# Report On

# ME5406
# Deep Learning For Robotics
# Project For Part 1

**Yue Zenglin**

**A0225268B**

**E0575902@u.nus.edu**

**+65 82442113**

# Table of contents

## 1. IMPLEMENTATION AND RESULTS

### 1.1. Environment Introduction

The environment of this program consists of ‘goal’, ‘holes’ and ‘explorer robots’. The program created the environment via ‘Tkinter’ and creating lines and rectangles with different colors to represent locations of holes and robot.

The creation is realized in the maze_env file. There are three main functions: _build_maze(), reset() and step(). The system use _build_maze() to create the environment at the beginning and reset() is used to initialize the location of the explorer in every different episode. In every step, the environment will make judgement about whether the robot has reach the holes or the goal based on the location of explorer, holes and goal. If so, it will return True Boolean value to indicate the completion of current episode, returning the relevant reward and renew the GUI window to let the robot move at the same time.

The ‘create_holes_randomly()’ is designed to arbitrarily generate frozen lake environments based on ‘random.choice()’ function. To test the performance of  different algorithm in a fair way, we created a fixed environment to make comparison in this report.

### 1.2. Monte Carlo Method

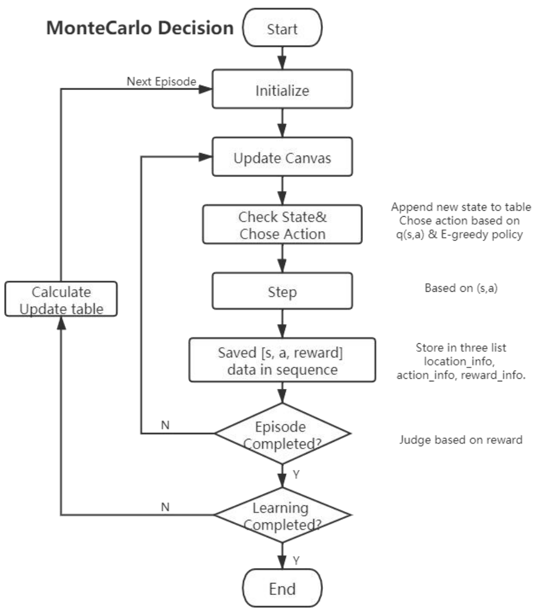
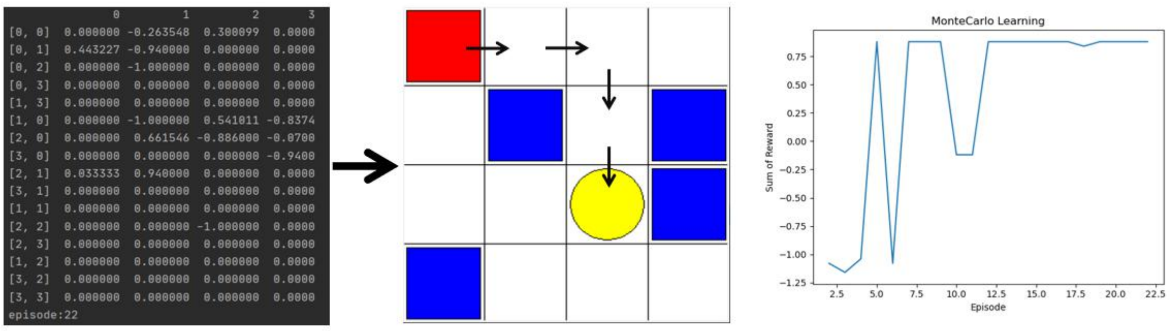

MonteCarlo Decision
Start
Initialize
Next Episode
Update Canvas
Check State& Chose Action
Append new state to table Chose action based on q(s,a) & E-greedy policy
Step
Based on (s,a)
Saved [s, a, reward] data in sequence
Store in three list location_info, action_info, reward_info.
Episode Completed?
Judge based on reward
N
Y
Learning Completed?
N
Y
Calculate Update table
End

**Figure 1-The Principle of Monte Carlo Method**

The Monte Carlo part consists of a value table and a total_times_of_g_value table to store the q value of each state and the overall times of g value. The program take use of the series and Data Frame part in the Panda package to store, renew and get the data.

In every episode, the Monte Carlo brain learns from a sequence of operations consist of creating the environment, initialize the information, check state, chose location, step the robot, collect information, calculate and renew the table after episode completion, reset the data and start the next episode.

A key challenge of Monte Carlo application is that we need to delete repeated state, action and reward. Then, calculate and renew the v table after the completion of current episode according to the information collected in a reversed trend. The program calculated from the last data by using a simple technique: range(len(information)-1,-1,-1). The repeated data is deleted in the first_visit_data_process() function. Details are discussed in the difficulties part.



```
                0         1         2       3
[0, 0]  0.000000 -0.263548  0.300099  0.0000
[0, 1]  0.443227 -0.940000  0.000000  0.0000
[0, 2]  0.000000 -1.000000  0.000000  0.0000
[0, 3]  0.000000  0.000000  0.000000  0.0000
[1, 3]  0.000000  0.000000  0.000000  0.0000
[1, 0]  0.000000 -1.000000  0.541011 -0.8374
[2, 0]  0.000000  0.661546 -0.886000 -0.0700
[3, 0]  0.000000  0.000000  0.000000 -0.9400
[2, 1]  0.033333  0.940000  0.000000  0.0000
[3, 1]  0.000000  0.000000  0.000000  0.0000
[1, 1]  0.000000  0.000000  0.000000  0.0000
[2, 2]  0.000000  0.000000 -1.000000  0.0000
[2, 3]  0.000000  0.000000  0.000000  0.0000
[1, 2]  0.000000  0.000000  0.000000  0.0000
[3, 2]  0.000000  0.000000  0.000000  0.0000
[3, 3]  0.000000  0.000000  0.000000  0.0000
episode:22
```

**Figure 2-Result - Episode in 4*4 Map**

## 1.3. Sarsa Learning Method&Q learning Method

The implementation of this project follow the sequence of the description of the project.

There are little differences lies in Sarsa-learning and Q-learning: In Sarsa learning, the $\varepsilon\text{-}greedy < 1$ and the system update the q table with the same $\varepsilon\text{-}greedy$ strategy, while in Q-learning, it chose action with the guiding of $\varepsilon\text{-}greedy\ policy$ but update the q table with 100% greedy policy.

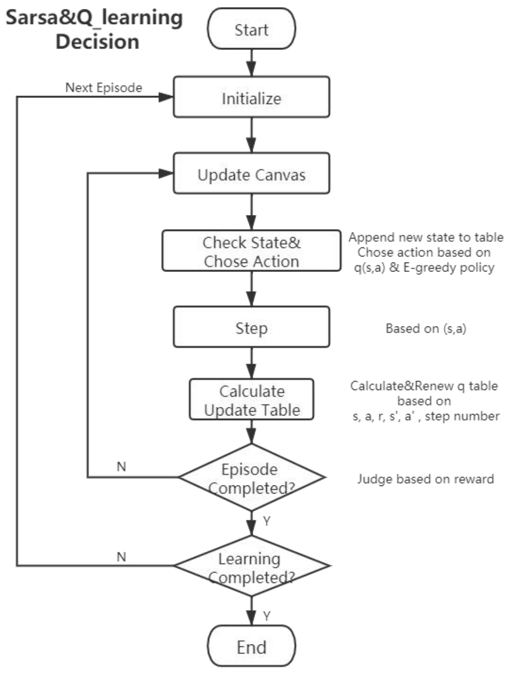

Sarsa&Q_learning Decision
Start
Next Episode
Initialize
Update Canvas
Check State& Chose Action
Append new state to table Chose action based on q(s,a) & E-greedy policy
Step
Based on (s,a)
Calculate Update Table
Calculate&Renew q table based on s, a, r, s', a' , step number
N
Episode Completed?
Judge based on reward
Y
N
Learning Completed?
Y
End

**Figure 3-The Principle of Sarsa and Q learning**

## 2. COMPARE AND CONTRAST

The comparison will be carried out to compare the episodes and time it takes for different algorithms to converge, finding the right way. The way to identified a converge state is depending on the final reward of the algorithms. If the average of the reward in a sequence of 20 episodes over a certain number, the system will label it as a converged state.

In the code, it has two kinds of judging methods. One is judged by the total number of the episodes, the other is judged by the average value of the reward in the episode_end_flag_list list. The length of the list will be depended on the input judge_number in the update() function.

To make our comparison more convicted and receivable, 30 learning process have been carried out in the test file and plotted in the result_display file. Only the performance in 10*10 maps will be discussed cause the 4*4 are too simple for all of the algorithm and the result can be easily be affected by random probability.

### 2.1. 10*10 Map

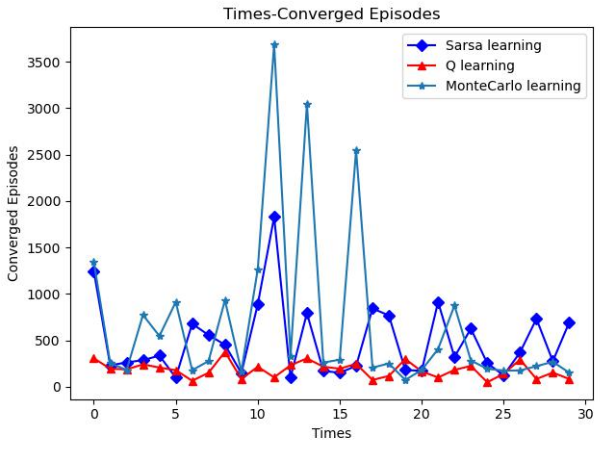


**Figure 4-Episodes Required for Converge in different Methods**

This line graph shows that Q learning takes least episodes to get into a converged state. Sarsa learning is slower than the Q learning but faster than the Monte Carlo learning in most of the time. Monte Carlo learning shows instability and fluctuation over the experiments. In extreme cases, 3000 episodes are needed for Monte Carlo to find the way to the get the goal.

The average running time for Q learning, Sarsa learning and Monte Carlo are 1219.919ms, 5705.537ms and 4399.156ms. Average episodes needed for converge for Q learning, Sarsa learning, Monte Carlo are 181.5, 490.0 and 681.333.

## 3. DIFFICULTIES

During the implement of the project, great efforts have been made to deal with various kinds of bugs and problems, such as how to search helpful information on the internet, use packages, functions and classes to work more efficiently and most important, be patient enough to debug.

### 3.1. A seemingly Smart Dual-Date Frame

In Monte Carlo programme design, pandas and numpy are used to construct data frame of the table. At the beginning stage, data of the Tkinter GUI and state-action frame are separated into two forms, which means the project created a state-action-value table to store the table while using the action to move the agent in the Tkinter environment at the same time. Map function are needed to transform and bridge locations and states.

The state table is like:

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

**Table 1-State of the 4*4 Map**

The action of 'up' will cause the value of state equals state+4. For example, when the current state is 6 and the action is down, the state will be renewed to 6+4 = 10 as the new state while moving the explorer figure into the corresponding place in tkinter. By this mapping strategy, original intention is to use actions to renew two table at the same time, reducing the dimension of the storage of the data so that the algorithm could be simple, clear and easy to handle with.

However, things become difficult and tricky to make corresponding changes in state-action-table with the Tkinter_map table at the same pace, especially in the case when we need to make changes to the index of the list. For example, in the first-visit situation calculation of Monte-Carlo Method in the 10*10 map. To solve this problems, the project give up this Dual-data Frame and create the data frame in the forms below:

The 'Date Frame' in the panda package are incorporated in the system to set the type of data as 'np.float64' to store the q value of each (s,a) state. In the process of the implication, the programme do not create the whole table directly. Instead, it will append (s,a) series value when there is a new (s,a) state emerge. (Mofan, 2020)

| | Up | Down | Right | Left |
|---|---|---|---|---|
| [x1, y1] | 0.0 | 0.0 | 0.0 | 0.0 |
| [x2, y2] | 0.0 | 0.0 | 0.0 | 0.0 |
| [x3, y3] | 0.0 | 0.0 | 0.0 | 0.0 |

**Table 2-Q Value Table of the 4*4 Map**

The project do not use state number 1, 2, 3, 4 to denote the state of the explorer. Instead, it use the coordinate of the explorer to represent the state while representing the location simultaneously. The q(s,a) changes become clear and easy to deal with.

## 3.2. First Visit Data Process in Monte Carlo

It was realized by a data process structure named double pointer algorithm. Basic principle of the algorithm is splitting the whole list into three part. One pointer is forward and the other one is backward. It is a useful technique and can improve the efficiency comparing with a double for loop. And avoiding the index out of the range cause that it start from the last element, even the length of the list have changed, the index is still valid. ("双指针算法_Lynch Blog -CSDN", 2020)

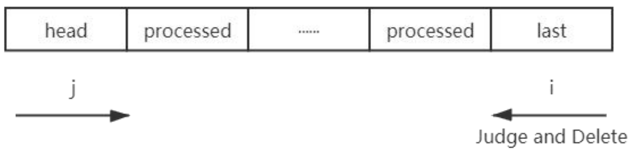


**Figure 5-Double Pointers Algorithm**

## 3.3. Random Environment Construction

The creation of random environment rely on the random.choice() function. The system choose location based on the range of map weight and height. Difficulties lie in how to judge or avoid the completely blocking access to the frisbee.

Extensive efforts have been made to develop measures to solve the problem, and one of the way to avoiding this extreme situation is by setting a special constrains in the recursive holes generation loop.

The relative location generate completely blocking is not the adjacent but the diagonal relationship between two holes. The algorithm can avoid the blocking by not allowing diagonal holes during the generation.

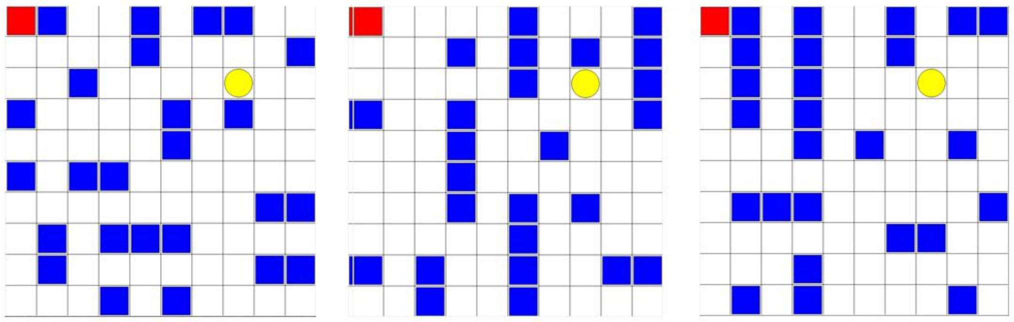


**Figure 6-Environments Generated by No-Diagonal Policy**

However, this is not exactly the random environment to some extend. Another approach is taking use of DFS policy to generate environment. In this approach, the system will not set too much constrains when choosing holes locations. Instead, we carry out DFS process to find all the valid place which the agent can reach from the start point. It is a kind of deep first search recursive process and the agent will follow 'up', 'right', 'left' and 'down' sequence to lab and store all the valid place. The generating process will continue until the total number of valid place satisfied the constrain. ("DFS Princlple Analyze_菜鸟算法-CSDN Blog", 2020)

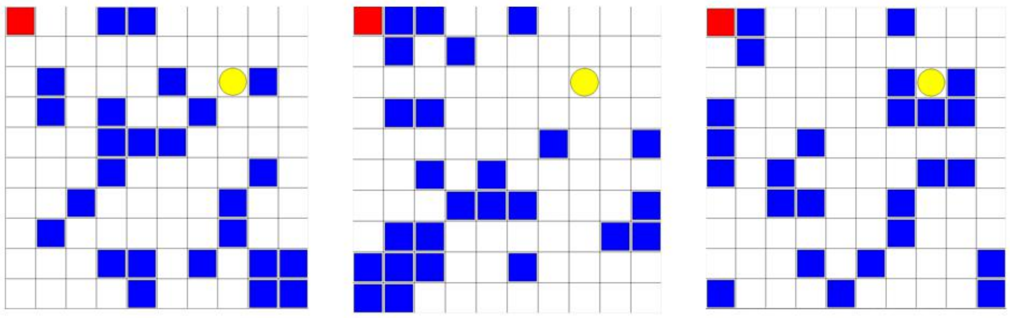


**Figure 7-Environments Generated by DFS Policy**

## 3.4. Little But Serious Details

Some of the function worked in a weird way such as 'range()'. It's usage appears like range(start, end, gap), which can get the first value but can not get the end value. There is a bug cost me a long time: 'range(len(list)), 0, -1)' can't get to 0. It will always stop when the range value equals to 1.

## 3.5. Action Choice in Special Cases

A very tricky situation is when q(s,up) = q(s,down) = q(s,right) > q(s,left). The program have to make sure the action will be the random of up, down, right while don't chose the left action.

Then, we carry out sort action toward the key-value list and select the max value. When there are the same values, the program will judge the whole key-value of every action and add them to the action list so that it can chose random of them.

# 4. INITIATIVES

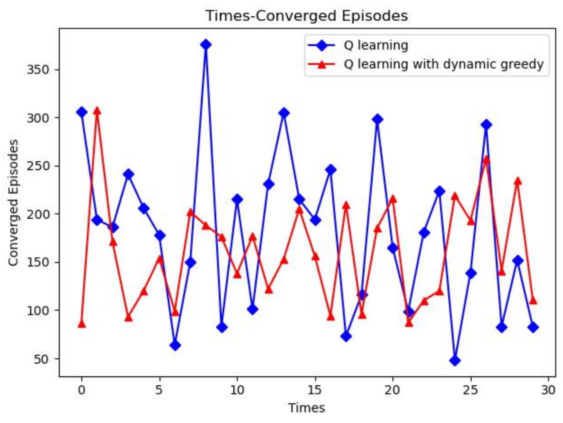
## 4.1. Action Choice in Special Locations

Especially in the edge of the map, the actions which will lead the explorer move outside of the map are meaningless. And there are actually five kinds of action in one state: up, down left, right and stay. Also, the stay is meaningless because it cause no change and if there is a hole nearby, the values of moving outside is likely be evaluated as negative. The strategy tends to stay at the previous location or choose those meaningless actions, which will affect the efficient of the algorithm to a large extent. In this project, invalid action will not be chose.

The algorithm identifies the invalid action by the location of x and y. The 0 x-location indicates that the explorer robot is standing on the edge of the map and any action cause decrease of the x-location is meaningless. The moving left action will be avoided in this situation by deleting it from the action list or setting the key-value to -99, so that the random choice process in the algorithm will not chose turning left action from the action list. (When the key-value is been set to -99, it is far below the normal key-value, which are equal to q(s,a)).

## 4.2. Dynamic Greedy Rate

During the process of exploring, it will take a long time for the agent to find the goal for the first time. After the first successful trial, the agent can find the right way easily rely on the successful experiences get from the previous episode. This observation provides a perspective and shed light on the acceleration of algorithm. It is a reasonable idea to increase greedy rate as the robot finding the way to goal for many times. In the program, change_greedy() is used to change the greedy rate. Every time when it gets the goal, the greedy value will increase 0.0001. In test file, the system carry out 30 experiments in the same condition described before. Average episodes of Sarsa learning and Q learning are 433.1 and 163.3 comparing to the previous 490.0 and 181.5. The converge speed is increased with 10.0275% and 11.61% in Q learning and Sarsa learning process.



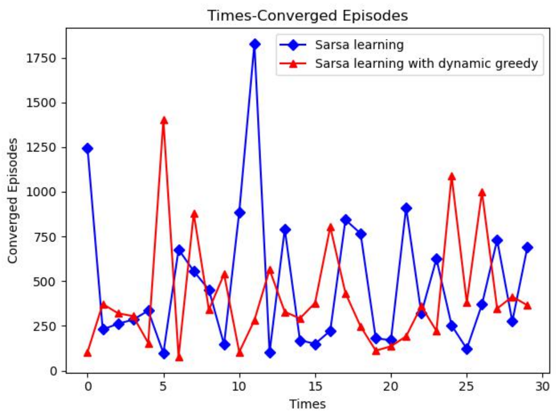


**Figure 8- Converged Result with Dynamic Greedy Policy**

## 4.3. Brave Agent Policy

In some conditions, the agent tend to explore forward then come back and repeated this actions until there is an random action led the agent to other places. This kind of circulation may cause delay and make the algorithm less efficient. In this part, the system is designed to be more brave and the agent tend to explore forward and seldom looking back. The action choice will not only based on the current state, value table, but also based on the past action. Under this kind of brave policy, it will not chose the action which would cause a backward move.

In the test file, the system carry out 30 experiments in the same parameters described before, the average episodes of Sarsa learning and Q learning with brave policy are 491.867 and 253.933 comparing to the previous 490.0 and 181.5. The figure indicate that it may not be a good choice to behave bravely. Hesitation of the agent during the exploration can help the agent to back to the previous places in bad situations, thus enabling the agent to find the right way quickly.

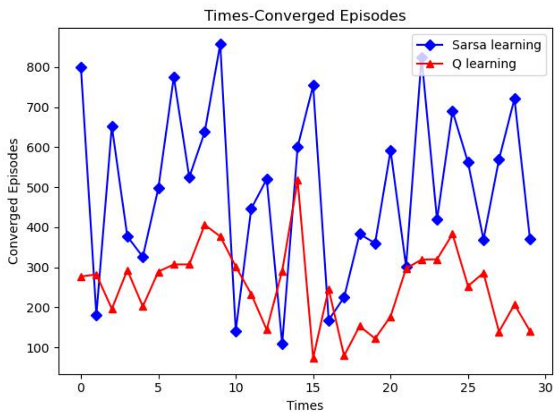


**Figure 9- Converged Result with Brave Policy**

## 5. CONCLUSION

In this project, agent tried to get the goal and avoiding the holes by learning from the q(s,a) experiences get from exploration episodes. Trials have been made to incorporated brave, dynamic greedy elements into algorithm and dynamic greedy shows a positive effect. Frozen lake is a simple problem, while it still takes a long time for agent to learn. More details should be taken into consideration and more powerful policy should be designed to deal with problems in reality. Monte Carlo learning, Sarsa learning and Q learning are far from enough.

## 6. REFERENCES

Akbulut, Y., Sengur, A., Budak, U., & Ekici, S. (2017). Deep learning based face liveness detection in videos. *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*. doi: 10.1109/idap.2017.8090202

Mofan, Z. (2020). Reinforcement Learning. Mofan Python. Retrieved 15 October 2020, from https://mofanpy.com/tutorials/machine-learning/reinforcement-learning/.

双指针算法_Lynch Blog -CSDN. Blog.csdn.net. (2020). Retrieved 15 October 2020, from https://blog.csdn.net/qq_41995258/article/details/90598089.

DFS Princlple Analyze_菜鸟算法-CSDN Blog. Blog.csdn.net. (2020). Retrieved 17 October 2020, from https://blog.csdn.net/li_jeremy/article/details/83714298.

## APPENDIX

### Holes Creating With No-Diagonal Policy

```
def create_holes_randomly(self, number):
    origin = np.array([UNIT * 0.5, UNIT * 0.5])
    # Create origin location in the environment frame.
    holes = []
    global holes_location
    global goal_location
    hell_ = list(range(number))
    for i in range(number):
        list_y = list(range(self.height))
        list_x = list(range(self.width))
        x_location = random.choice(list_x)
        y_location = random.choice(list_y)
        # Because only 25% of the whole place is holes, there is enough room to distribute all the holes
without
        # any adjacent holes. By this assume, we can avoid the situation the completed blocked situation.
        # Avoid the hole appear at the initial location & the goal location
        # Avoid the hole appear at the previous hole locations
        flag = 0
        while (
                [x_location, y_location] == [0, 0] or
                [x_location, y_location] in holes or
                [x_location+1, y_location+1] in holes or
                [x_location+1, y_location-1] in holes or
                [x_location-1, y_location+1] in holes or
                [x_location-1, y_location-1] in holes or
                [x_location, y_location] == goal_set) and flag < 100:
            flag += 1
            # Avoid chose for too many times
            x_location = random.choice(list_x)
            y_location = random.choice(list_y)
        holes.append([x_location, y_location])
        hell1_center = origin + np.array([UNIT * x_location, UNIT * y_location])
        hell_append = self.canvas.create_rectangle(
            hell1_center[0] - size_of_element, hell1_center[1] - size_of_element,
            hell1_center[0] + size_of_element, hell1_center[1] + size_of_element,
            fill='blue')
        holes_location.append(self.canvas.coords(hell_append))
    return 0
```

## Holes Creating With DFS Policy

```
def create_holes_randomly_DFS(self, number):

    global map
    global have_reached
    have_reached = []
    count = 0

    origin = np.array([UNIT * 0.5, UNIT * 0.5])
    # Create origin location in the environment frame.
    holes = []
    global holes_location
    global goal_location

    def dfs(location):
        if location in have_reached:
            return
        have_reached.append(location)
        up = [location[0]-1, location[1]]
        right = [location[0], location[1]+1]
        left = [location[0], location[1]-1]
        down = [location[0]+1, location[1]]
        if (0 <= up[0] <= 9) and (0 <= up[1] <= 9):
            if map[up[0], up[1]] != -1:
                dfs(up)
        if (0 <= right[0] <= 9) and (0 <= right[1] <= 9):
            if map[right[0], right[1]] != -1:
                dfs(right)
        if (0 <= left[0] <= 9) and (0 <= left[1] <= 9):
            if map[left[0], left[1]] != -1:
                dfs(left)
        if (0 <= down[0] <= 9) and (0 <= down[1] <= 9):
            if map[down[0], down[1]] != -1:
                dfs(down)
        return have_reached

    list_y = list(range(self.height))
    list_x = list(range(self.width))

    while True:
        print('世界生成中·········')
        map = np.zeros(100).reshape(10, 10)
        holes.clear()
        have_reached.clear()
        time.sleep(0.1)
# Creating holes for 25
        for i in range(number):
            x_location = random.choice(list_x)
            y_location = random.choice(list_y)
            while (
                    [x_location, y_location] == [0, 0] or
                    [x_location, y_location] in holes or
                    [x_location, y_location] == goal_set
            ):
                x_location = random.choice(list_x)
                y_location = random.choice(list_y)
            holes.append([x_location, y_location])

        for i in holes:
            map[i[1], i[0]] = -1
        have_reached.clear()
        valid_place = dfs([0, 0])
        print('valid place:', valid_place)
        print(map)
        try:
            if len(valid_place) >= 75:
                print('有效长度', len(valid_place))
                print('出循环')
                break
        except:
            print('')
for iii in holes:
        x_location = iii[0]
        y_location = iii[1]
        hell1_center = origin + np.array([UNIT * x_location, UNIT * y_location])
        hell_append = self.canvas.create_rectangle(
            hell1_center[0] - size_of_element, hell1_center[1] - size_of_element,
            hell1_center[0] + size_of_element, hell1_center[1] + size_of_element,
            fill='blue')
        holes_location.append(self.canvas.coords(hell_append))
return 0
```

## Action Choice Policy of Q and Sarsa

```
def choose_action(self, location):
    # action selection
    def chose_again(value):
        action = value
        while action == value:
            action = int(np.random.uniform(0, 4))
        return action

    state_old = str(location)
    if np.random.rand() < self.greedy:
        # choose best action
        # 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)
        # find best policy based on the current table
        action_list = []
        key_value = {}
        key_value[0] = self.locate_table_value(state_old, 0)
        key_value[1] = self.locate_table_value(state_old, 1)
        key_value[2] = self.locate_table_value(state_old, 2)
        key_value[3] = self.locate_table_value(state_old, 3)
        if location[0] == 0:
            key_value[3] = -99

        if location[0] == 9:
            key_value[2] = -99

        if location[1] == 9:
            key_value[1] = -99

        if location[1] == 0:
            key_value[0] = -99

        key_value = sorted(key_value.items(), key=lambda kv: (kv[1], kv[0]))
        action_list.append(key_value[3][0])
        if key_value[3][1] == key_value[2][1]:
            action_list.append(key_value[2][0])
            if key_value[2][1] == key_value[1][1]:
                action_list.append(key_value[1][0])
                if key_value[1][1] == key_value[0][1]:
                    action_list.append(key_value[0][0])
        action = random.choice(action_list)
    else:
        # choose random action to make exploration
        action_list = [0, 1, 2, 3]
        if location[0] == 0:
            del action_list[action_list.index(0)]
        if location[0] == 3:
            del action_list[action_list.index(1)]
        if location[1] == 3:
            del action_list[action_list.index(2)]
        if location[1] == 0:
            del action_list[action_list.index(3)]
        action = random.choice(action_list)
    return action
```