an impacting effect on finding the shortest path possible for the robot. Based on the results obtained, we have concluded how our algorithm is better than the current standard reinforcement learning algorithms.

Keywords - Reinforcement learning, Optimal path, On-policy, Off-policy, SARSA, Q-learning, Iterative SARSA

1. INTRODUCTION

The optimal path is the shortest path there is, after applying all the parameters and restrictions. The optimal path can be either safe or unsafe to execute. The safe paths do not pass close to obstacles or cliffs, unless absolutely necessary. On the other hand, the unsafe paths disregard their closeness to cliffs and obstacles in order to find the most optimal path (or the shortest path). The main need of path finding algorithms heightened when the concept of self-driving cars started to float. In the present scenario, most of the path calculations are done via reinforcement learning algorithms. Some of the reinforcement learning algorithms that have been highlighted in the recent times are SARSA and Q-learning. These algorithms trace the path of robot, helping it to avoid obstacles as well as reach the target location via the shortest path. The main concept used in these algorithms is the concept of rewards. Rewards are calculated based on two key policies, i.e., on-policy and off-policy. The on-policy method calculates the coordinate of the next step using the coordinate of the current step as well as the action of the current policy, by the process of exploration. This removes the question of experimental risk, which might lead to a sudden collapse of the program. SARSA uses this risk-free approach. On the other hand, the off-policy method predicts the next coordinate of the robot using the current coordinate of the robot along with a greedy action. The reward is calculated using this greedy policy, although the robot follows the path as though the greedy action wasn't used. The off-policy has a better outcome factor in terms of shortest path execution speed, but there is a heightened risk wherein the program might end up crashing. Developing on these algorithms, we have come up with a new algorithm, called Iterative SARSA, that provides us with the safest as well as the most optimal path available.

2. LITERATURE SURVEY

Reinforcement Learning

A striking assortment of issues in mechanical autonomy might be normally expressed as issues of reinforcement learning. Reinforcement learning empowers a robot to self-sufficiently find an ideal conduct through experimentation associations with the given condition. Rather than expressly specifying the answer to the issue, in order to calculate each step of the robot, the reinforcement methodology utilizes the feedback received from the agent in the form of a scalar function.

The main aim of reinforcement learning to figure out an optimal policy $\pi*$ that maps states to actions, hence, maximizing the expected return J, which is analogous to the aggregated expected reward.

$$J = E \{ \sum_{h=0}^{H} R_h \}$$

The rewards can also be discounted to accommodate errors. This is done using the discount factor γ (where, $0 \leq \gamma < 1$)

$$J = E \{ \sum_{h=0}^{\infty} \gamma^h R_h \}$$

where,
R_h is the overall reward w.r.t. agent h

Optimal Path

Traditionally, paths were calculated manually by entering the coordinates of the path into the program. Since this method was tedious and posed various problems, the need to create a new self-learning approach to autonomously calculate the path was needed. To calculate the path, the work environment with all the cliffs and obstacles is created and provided. The program runs the algorithm on this environment. This training is done with reinforcement learning and neural networks.

On-policy

The on-policy is a method of reinforcement learning in which the next state is calculated using the current state along with the data received by the exploration steps. The SARSA algorithm uses this policy. Although, this policy may be a bit more expensive, when compared to the off-policy method, it is safer and has lower risks.

Off-policy

The off-policy is a method of reinforcement learning in which the next state is calculated using the current state along with a greedy step. This policy predicts the next coordinate/state instead of calculating it, as seen in the on-policy method. The coordinate with the greatest reward is chosen as the next state. Even though the next state is chosen based on the reward system, the program processes it as though the greedy algorithm wasn’t used at all. On using this algorithm, even though we get the most optimal path possible, it might be risky in some situations, in which we end up with a high negative reward. The Q-learning algorithm uses this policy.

SARSA

SARSA (State-Action-Reward-State-Action) is called so, because it experiences to update the Q-values. It uses the on-policy method. SARSA is represented with the attributes {S, A, γ, α}, where, S is the set of states, A is a set of actions, $\gamma(0\leq\alpha\leq1)$ is the discount and $\alpha(0<\alpha\leq1)$ is the step size. These are used to calculate the Q-values (Q [S, A]).

$$Q_{new}(s_t,a_t) \leftarrow Q_{old}(s_t,a_t) + \alpha[r_t + \gamma.Q_{exploration\ value}(s_{t+1},a_{t+1}) - Q_{old}(s_t,a_t)]$$

where,
s_t represents the initial state,
s_{t+1} represents the final state,
r_t represents the reward

Q-learning

Q-learning is the reinforcement algorithm in which, the agent takes the most optimal action under different circumstances. This algorithm uses the off-policy model to obtain its Q-values, and hence, reward is calculated based on the greedy action. Q-learning is also represented with the attributes {S, A, γ, α}, where, S is the set of states, A is a set of actions, $\gamma(0\leq\alpha\leq1)$ is the discount and $\alpha(0<\alpha\leq1)$ is the step size. These are used to calculate the Q-values (Q [S, A]).

$$Q_{new}(s_t,a_t) \leftarrow Q_{old}(s_t,a_t) + \alpha[r_t + \gamma.\max_a.Q_{estimated\ future\ value}(s_{t+1},a) - Q_{old}(s_t,a_t)]$$

where,
s_t represents the initial state,
s_{t+1} represents the final state,
r_t represents the reward

3. PROPOSED METHOD

3.1 Iterative SARSA

Elaborating on the concept of the SARSA algorithm, we have proposed our new iterative algorithm. Like in the SARSA algorithm, the actor moves from the start to the end location. In the iterative SARSA algorithm, once the actor reaches the final destination, the start and end point are interchanged and the algorithm is rerun. According to the concept of SARSA, the path found is always a safe one, unlike in the Q-learning algorithm. Hence, any path that's calculated is going to be safe. Once the iteration is completed, the shortest path is selected out of all the iterations and the path is highlighted. Although, iterative SARSA has a high time complexity, it provides us with the best safe path.

3.2 Algorithm

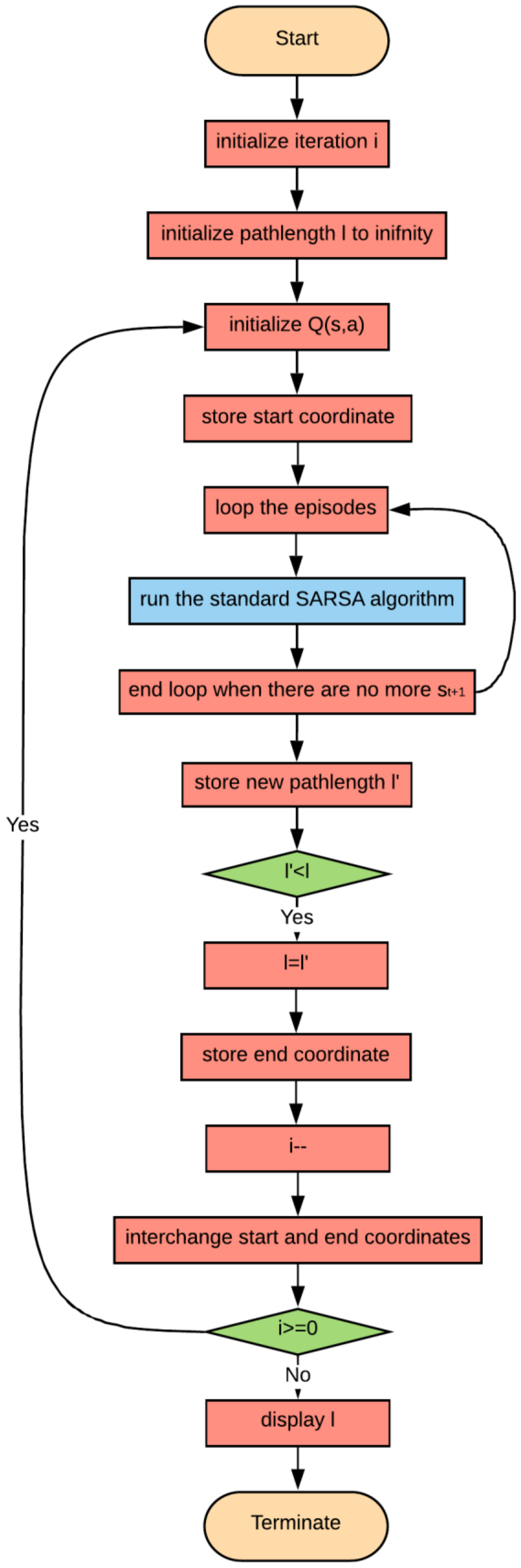
```
1: initialize number of iterations ‘i’
2: initialize pathlength $l \rightarrow \infty$ (based on grid environment)
3: initialize Q (s, a) for all s, a
4: store the coordinate value of the start location
5: loop over episodes
6: initialize s
7: repeat this for each step of the episode
8: choose a from s using the $\pi$* policy extracted from Q

9: $Q_{new}(s_t,a_t) \leftarrow Q_{old}(s_t,a_t)$
            $+\alpha[r_t + \gamma . Q_{exploration\ value}(s_{t+1},a_{t+1}) - Q_{old}(s_t,a_t)]$
10: $s_t \leftarrow s_{t+1}$
11: end loop
12: store the length of the path calculated in l’

13: if (l’< l)
    {
        l = l’;
    }
14: store the coordinate value of the end location
15: i--;
16: interchange the coordinate values of the start and end location

17: if ($i \geq 0$)
    {
        jump to step 3
    }
    else
    {
        display l and terminate
    }
```

3.3 Flowchart



4. EXPERIMENTAL SETUP

4.1 Hardware Requirements

CPU Specifications	
System Used	Dell Inspiron 7560
Processor	Intel® Core™ i7-7500 CPU @3.10 Ghz
Installed RAM	12.00 GB
System Type	64-bit Operating System, x64-based processor
GPU Specifications	
GPU Used	NVIDIA GeForce GTX 1080
Graphic Processor	GP104
Cores	2560
TMUS	160
ROPS	64
Memory Size	8.00 GB
Memory Type	GDDR5X
Bus Width	256 bit

4.2 Software Requirements

We have executed the algorithm on Python 3.7.0 and its dependent modules. The modules used in the program are:

Pandas	Used to read and manipulate the data.
Matplotlib	It is a python library used to depict the data in the form of various interactive graphs.
Pillow	It's image library in Python to manage, manipulate and save.
Timeit	It's a python library used to time blocks of code.

5. IMPLEMENTATION

The analysis for the program was done by creating a fixed environment wherein, we ran the Q-learning, SARSA and Iterative SARSA algorithm. Based on the time of execution, optimal path and safeness of the algorithm, we have concluded the appropriate scenarios for each algorithm. For the shortest path, we have used the Q-table to calculate and count the path length. As for the time calculation, we have used the timeit() function. The episode via steps and episode via cost graphs have also been calculated. An, episode is a sequence of steps the actor takes in each traversal. The episode via steps graph shows us the number of episodes vs. the number of steps in each episode and the episode via cost graph shows us the number of episodes vs. the cost of each episode.

Environment

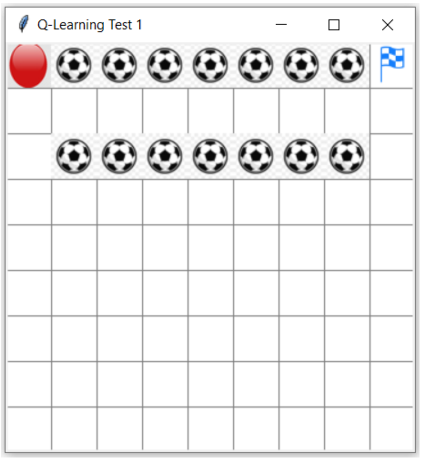


Fig1. The red dot is the actor and also represent the start location. The line of footballs represents the cliff and the flag represents the end location.

Q-learning

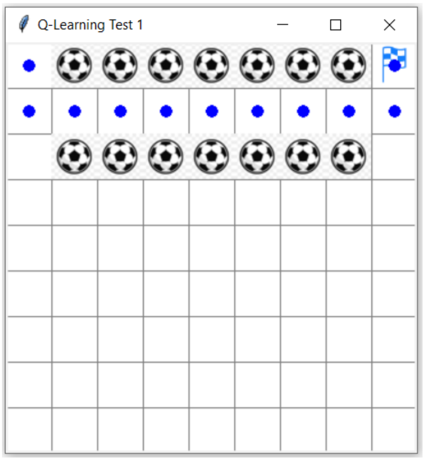


Fig2. Shortest path calculated using Q-learning with path length of 10

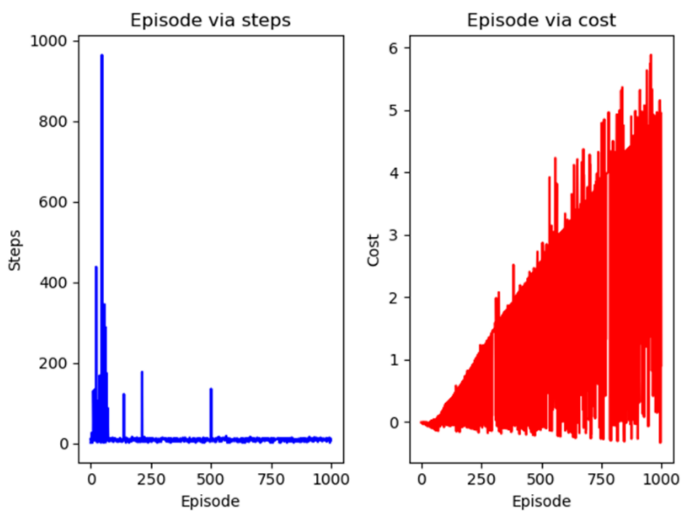


Fig3. Episode via steps and Episode via cost graphs

SARSA

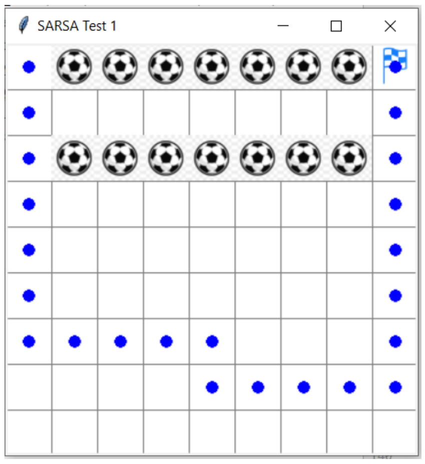


Fig4. One of the safe paths calculated using the SARSA algorithm with path length of 22

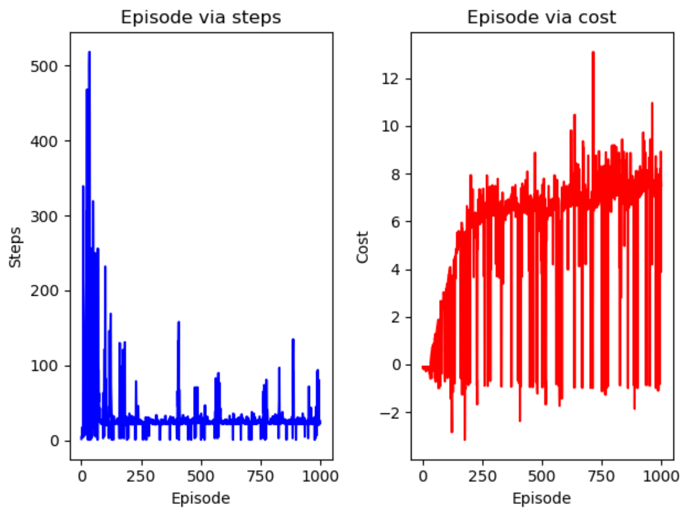
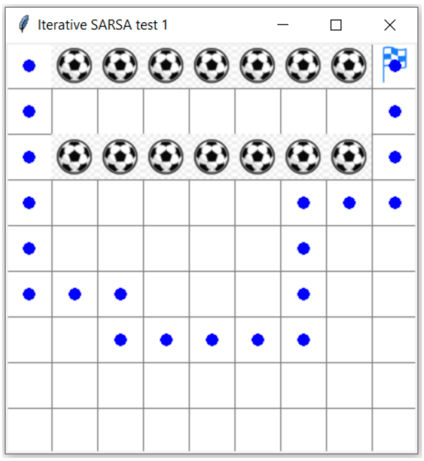


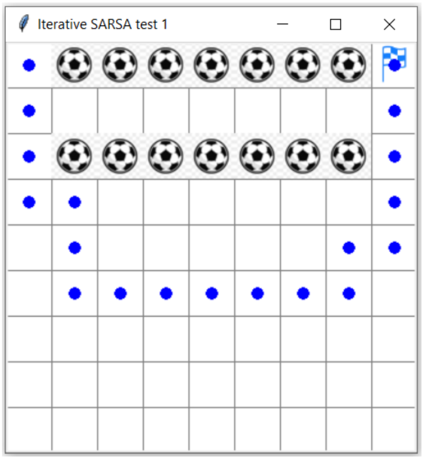
Fig5. Episode via steps and Episode via cost Graphs

Iterative SARSA

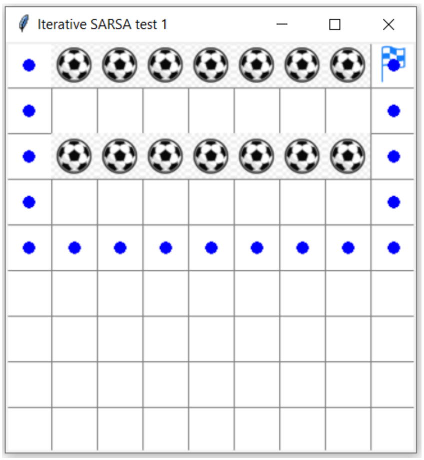
Iterations considered while Running the Code are 3.



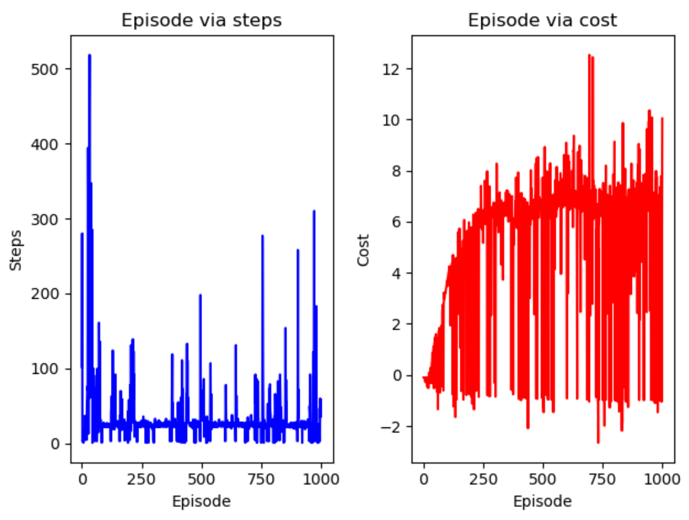
(a) Iteration 1 (from start to end) with path length 20



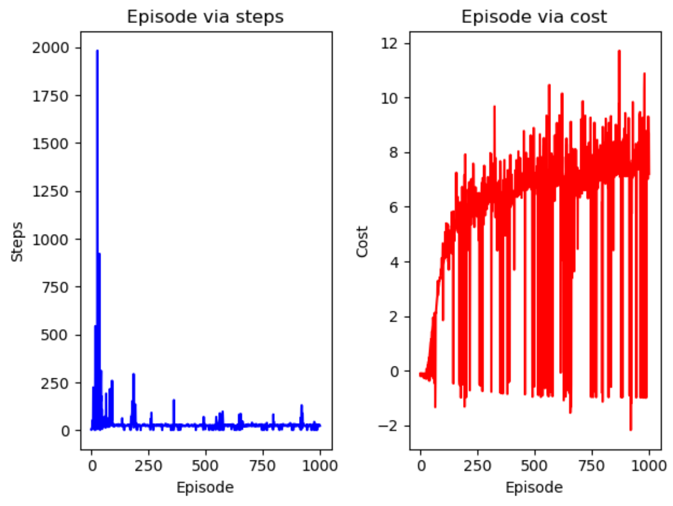
(b) Iteration 2 (from end to start) with path length 18



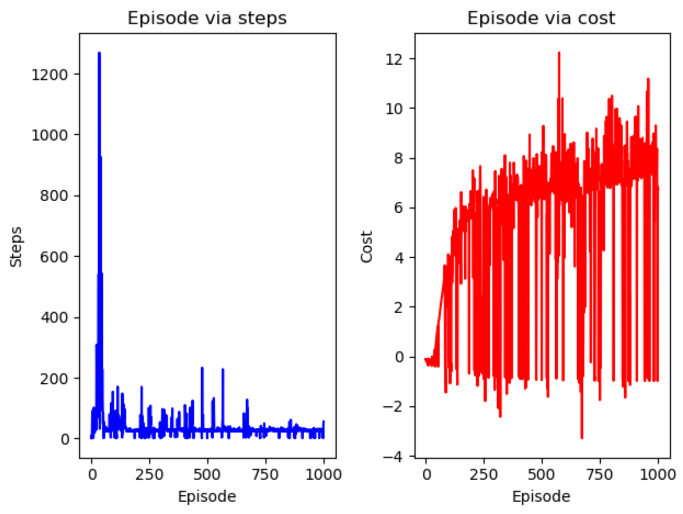
(c) Iteration 3 (from start to end) with path length 16
Fig6(a)(b)(c). Safe path calculated for each iteration using the Iterative SARSA algorithm



(a) Iteration number 1



(b) Iteration number 2



(c) Iteration number 3

Fig7(a)(b)(c). Episode via steps and Episode via cost graphs for each iteration using the Iterative SARAS algorithm

Inference Table

Algorithm		**Path Length**	**Execution Time (sec)**	**Safety**
Q-learning		10	11.3482	Executional Risk Present
SARSA		22	16.2297	Safe
Iterative SARSA	Iteration 1	20	17.6531	Safe
	Iteration 2	18	14.5246	
	Iteration 3	16	16.3137	

CONCLUSION

According to our inference, the Q-learning algorithm executes the fastest and gives the most optimal path. But, as we can see in Fig2., the path chosen by Q-learning passes between the 2 cliffs. This possesses a high executional risk in which the program might crash. Coming to SARSA, this algorithm takes comparatively more time to execute. Although this algorithm is safe, it has a longer path length (Fig4.). As we come to the Iterative SARSA algorithm, although it has a much higher time complexity, it finds the safest optimal path (Fig6(c).).
Hence, if one wants to calculate the path without considering the executional risks, the Q-learning algorithm can be considered. If a safe path, disregarding the path length, and with low execution time has to be calculated, the SARSA algorithm can be used. The Iterative SARSA algorithm can be used when time complexity is not an issue and we want the safest most optimal path.

REFERENCES

[1] Banerjee, Bikramjit & Sen, Sandip & Peng, Jing. (2004). On-policy concurrent reinforcement learning. J. Exp. Theor. Artif. Intell.. 16. 245-260. 10.1080/09528130412331297956.

[2] Park, Kui-Hong, Yong-Jae Kim, and Jong-Hwan Kim. "Modular Q-learning based multi-agent cooperation for robot soccer." Robotics and Autonomous Systems 35.2 (2001): 109-122.

[3] Chen, Chunlin, Han-Xiong Li, and Daoyi Dong. "Hybrid control for robot navigation-a hierarchical Q-learning algorithm." IEEE Robotics & Automation Magazine 15.2 (2008): 37-47.

[4] Konar, Amit, et al. "A deterministic improved Q-learning for path planning of a mobile robot." IEEE Transactions on Systems, Man, and Cybernetics: Systems 43.5 (2013): 1141-1153.

[5] Wang, Yin-Hao, Tzuu-Hseng S. Li, and Chih-Jui Lin. "Backward Q-learning: The combination of Sarsa algorithm and Q-learning." Engineering Applications of Artificial Intelligence 26.9 (2013): 2184-2193.

[6] H. van Seijen, H. van Hasselt, S. Whiteson and M. Wiering, "A theoretical and empirical analysis of Expected Sarsa," 2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, Nashville, TN, 2009, pp. 177-184.

[7] Stone, Peter, Richard S. Sutton, and Gregory Kuhlmann. "Reinforcement learning for robocup soccer keepaway." Adaptive Behavior 13.3 (2005): 165-188.

[8] Lin, Lian-ming, Hao Wang, and Yi-xiong Wang. "Sarsa Reinforcement Learning Algorithm based on Neural Networks." Computer Technology and Development 1 (2006): 30-32.

[9] Sariff, N., and Norlida Buniyamin. "An overview of autonomous mobile robot path planning algorithms." 2006 4th Student Conference on Research and Development. IEEE, 2006.

[10] Jan, Gene Eu, Ki Yin Chang, and Ian Parberry. "Optimal path planning for mobile robot navigation." IEEE/ASME transactions on mechatronics 13.4 (2008): 451-460.

[11] Fan, Xiaoping, et al. "Optimal path planning for mobile robots based on intensified ant colony optimization algorithm." IEEE International Conference on Robotics, Intelligent Systems and Signal Processing, 2003. Proceedings. 2003. Vol. 1. IEEE, 2003.

[12] Wang, Zeying, et al. "The optimization of path planning for multi-robot system using Boltzmann Policy based Q-learning algorithm." 2013 IEEE International Conference on Robotics and Biomimetics (ROBIO). IEEE, 2013.

[13] Gu, Shixiang, et al. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates." 2017 IEEE international conference on robotics and automation (ICRA). IEEE, 2017.

[14] Nair, Ashvin, et al. "Overcoming exploration in reinforcement learning with demonstrations." 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2018.

[15] Plappert, Matthias, et al. "Multi-goal reinforcement learning: Challenging robotics environments and request for research." arXiv preprint arXiv:1802.09464 (2018).

[16] Raja, Purushothaman, and Sivagurunathan Pugazhenthi. "Optimal path planning of mobile robots: A review." International journal of physical sciences 7.9 (2012): 1314-1320.

[17] Ichter, Brian, James Harrison, and Marco Pavone. "Learning sampling distributions for robot motion planning." 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2018.

[18] Tai, Lei, Shaohua Li, and Ming Liu. "A deep-network solution towards model-less obstacle avoidance." 2016 IEEE/RSJ international conference on intelligent robots and systems (IROS). IEEE, 2016.

[19] Tai, Lei, and Ming Liu. "A robot exploration strategy based on q-learning network." 2016 IEEE International Conference on Real-time Computing and Robotics (RCAR). IEEE, 2016.

[20] Ganger, Michael, Ethan Duryea, and Wei Hu. "Double Sarsa and double expected Sarsa with shallow and deep learning." Journal of Data Analysis and Information Processing 4.4 (2016): 159-176.

[21] Yu, Shanqing, et al. "Q value-based Dynamic Programming with SARSA Learning for real time route guidance in large scale road networks." The 2012 International Joint Conference on Neural Networks (IJCNN). IEEE, 2012.

[22] Luo, Wei, et al. "Deep-Sarsa based multi-UAV path planning and obstacle avoidance in a dynamic environment." International Conference on Sensing and Imaging. Springer, Cham, 2018.

[23] Gosavi, Abhijit. "A reinforcement learning algorithm based on policy iteration for average reward: Empirical results with yield management and convergence analysis."

SAMPLE CODE

A. Environment

```
# Importing libraries
import numpy as np  # To deal with data in form of matrices
import tkinter as tk  # To build GUI
import time  # Time is needed to slow down the agent and to see how he runs
from PIL import Image, ImageTk  # For adding images into the canvas widget

# Setting the sizes for the environment
pixels = 40   # pixels
env_height = 9  # grid height
env_width = 9  # grid width

# Global variable for dictionary with coordinates for the final route
a = {}

# Creating class for the environment
class Environment(tk.Tk, object):
    def __init__(self):
        super(Environment, self).__init__()
        self.action_space = ['up', 'down', 'left', 'right']
        self.n_actions = len(self.action_space)
        self.title('Q-Learning Test 1')
        self.geometry('{0}x{1}'.format(env_height * pixels, env_height * pixels))
        self.build_environment()

        # Dictionaries to draw the final route
        self.d = {}
        self.f = {}

        # Key for the dictionaries
        self.i = 0
        # Writing the final dictionary first time
        self.c = True

        # Showing the steps for longest found route
        self.longest = 0

        # Showing the steps for the shortest route
        self.shortest = 0

    # Function to build the environment
    def build_environment(self):
        self.canvas_widget = tk.Canvas(self,  bg='white',
                                           height=env_height * pixels,
                                           width=env_width * pixels)

        # Uploading an image for background
        # img_background = Image.open("images/bg.png")
        # self.background = ImageTk.PhotoImage(img_background)
        # # Creating background on the widget
        # self.bg = self.canvas_widget.create_image(0, 0, anchor='nw', image=self.background)

        # Creating grid lines
        for column in range(0, env_width * pixels, pixels):
            x0, y0, x1, y1 = column, 0, column, env_height * pixels
            self.canvas_widget.create_line(x0, y0, x1, y1, fill='grey')
        for row in range(0, env_height * pixels, pixels):
            x0, y0, x1, y1 = 0, row, env_height * pixels, row
            self.canvas_widget.create_line(x0, y0, x1, y1, fill='grey')

        # Creating objects of  Obstacles
        # Obstacle type 1 - road closed1
        img_obstacle1 = Image.open("images/road_closed1.png")
        self.obstacle1_object = ImageTk.PhotoImage(img_obstacle1)
```

```
# Obstacle type 2 - tree1
img_obstacle2 = Image.open("images/tree1.png")
self.obstacle2_object = ImageTk.PhotoImage(img_obstacle2)
# Obstacle type 3 - tree2
img_obstacle3 = Image.open("images/tree2.png")
self.obstacle3_object = ImageTk.PhotoImage(img_obstacle3)
# Obstacle type 4 - building1
img_obstacle4 = Image.open("images/tree1.png")
self.obstacle4_object = ImageTk.PhotoImage(img_obstacle4)
# Obstacle type 5 - building2
img_obstacle5 = Image.open("images/building2.png")
self.obstacle5_object = ImageTk.PhotoImage(img_obstacle5)
# Obstacle type 6 - road closed2
img_obstacle6 = Image.open("images/road_closed2.png")
self.obstacle6_object = ImageTk.PhotoImage(img_obstacle6)
# Obstacle type 7 - road closed3
img_obstacle7 = Image.open("images/tree1.png")
self.obstacle7_object = ImageTk.PhotoImage(img_obstacle7)
# Obstacle type 8 - traffic lights
img_obstacle8 = Image.open("images/traffic_lights.png")
self.obstacle8_object = ImageTk.PhotoImage(img_obstacle8)
# Obstacle type 9 - pedestrian
img_obstacle9 = Image.open("images/pedestrian.png")
self.obstacle9_object = ImageTk.PhotoImage(img_obstacle9)
# Obstacle type 10 - shop
img_obstacle10 = Image.open("images/shop.png")
self.obstacle10_object = ImageTk.PhotoImage(img_obstacle10)
# Obstacle type 11 - bank1
img_obstacle11 = Image.open("images/bank1.png")
self.obstacle11_object = ImageTk.PhotoImage(img_obstacle11)
# Obstacle type 12 - bank2
img_obstacle12 = Image.open("images/bank2.png")
self.obstacle12_object = ImageTk.PhotoImage(img_obstacle12)
```

```
        # Creating obstacles themselves
        # Obstacles from 1 to 22
        self.obstacle1 = self.canvas_widget.create_image(pixels*2, 0, anchor='nw', image=self.obstacle2_object)
        # Obstacle 2
        self.obstacle2 = self.canvas_widget.create_image(pixels*3, 0, anchor='nw', image=self.obstacle6_object)
        # Obstacle 3
        self.obstacle3 = self.canvas_widget.create_image(pixels*5, 0, anchor='nw', image=self.obstacle5_object)
        # Obstacle 4
        self.obstacle4 = self.canvas_widget.create_image(pixels * 1, pixels * 0, anchor='nw', image=self.obstacle2_object)
        # Obstacle 5
        self.obstacle5 = self.canvas_widget.create_image(pixels * 4, 0, anchor='nw', image=self.obstacle12_object)
        # Obstacle 6
        self.obstacle6 = self.canvas_widget.create_image(pixels * 6, 0, anchor='nw', image=self.obstacle7_object)
        # Obstacle 7
        self.obstacle7 = self.canvas_widget.create_image(pixels * 7, 0, anchor='nw', image=self.obstacle7_object)
        self.obstacle8 = self.canvas_widget.create_image(pixels * 1,pixels * 2 , anchor='nw', image=self.obstacle7_object)
        self.obstacle9 = self.canvas_widget.create_image(pixels * 2, pixels * 2, anchor='nw', image=self.obstacle7_object)
        self.obstacle10 = self.canvas_widget.create_image(pixels * 3, pixels * 2, anchor='nw', image=self.obstacle7_object)
        self.obstacle11 = self.canvas_widget.create_image(pixels * 4, pixels * 2, anchor='nw', image=self.obstacle7_object)
        self.obstacle12 = self.canvas_widget.create_image(pixels * 5, pixels * 2, anchor='nw', image=self.obstacle7_object)
        self.obstacle13 = self.canvas_widget.create_image(pixels * 6, pixels * 2, anchor='nw', image=self.obstacle7_object)
        self.obstacle14 = self.canvas_widget.create_image(pixels * 7, pixels * 2, anchor='nw', image=self.obstacle7_object)
        # Final Point
        img_flag = Image.open("images/flag.png")
        self.flag_object = ImageTk.PhotoImage(img_flag)
        self.flag = self.canvas_widget.create_image(pixels * 8, 0, anchor='nw', image=self.flag_object)

        # Uploading the image of Mobile Robot
        img_robot = Image.open("images/agent1.png")
        self.robot = ImageTk.PhotoImage(img_robot)

        # Creating an agent with photo of Mobile Robot
        self.agent = self.canvas_widget.create_image(0, 0, anchor='nw', image=self.robot)

        # Packing everything
        self.canvas_widget.pack()

    # Function to reset the environment and start new Episode
    def reset(self):
        self.update()
        #time.sleep(0.1)

        # Updating agent
        self.canvas_widget.delete(self.agent)
        self.agent = self.canvas_widget.create_image(0, 0, anchor='nw', image=self.robot)

        # # Clearing the dictionary and the i
        self.d = {}
        self.i = 0

        # Return observation
        return self.canvas_widget.coords(self.agent)

    # Function to get the next observation and reward by doing next step
    def step(self, action):
```

```
# Current state of the agent
state = self.canvas_widget.coords(self.agent)
base_action = np.array([0, 0])

# Updating next state according to the action
# Action 'up'
if action == 0:
    if state[1] >= pixels:
        base_action[1] -= pixels
# Action 'down'
elif action == 1:
    if state[1] < (env_height - 1) * pixels:
        base_action[1] += pixels
# Action right
elif action == 2:
    if state[0] < (env_width - 1) * pixels:
        base_action[0] += pixels
# Action left
elif action == 3:
    if state[0] >= pixels:
        base_action[0] -= pixels

# Moving the agent according to the action
self.canvas_widget.move(self.agent, base_action[0], base_action[1])

# Writing in the dictionary coordinates of found route
self.d[self.i] = self.canvas_widget.coords(self.agent)

# Updating next state
next_state = self.d[self.i]

# Updating key for the dictionary
self.i += 1
```

```
# Calculating the reward for the agent
if next_state == self.canvas_widget.coords(self.flag):
    reward = 1
    done = True
    next_state = 'goal'

    # Filling the dictionary first time
    if self.c == True:
        for j in range(len(self.d)):
            self.f[j] = self.d[j]
        self.c = False
        self.longest = len(self.d)
        self.shortest = len(self.d)

    # Checking if the currently found route is shorter
    if len(self.d) < len(self.f):
        # Saving the number of steps for the shortest route
        self.shortest = len(self.d)
        # Clearing the dictionary for the final route
        self.f = {}
        # Reassigning the dictionary
        for j in range(len(self.d)):
            self.f[j] = self.d[j]

    # Saving the number of steps for the longest route
    if len(self.d) > self.longest:
        self.longest = len(self.d)

elif next_state in [self.canvas_widget.coords(self.obstacle1),
                    self.canvas_widget.coords(self.obstacle2),
                    self.canvas_widget.coords(self.obstacle4),
                    self.canvas_widget.coords(self.obstacle7),
                    self.canvas_widget.coords(self.obstacle3),
                    self.canvas_widget.coords(self.obstacle5),
                        self.canvas_widget.coords(self.obstacle6),
                        self.canvas_widget.coords(self.obstacle8),
                        self.canvas_widget.coords(self.obstacle9),
                        self.canvas_widget.coords(self.obstacle10),
                        self.canvas_widget.coords(self.obstacle11),
                        self.canvas_widget.coords(self.obstacle12),
                        self.canvas_widget.coords(self.obstacle13),
                        self.canvas_widget.coords(self.obstacle14)]:
            reward = -1
            done = True
            next_state = 'obstacle'

            # Clearing the dictionary and the i
            self.d = {}
            self.i = 0

        else:
            reward = 0
            done = False

        return next_state, reward, done

    # Function to refresh the environment
    def render(self):
        #time.sleep(0.03)
        self.update()

    # Function to show the found route
    def final(self):
        # Deleting the agent at the end
        self.canvas_widget.delete(self.agent)
        # Showing the number of steps
        print('The shortest route:', self.shortest)
        print('The longest route:', self.longest)

        # Creating initial point
        origin = np.array([20, 20])
        self.initial_point = self.canvas_widget.create_oval(
            origin[0] - 5, origin[1] - 5,
            origin[0] + 5, origin[1] + 5,
            fill='blue', outline='blue')

        # Filling the route
        for j in range(len(self.f)):
            # Showing the coordinates of the final route
            print(self.f[j])
            self.track = self.canvas_widget.create_oval(
                self.f[j][0] + origin[0] - 5, self.f[j][1] + origin[0] - 5,
                self.f[j][0] + origin[0] + 5, self.f[j][1] + origin[0] + 5,
                fill='blue', outline='blue')
            # Writing the final route in the global variable a
            a[j] = self.f[j]


# Returning the final dictionary with route coordinates
# Then it will be used in agent_brain.py
def final_states():
    return a

# This we need to debug the environment
# If we want to run and see the environment without running full algorithm
if __name__ == '__main__':
    env = Environment()
    env.mainloop()
```

B. Q- learning

Executional Code

```
# Importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# Importing function from the env.py
from env import final_states

# Creating class for the Q-learning table
class QLearningTable:
    def __init__(self, actions, learning_rate=0.01, reward_decay=0.9, e_greedy=0.9):
        # List of actions
        self.actions = actions
        # Learning rate
        self.lr = learning_rate
        # Value of gamma
        self.gamma = reward_decay
        # Value of epsilon
        self.epsilon = e_greedy
        # Creating full Q-table for all cells
        self.q_table = pd.DataFrame(columns=self.actions, dtype=np.float64)
        # Creating Q-table for cells of the final route
        self.q_table_final = pd.DataFrame(columns=self.actions, dtype=np.float64)

    # Function for choosing the action for the agent
    def choose_action(self, observation):
        # Checking if the state exists in the table
        self.check_state_exist(observation)
        # Selection of the action - 90 % according to the epsilon == 0.9
        # Choosing the best action
        if np.random.uniform() < self.epsilon:
            state_action = self.q_table.loc[observation, :]
            state_action = state_action.reindex(np.random.permutation(state_action.index))
            action = state_action.idxmax()
        else:
            # Choosing random action - left 10 % for choosing randomly
            action = np.random.choice(self.actions)
        return action

    # Function for learning and updating Q-table with new knowledge
    def learn(self, state, action, reward, next_state):
        # Checking if the next step exists in the Q-table
        self.check_state_exist(next_state)

        # Current state in the current position
        q_predict = self.q_table.loc[state, action]

        # Checking if the next state is free or it is obstacle or goal
        if next_state != 'goal' or next_state != 'obstacle':
            q_target = reward + self.gamma * self.q_table.loc[next_state, :].max()
        else:
            q_target = reward

        # Updating Q-table with new knowledge
        self.q_table.loc[state, action] += self.lr * (q_target - q_predict)

        return self.q_table.loc[state, action]

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

# Printing the Q-table with states
def print_q_table(self):
    # Getting the coordinates of final route from env.py
    e = final_states()

    # Comparing the indexes with coordinates and writing in the new Q-table values
    for i in range(len(e)):
        state = str(e[i])  # state = '[5.0, 40.0]'
        # Going through all indexes and checking
        for j in range(len(self.q_table.index)):
            if self.q_table.index[j] == state:
                self.q_table_final.loc[state, :] = self.q_table.loc[state, :]

    print()
    print('Length of final Q-table =', len(self.q_table_final.index))
    print('Final Q-table with values from the final route:')
    print(self.q_table_final)

    print()
    print('Length of full Q-table =', len(self.q_table.index))
    print('Full Q-table:')
    print(self.q_table)

# Plotting the results for the number of steps
def plot_results(self, steps, cost):
    #
    f, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)
    #
    ax1.plot(np.arange(len(steps)), steps, 'b')
    ax1.set_xlabel('Episode')
    ax1.set_ylabel('Steps')
```

```
ax1.set_title('Episode via steps')

#
ax2.plot(np.arange(len(cost)), cost, 'r')
ax2.set_xlabel('Episode')
ax2.set_ylabel('Cost')
ax2.set_title('Episode via cost')

plt.tight_layout()  # Function to make distance between figures

#
plt.figure()
plt.plot(np.arange(len(steps)), steps, 'b')
plt.title('Episode via steps')
plt.xlabel('Episode')
plt.ylabel('Steps')

#
plt.figure()
plt.plot(np.arange(len(cost)), cost, 'r')
plt.title('Episode via cost')
plt.xlabel('Episode')
plt.ylabel('Cost')

# Showing the plots
plt.show()
```

Code for Visualization

```
# Importing classes
from env import Environment
from agent_brain import QLearningTable

def update():
    # Resulted list for the plotting Episodes via Steps
    steps = []

    # Summed costs for all episodes in resulted list
    all_costs = []

    for episode in range(1000):
        # Initial Observation
        observation = env.reset()

        # Updating number of Steps for each Episode
        i = 0

        # Updating the cost for each episode
        cost = 0

        while True:
            # Refreshing environment
            env.render()

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

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

            # RL learns from this transition and calculating the cost
            cost += RL.learn(str(observation), action, reward, str(observation_))
            # Swapping the observations - current and next
            observation = observation_

            # Calculating number of Steps in the current Episode
            i += 1

            # Break while loop when it is the end of current Episode
            # When agent reached the goal or obstacle
            if done:
                steps += [i]
                all_costs += [cost]
                break

    # Showing the final route
    env.final()

    # Showing the Q-table with values for each action
    RL.print_q_table()

    # Plotting the results
    RL.plot_results(steps, all_costs)

# Commands to be implemented after running this file
if __name__ == "__main__":
    # Calling for the environment
    env = Environment()
    # Calling for the main algorithm
    RL = QLearningTable(actions=list(range(env.n_actions)))
    # Running the main loop with Episodes by calling the function update()
    env.after(100, update)  # Or just update()
    env.mainloop()
```

B. SARSA

Executional Code

```
# Importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# Importing function from the env.py
from env import final_states

# Creating class for the SarsaTable
class SarsaTable:
    def __init__(self, actions, learning_rate=0.01, reward_decay=0.9, e_greedy=0.9):
        # List of actions
        self.actions = actions
        # Learning rate
        self.lr = learning_rate
        # Value of gamma
        self.gamma = reward_decay
        # Value of epsilon
        self.epsilon = e_greedy
        # Creating full Q-table for all cells
        self.q_table = pd.DataFrame(columns=self.actions, dtype=np.float64)
        # Creating Q-table for cells of the final route
        self.q_table_final = pd.DataFrame(columns=self.actions, dtype=np.float64)

    # Function for choosing the action for the agent
    def choose_action(self, observation):
        # Checking if the state exists in the table
        self.check_state_exist(observation)
        # Selection of the action - 90 % according to the epsilon == 0.9
        # Choosing the best action
        if np.random.uniform() < self.epsilon:
            state_action = self.q_table.loc[observation, :]
            state_action = state_action.reindex(np.random.permutation(state_action.index))
            action = state_action.idxmax()
        else:
            # Choosing random action - left 10 % for choosing randomly
            action = np.random.choice(self.actions)
        return action

    # Function for learning and updating Q-table with new knowledge
    def learn(self, state, action, reward, next_state, next_action):
        # Checking if the next step exists in the Q-table
        self.check_state_exist(next_state)

        # Current state in the current position
        q_predict = self.q_table.loc[state, action]

        # Checking if the next state is free or it is obstacle or goal
        if next_state != 'goal' or next_state != 'obstacle':
            q_target = reward + self.gamma * self.q_table.loc[next_state, next_action]
        else:
            q_target = reward

        # Updating Q-table with new knowledge
        self.q_table.loc[state, action] += self.lr * (q_target - q_predict)

        return self.q_table.loc[state, action]

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

# Printing the Q-table with states
def print_q_table(self):
    # Getting the coordinates of final route from env.py
    e = final_states()

    # Comparing the indexes with coordinates and writing in the new Q-table values
    for i in range(len(e)):
        state = str(e[i])  # state = '[5.0, 40.0]'
        # Going through all indexes and checking
        for j in range(len(self.q_table.index)):
            if self.q_table.index[j] == state:
                self.q_table_final.loc[state, :] = self.q_table.loc[state, :]

    print()
    print('Length of final Q-table =', len(self.q_table_final.index))
    print('Final Q-table with values from the final route:')
    print(self.q_table_final)

    print()
    print('Length of full Q-table =', len(self.q_table.index))
    print('Full Q-table:')
    print(self.q_table)

# Plotting the results for the number of steps
def plot_results(self, steps, cost):
    #
    f, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)
    #
    ax1.plot(np.arange(len(steps)), steps, 'b')
    ax1.set_xlabel('Episode')
    ax1.set_ylabel('Steps')
```

```
ax1.set_title('Episode via steps')

#
ax2.plot(np.arange(len(cost)), cost, 'r')
ax2.set_xlabel('Episode')
ax2.set_ylabel('Cost')
ax2.set_title('Episode via cost')

plt.tight_layout()  # Function to make distance between figures

#
plt.figure()
plt.plot(np.arange(len(steps)), steps, 'b')
plt.title('Episode via steps')
plt.xlabel('Episode')
plt.ylabel('Steps')

#
plt.figure()
plt.plot(np.arange(len(cost)), cost, 'r')
plt.title('Episode via cost')
plt.xlabel('Episode')
plt.ylabel('Cost')

# Showing the plots
plt.show()
```

Code for Visualization

```
# Importing classes
from env import Environment
from agent_brain import SarsaTable

def update():
    # Resulted list for the plotting Episodes via Steps
    steps = []

    # Summed costs for all episodes in resulted list
    all_costs = []

    for episode in range(1000):
        # Initial Observation
        observation = env.reset()

        # Updating number of Steps for each Episode
        i = 0

        # Updating the cost for each episode
        cost = 0

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

        while True:
            # Refreshing environment
            env.render()

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

            # RL choose action based on next observation
            action_ = RL.choose_action(str(observation_))
            # RL learns from the transition and calculating the cost
            cost += RL.learn(str(observation), action, reward, str(observation_), action_)

            # Swapping the observations and actions - current and next
            observation = observation_
            action = action_

            # Calculating number of Steps in the current Episode
            i += 1

            # Break while loop when it is the end of current Episode
            # When agent reached the goal or obstacle
            if done:
                steps += [i]
                all_costs += [cost]
                break

    # Showing the final route
    env.final()

    # Showing the Q-table with values for each action
    RL.print_q_table()

    # Plotting the results
    RL.plot_results(steps, all_costs)

# Commands to be implemented after running this file
if __name__ == "__main__":
    # Calling for the environment
    env = Environment()
    # Calling for the main algorithm
    RL = SarsaTable(actions=list(range(env.n_actions)),
                    learning_rate=0.1,
                    reward_decay=0.9,
                    e_greedy=0.9)
    # Running the main loop with Episodes by calling the function update()
    env.after(100, update)  # Or just update()
    env.mainloop()
```

C. Iterative SARSA

Executional Code

```
# Importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# Importing function from the env.py
from env import final_states

# Creating class for the Q-learning table
class QLearningTable:
    def __init__(self, actions, learning_rate=0.01, reward_decay=0.9, e_greedy=0.9):
        # List of actions
        self.actions = actions
        # Learning rate
        self.lr = learning_rate
        # Value of gamma
        self.gamma = reward_decay
        # Value of epsilon
        self.epsilon = e_greedy
        # Creating full Q-table for all cells
        self.q_table = pd.DataFrame(columns=self.actions, dtype=np.float64)
        # Creating Q-table for cells of the final route
        self.q_table_final = pd.DataFrame(columns=self.actions, dtype=np.float64)

    # Function for choosing the action for the agent
    def choose_action(self, observation):
        # Checking if the state exists in the table
        self.check_state_exist(observation)
        # Selection of the action - 90 % according to the epsilon == 0.9
        # Choosing the best action
        if np.random.uniform() < self.epsilon:
            state_action = self.q_table.loc[observation, :]
            state_action = state_action.reindex(np.random.permutation(state_action.index))
            action = state_action.idxmax()
        else:
            # Choosing random action - left 10 % for choosing randomly
            action = np.random.choice(self.actions)
        return action

    # Function for learning and updating Q-table with new knowledge
    def learn(self, state, action, reward, next_state):
        # Checking if the next step exists in the Q-table
        self.check_state_exist(next_state)

        # Current state in the current position
        q_predict = self.q_table.loc[state, action]

        # Checking if the next state is free or it is obstacle or goal
        if next_state != 'goal' or next_state != 'obstacle':
            q_target = reward + self.gamma * self.q_table.loc[next_state, :].max()
        else:
            q_target = reward

        # Updating Q-table with new knowledge
        self.q_table.loc[state, action] += self.lr * (q_target - q_predict)

        return self.q_table.loc[state, action]

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

# Printing the Q-table with states
def print_q_table(self):
    # Getting the coordinates of final route from env.py
    e = final_states()

    # Comparing the indexes with coordinates and writing in the new Q-table values
    for i in range(len(e)):
        state = str(e[i])  # state = '[5.0, 40.0]'
        # Going through all indexes and checking
        for j in range(len(self.q_table.index)):
            if self.q_table.index[j] == state:
                self.q_table_final.loc[state, :] = self.q_table.loc[state, :]

    print()
    print('Length of final Q-table =', len(self.q_table_final.index))
    print('Final Q-table with values from the final route:')
    print(self.q_table_final)

    print()
    print('Length of full Q-table =', len(self.q_table.index))
    print('Full Q-table:')
    print(self.q_table)

# Plotting the results for the number of steps
def plot_results(self, steps, cost):
    #
    f, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)
    #
    ax1.plot(np.arange(len(steps)), steps, 'b')
    ax1.set_xlabel('Episode')
    ax1.set_ylabel('Steps')
```

```
ax1.set_title('Episode via steps')

#
ax2.plot(np.arange(len(cost)), cost, 'r')
ax2.set_xlabel('Episode')
ax2.set_ylabel('Cost')
ax2.set_title('Episode via cost')

plt.tight_layout()  # Function to make distance between figures

#
plt.figure()
plt.plot(np.arange(len(steps)), steps, 'b')
plt.title('Episode via steps')
plt.xlabel('Episode')
plt.ylabel('Steps')

#
plt.figure()
plt.plot(np.arange(len(cost)), cost, 'r')
plt.title('Episode via cost')
plt.xlabel('Episode')
plt.ylabel('Cost')

# Showing the plots
plt.show()
```

Code for Visualization

```
# Importing classes
from env import Environment
from agent_brain import SarsaTable

def update():
    # Resulted list for the plotting Episodes via Steps
    steps = []

    # Summed costs for all episodes in resulted list
    all_costs = []

    for episode in range(1000):
        # Initial Observation
        observation = env.reset()

        # Updating number of Steps for each Episode
        i = 0

        # Updating the cost for each episode
        cost = 0

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

        while True:
            # Refreshing environment
            env.render()

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

            # RL choose action based on next observation
            action_ = RL.choose_action(str(observation_))
            # RL learns from the transition and calculating the cost
            cost += RL.learn(str(observation), action, reward, str(observation_), action_)

            # Swapping the observations and actions - current and next
            observation = observation_
            action = action_

            # Calculating number of Steps in the current Episode
            i += 1

            # Break while loop when it is the end of current Episode
            # When agent reached the goal or obstacle
            if done:
                steps += [i]
                all_costs += [cost]
                break

    # Showing the final route
    env.final()

    # Showing the Q-table with values for each action
    RL.print_q_table()

    # Plotting the results
    RL.plot_results(steps, all_costs)

# Commands to be implemented after running this file
if __name__ == "__main__":
    # Calling for the environment
    env = Environment()
    # Calling for the main algorithm
    RL = SarsaTable(actions=list(range(env.n_actions)),
                    learning_rate=0.1,
                    reward_decay=0.9,
                    e_greedy=0.9)
    # Running the main loop with Episodes by calling the function update()
    env.after(100, update)  # Or just update()
    env.mainloop()
```