Reinforcement Learning – Homework #2: Grid World Experiments

Indra Imanuel / 20215136

a. **Policy Iteration**

- *environment.py*

  The *environment.py* file define how the grid world (the environment) looks like along with its properties for the agent to interact with. Some important properties defined in *environment.py* includes:

  - The grid world size, in this case it's 5x5
  - The agent possible actions, which are up, down, left, right.
  - Position of the obstacles, represented by triangle, and the goal, represented by circle.
  - 5x5 Reward table, where each value represents the reward for that grid position (state). For example, reward for the triangle obstacle is -1, and for the circle goal is 1.
  - The states, which are the position of the agent x and y, represented in an array [x, y], thus there are 5x5 states in this experiment.
  - User interface for the Grid World.

  There are some functions also in *environment.py*, which are:

  - Getting the next state after an action is taken.
  - Getting the reward based on the new state from taking an action.
  - Getting all of the states of the grid.
  - Checking the boundary so that the agent doesn't go over the grid world borders.
  - Some functions related to the UI.

- *policy_iteration.py*

  The *policy_iteration.py* file define how the agent learn using the policy iteration method, so there are properties related to that, which are :

  - 5x5 state-value table, which define the state-value, v, for each of the grid.
  - 5x5x4 policy table, which define the policy, π, or the probability of the 4 actions (up, down, left, right) in each grid (except for the goal circle grid). Initially, all the probability for the 4 actions are 0.25.
  - The discount factor, γ, which is set to 0.9.

  In this file, there are also function related to the policy iteration method, which are *policy_evaluation* and *policy_improvement*. The *policy_evaluation* function is used to do the policy evaluation, which updates the state-value of each state based on the agent's policy. This works by calculating the state-value $v(s)$ for every possible action in each grid. The policy evaluation function is run every time the user clicks the "Evaluate" button. In the book, it is said that policy evaluation is done continuously until the delta (change in state-value) less than the threshold (small positive number). There's no threshold in the code, instead the user decides when to stop doing the policy evaluation, generally

they should stop when the state-value doesn't change anymore. Below are the steps of policy evaluation based on the code:

1. For all possible states in the environment, do the following:
    a. Initialize state-value for the current state, $v(s) = 0$
    b. Check all possible action in the current state, and for each of that action, do the following:
        i. Get the next state, s', based on the current state s, and action a.
        ii. Get the reward, r, of that next state s'.
        iii. Get the next state-value based on the next state, $v(s')$
        iv. Update the state-value of the current state, $v(s)$, using the formula:
        $$v(s) \leftarrow \pi(a|s)[r + \gamma v(s')]$$
        v. Put the updated state-value, $v(s)$ into the value-table, corresponding to the current state, s.

After doing policy evaluation many times until the state-value doesn't change anymore, the next step is to do policy improvement. Policy improvement is used to improve the policy by finding the actions that give the maximum state-value for each of the grid (state). After finding those actions in each state, the policy is updated for each state so that it has the probability to take those actions. Below are the steps to do policy improvement based on the code:

1. For all possible states in the environment, do the following:
    a. Initialize *value* variable, with value -9999, a temporary list *max_index*, and a temporary array *result*, to store all of the probabilities of taking all of possible actions in a certain state (The probabilities are all initialized as 0). These variables will be used to help finding the maximum state-value, $v(s)$.
    b. Check all possible action in the current state, and for each of that action, do the following:
        i. Get the next state, s', based on the current state s, and action a.
        ii. Get the reward, r, of that next state s'.
        iii. Get the next state-value based on the next state, $v(s')$.
        iv. Calculate the state-value for the current state and store it in a *temp* variable, with the formula: $v(s) \leftarrow r + \gamma v(s')$.
        v. Compare the variable *temp* with the variable *value*. If *temp* is equal to *value*, store the current action into the temporary list *max_index*, else if the variable *temp* is greater than *value*, clear the temporary list *max_index* and store the current action into it. This will result in *max_index* having actions that maximize the state-value $v(s)$. The code allows the agent to choose more than 1 action that maximize the $v(s)$. The probability of choosing the said actions is distributed evenly.
    c. Update the temporary array *result* to only have the probability of taking the action that maximize the $v(s)$. If there are more than 1 action, the probability is distributed evenly.
    d. Update the policy table with the variable *result*, corresponding to the current state s.

The policy improvement function is run when the user clicks on "Improve" button. After the user does the policy evaluation many times until the state-value for each grid doesn't change, and then do policy improvement, the result can still be not good. In that case, the user has to do policy evaluation many times until the state-value for each grid doesn't change again, then run the policy improvement again.

The optimal result should be like in Figure 1. We can see in the figure that the arrows direct the agent (square) to move to the goal (circle), while avoiding the obstacles (triangle).

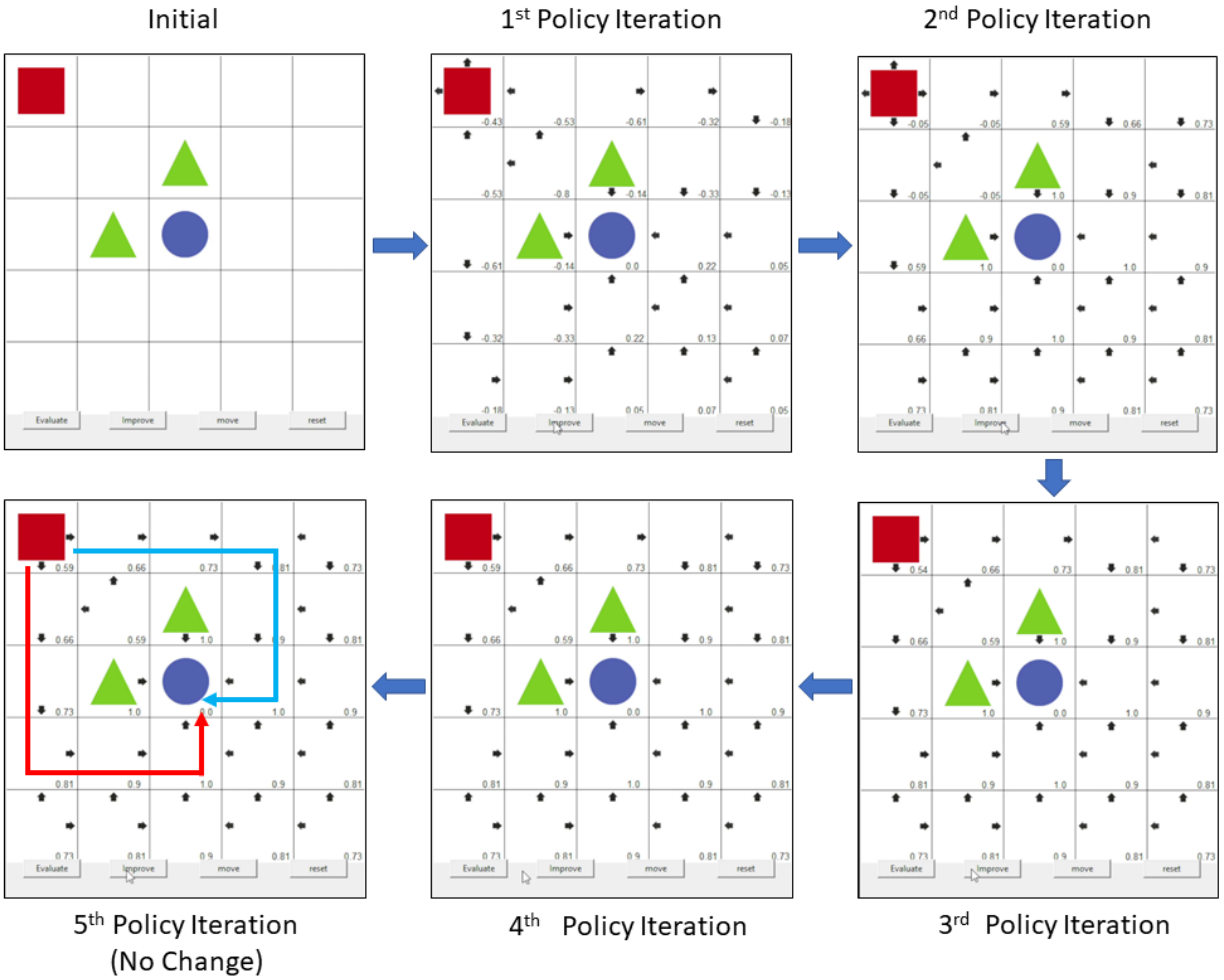

Figure 1. Policy Iteration Method in Grid World Experiment.

We can see from Figure 1, that after the 4th policy iteration, the state-value of each state (grid) stops updating. This means that the optimal policy is found). Although, in the 3rd policy iteration, the policy is actually already close to optimal, and the agent can reach the goal too in the 3rd policy iteration. The agent managed to finds 2 optimal routes to reach the goa. The routes are indicated with blue and red arrow.

b. **Value Iteration**

- *environment.py*
  The *environment.py* file for value iteration is mostly the same with the one in policy iteration.
- *value_iteration.py*
  The *value_iteration.py* file has similar properties and function as the *policy_iteraton.py* file. The difference is that in this file, there's no policy table and instead of *policy_evaluation* and *policy_iteration* function, there is *value_iteration* function.

  The *value_iteration* function follows the pseudo-code for value iteration in the book. The value iteration method removes the need of having policy evaluation and policy improvement, as it doesn't really update the agent's policy iteratively, hence there isn't a policy table in the code. Instead it calculates the state-value $v(s)$ for each possible action in every state, and choose the maximum state-value for every state. The agent will then just go to the direction that has maximum state-value.

  Similar with policy evaluation, in the pseudo-code for value iteration, it should've stopped when the state-value delta is less than the threshold. But since there's no threshold in the code, the user has to repeatedly click the "Calculate" button, which will run the *value_iteration* function, until the state-value for all the state doesn't change anymore. Then, the user can click "Print Policy" button, to show the policy of each state. Below are the steps of doing value iteration based on the *value_iteration* function in the code:

  1. For all possible state in the environment, do the following:
     a. Initialize a temporary list *value_list*.
     b. Check all possible action in the current state, and for each of that action, do the following:
        i. Get the next state, s', based on the current state, s, and action, a.
        ii. Get the reward in that next state s'.
        iii. Get the next state-value based on the next state, $v(s')$
        iv. Update the state-value for the current state, $v(s)$, using the formula: $v(s) \leftarrow r + \gamma v(s')$.
        v. Store the updated value in the temporary list *value_list*.
     c. The temporary list *value_list* is used to store all possible updated $v(s)$. Since $v(s)$ is updated based on s' and reward in that s', the updated value could have different value based on the action taken, a, in the current state s. As we are checking all possible action a in the current state s, it is guaranteed that we will have multiple value of the state-value $v(s)$. The $v(s)$ with the maximum value will be chosen and stored to the state-value table, corresponding to the current state s.

  By running the above steps multiple times (by pressing the "Calculate" button multiple times until the state-value ,$v(s)$, of each state doesn't change anymore), the agent will eventually find the optimal or close to optimal route to the goal. The agent learning process and result can be seen in Figure 2.

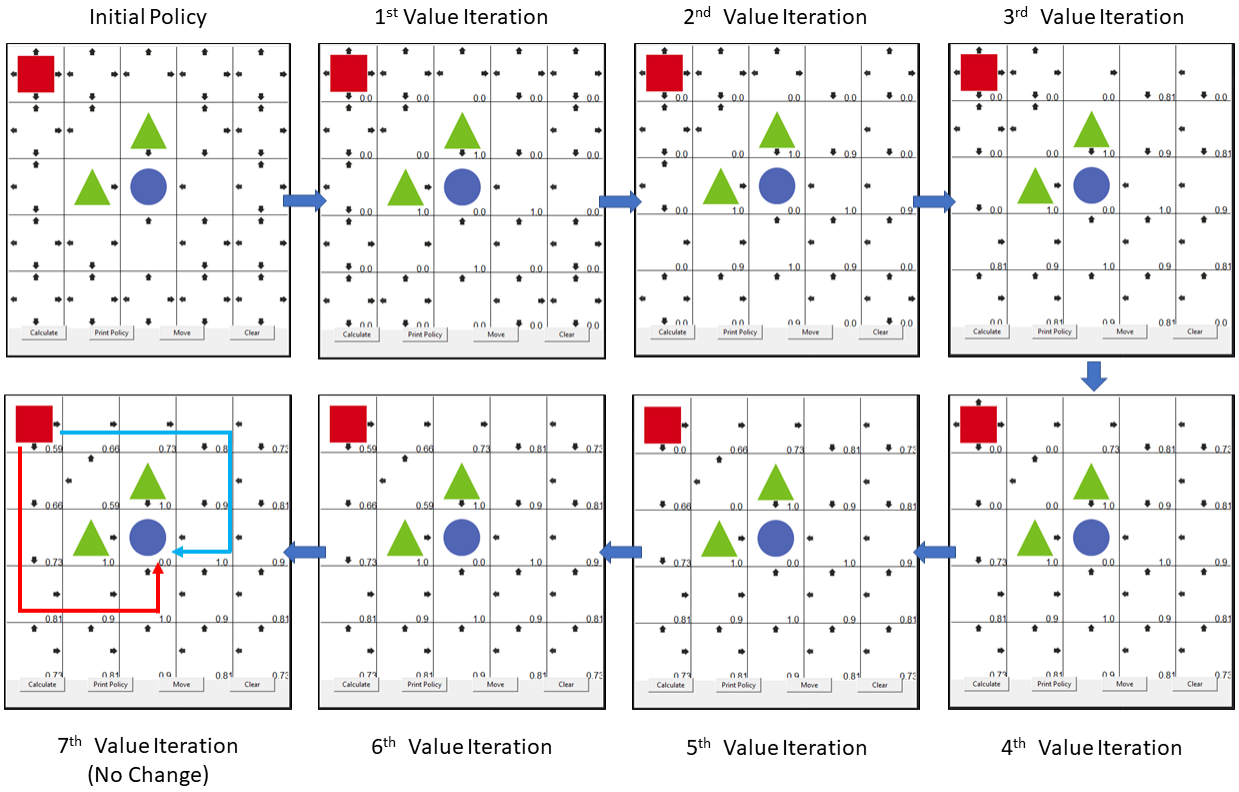

Figure 2. Value Iteration Method in Grid World Experiment.

From Figure 2, we can see that it takes 6 value iteration for the agent to find the optimal policy. We can see that there are no change in state-value for each grid between the 6th and 7th value iteration, indicating the optimal policy was found in the 6th value iteration. Just like policy iteration method, the agent found two optimal routes to reach the goal, indicated by the blue and red arrow.

c. **Monte Carlo**

- *environment.py*

  The *environment.py* file inside the Monte Carlo method contains the reward system definition, where reward +100 when agent reaches goal, reward -100 when agent got into an obstacle, and reward 0 for any other place. It also defines that an episode ends when the agent reaches the goal or got into an obstacle. Other than that, this file contains mostly about the UI behavior, such as loading the image assets, moving the agent, and other UI related things. The main function in this file is "step", where the function translates the action to agent movement. The reward system is also defined in the "step" function.

- *mc_agent.py*

  The *mc_agent.py* file contains the Monte Carlo agent learning and movement related properties and function. The properties that are related to the Monte Carlo learning process include:

  1. Learning rate (α), which is set to 0.01, determines how fast/ big of a jump the agent updates the state-value $v(s)$.
  2. Discount factor (γ), which is set to 0.9, for discounting the reward
  3. Exploration rate (ϵ), which is set to 0.1, to determine the probability of the agent doing exploration or exploitation in taking action. The agent will generate a random number between 0 and 1 everytime it wants to take an action, it'll compare the random number with ϵ. If the number is lower than ϵ, the agent will do exploration and take a random action. On the other hand, if the random number is greater than ϵ, the agent will do exploitation and take action with the maximum state-value $v(s)$ in that state. This strategy is called the epsilon greedy strategy.
  4. Samples Memory, a list that is used to store all the states, along with the states' reward, that the agent has visited.
  5. 5x5 State-value table, to store the state-value $v(s)$ of every state.

  The *mc_agent.py* file follows the First-Visit Monte Carlo method from the book. Although, in the code implementation, the formula of the return, G, is different with the book. Also, instead of using the average G to update the $v(s)$ like in the book, the code uses incremental update rule instead (As explained in Chapter 2.4 from the book). The Monte Carlo agent learns for a maximum of 1000 episode, with maximum of 100 steps in each episode. In each episode the agent follows these steps:

  1. The agent takes an action based on the state its currently in with the epsilon greedy strategy.
  2. Based on the action taken, the agent moves to the next state while getting the reward in that next state.
  3. The agent saves the next state, s', and the reward it gets in the next state, r, to the sample memory.
  4. The agent get the next action, a', based on the next state (using the epsilon greedy strategy again).
  5. Repeat step 2 until 4 as long as the episode hasn't been terminated yet. The termination condition was if the agent reaches the goal or hit an obstacle.
  6. When the episode is terminated, update the agent using the sample memory. This is done by unrolling the sample memory from the most recent to the oldest one. At the start of update process, before unrolling the sample memory, initialize return, G, as 0.

7. Remember, sample memory contains a series of state and its corresponding reward that the agent has visited. These states might have duplicate, but since this is First-visit Monte Carlo, during the updating process, the agent will only visit each of the state in the sample memory only once. The agent will only consider the first time it visits those states, resulting in no duplicate states visited during the update process. For every state in the unrolling process, do the following:
    a. Update the G with the formula $G \leftarrow \gamma \times (r + G)$
    b. Update the state-value function with the formula $V(s) \leftarrow V(s) + \alpha\big(G - V(s)\big)$, and put that updated value to the state-value table.

In the code, the episode was set to 1000, but around 50 episodes are enough for the agent to find the optimal or close to optimal route to the goal. The Monte Carlo learning process and result can be seen in Figure 3.

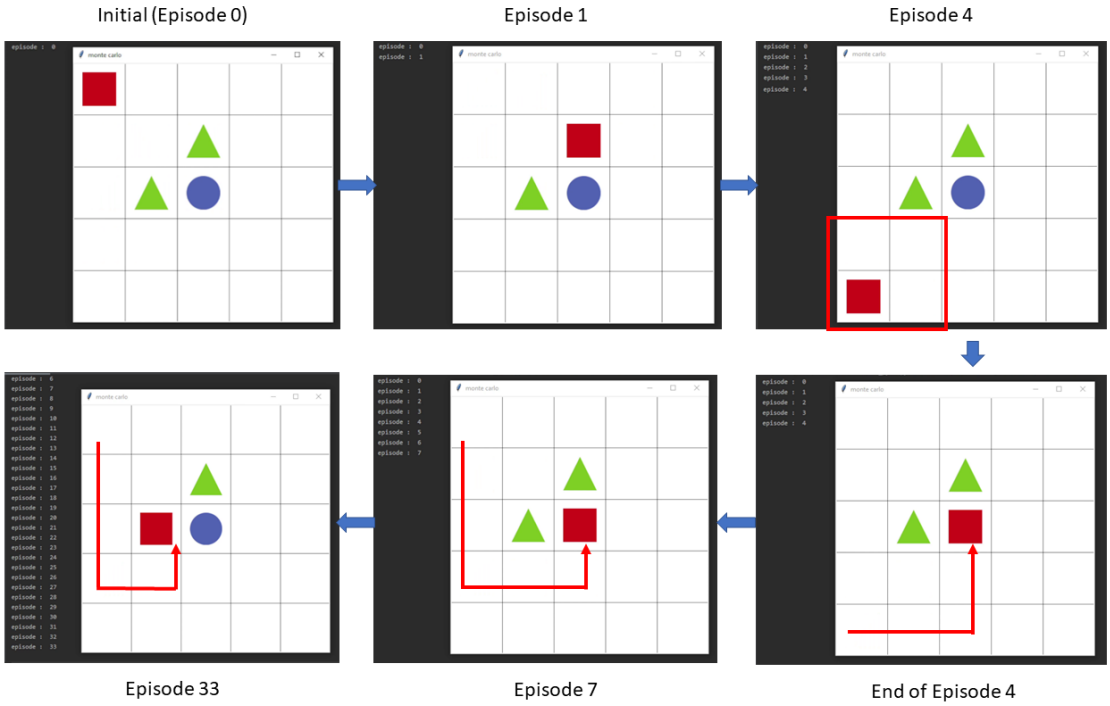

Figure 3. Monte Carlo Method for Grid World Experiment.

In Figure 3, it is shown that the agent was stuck in Episode 4, where the agent would move around in circle in the red box area. The agent could get stuck for a very long time, in the code there's no termination condition when this happen, so I added the condition that the agent could only take a maximum of 100 steps during each episode. This termination condition is also applied to the SARSA and Q-Learning experiment, as it can happen in those methods as well. In this case though, the agent was actually able to get out of the area it was stuck in, and actually managed to reach for the goal by the end of episode 4. By episode 7, the agent was starting to get consistent in reaching the goal, shown by the red arrow. The agent follows this route again and again in the following episode, although sometimes it deviates to other direction due to random movement made by doing exploration. This route that the agent learned is actually one of the optimal routes, as shown in the Policy Iteration and Value Iteration method.

d. **SARSA**

- *environment.py*

  The *environment.py* file inside the SARSA method is more or less the same with the Monte Carlo one. It contains the same reward system definition, and episode termination condition. It also contains UI related behavior function and the function "step" to translate action into agent's movement.

- *sarsa_agent.py*

  The *sarsa_agent.py* file contains the SARSA agent learning and movement related properties and function. Some important properties that are related to the SARSA agent learning process includes:

  1. Learning rate (α), which is set to 0.01, to determine how fast/big of a jump the Q-value should be updated.
  2. Discount factor (γ), which is set to 0.9, to discount the reward.
  3. Exploration rate (ϵ), which is set to 0.1, to determine the probability of the agent doing exploration or exploitation in taking the action using the epsilon greedy strategy. For the exploitation one, different with Monte Carlo method that takes the maximum state-value $v(s)$ from the state-value table, the SARSA agent will take the maximum action-value (Q-value) from the Q-table.
  4. 5x5x4 Q-table to store every Q-value state-action pair in every state.

  The *sarsa_agent.py* follows the SARSA on-policy method from the book. SARSA method works by estimating action-value (Q-value). SARSA uses the current state (s), action (a), reward in the new state (r), next state (s'), and next action (a') as samples to learn the Q-value, forming the abbreviation "s, a, r, s', a'", hence the name SARSA method. The learning process and steps of SARSA method based on the code is explained in the next paragraph.

  The SARSA agent learning process can be mostly seen in the *main* and *learn* function in the code. Based on the code, the agent learns for a maximum of 1000 episode, with maximum of 100 steps in each episode. The SARSA agent repeats the steps below in each episode :

  1. The agent takes an action based on its current state, using the epsilon greedy strategy.
  2. Based on the action taken, the agent will move to the next state, and get the reward for that state. After that, the agent will take an action again (with epsilon greedy strategy again), but this time, it's based on the next state.
  3. Now, we have the old state (s), action taken on the old state (a), reward gotten in the new state (r), the next state (s'), and action taken on the next state (a'). The agent will use these parameters to learn.
  4. To learn, first the agent needs to lookup the Q-table to get the Q-value of the old state-action pair ($q(s, a)$), and the Q-value of the next state-action pair ($q(s', a')$).
  5. After getting $q(s, a)$ and $q(s', a')$, we can perform the Q-value update with the formula $q(s, a) \leftarrow q(s, a) + \alpha[r + \gamma q(s', a') - q(s, a)]$.
  6. We then put the updated Q-value, $q(s, a)$, into the Q-table with the corresponding state s, and action a.
  7. Then, we set the next state and action as the current state and action.
  8. We repeat step 2 until 7 as long as the episode is not terminated yet.

By repeating the above steps for a number of episodes, the agent will eventually be smart enough to find the optimal, or at least close to optimal, path to the goal. We can see the agent learning process and result in Figure 4.

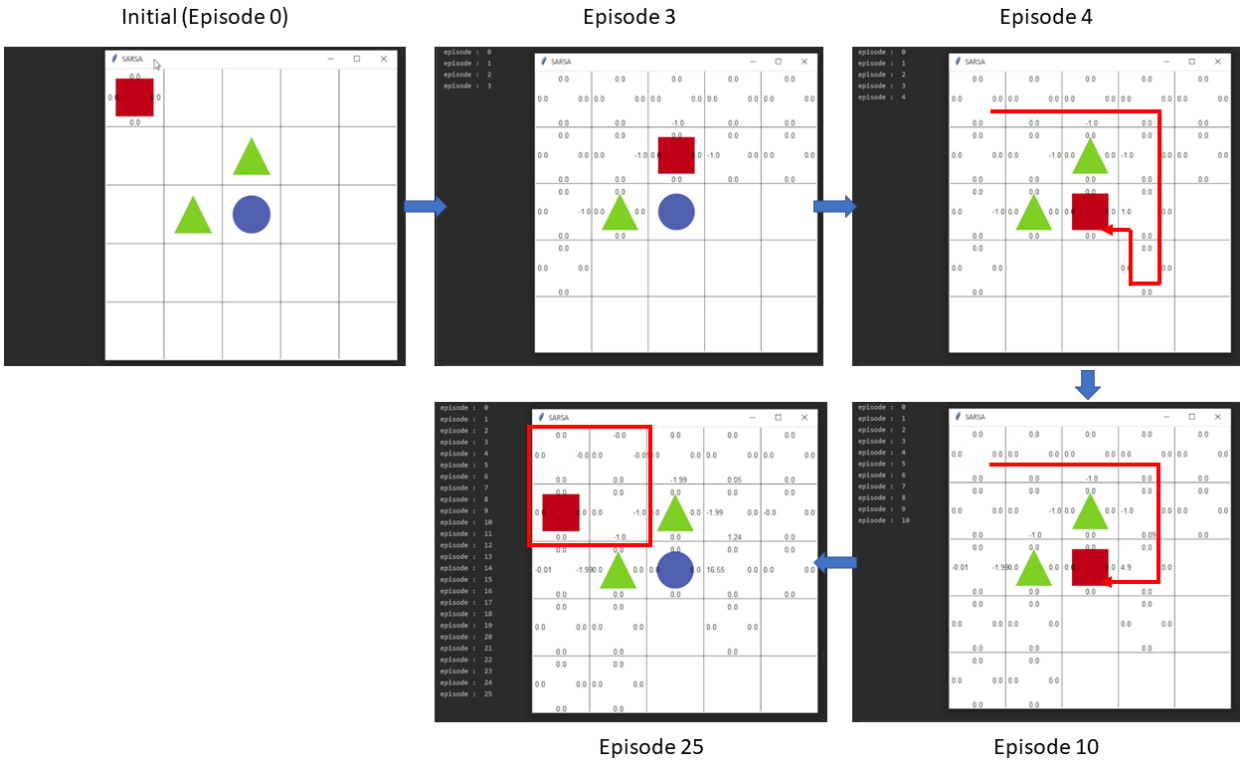

Figure 4. SARSA Method for Grid World Experiment.

From Figure 4, we can see that the agent was able to reach the goal for the first time in Episode 4. In the previous Episode, the agent would go almost randomly, sometimes hitting an obstacle, as it hasn't learned for enough episode yet, as shown in the Episode 3 image. By episode 10, the agent finds one of the optimal routes, and use that route for the following episode as well. But, as more episode goes on, SARSA agent tends to get stuck in an area of the world, as shown in the Episode 25 image.

e. **Q-Learning**

- *environment.py*

  The content of *environment.py* of the Q-Learning method is more or less the same with the ones in Monte Carlo and SARSA method. It contains the same reward system definition and episode termination condition. UI related behavior function and the function "step" to translate action into agent's movement is also here.

- *q_learning_agent.py*

  The *q_learning_agent.py* file contains properties and function related to the Q-Learning agent learning process. The properties in this file is the same with the ones in SARSA, which are the learning rate (α), discount factor (γ), exploration rate (ϵ), and 5x5x4 Q-table. Those properties also have the same value with the ones in SARSA.

  The *q_learning_agent.py* follows the Q-learning off-policy method from the book. The Q-learning method, similar to SARSA, also works by updating the action-value (Q-value). The parameters used and the steps to update the Q-value is a little bit different with SARSA. The learning steps and process will be explained in the next paragraph.

  The Q-learning agent learning process can be seen in the *main* and *learn* function in the code. Based on the code, the agent is trained for a maximum of 1000 episodes, with maximum of 100 steps in each episode. The Q-learning agent repeats these steps for each episode:

  1. The agent takes an action based on the current state using the epsilon greedy strategy.
  2. The agent gets the next state and reward in that next state based on the action taken.
  3. Now that the agent has the old state (s), old action (a), reward from the next state (r), and next state (s'), the agent can use these parameters to learn.
  4. Get the Q-value of the old state-action pair ($q(s,a)$) from the Q-table.
  5. Update the Q-value using Bellman Optimality Equation:
     $q(s,a) \leftarrow q(s,a) + \alpha\left[r + \gamma \max_a q(s',a) - q(s,a)\right]$
  6. Put the updated Q-value into the Q-table with the corresponding state s, and action a.
  7. Set the next state as the current state.
  8. Step 1 until 7 is repeated as long as the episode hasn't been terminated yet.

  The above steps are repeated for 1000 episodes, which will result in the agent getting smarter and smarter until it can find the optimal, or close to optimal, path to the goal. We can see from the steps above that Q-learning and SARSA are quite similar, but notice that the parameters used to update the Q-value in Q-learning is only, "s, a, r, s'" as opposed to SARSA's "s, a, r, s', a'". The a' in SARSA is used to get the Q-value of the next state-action pair, $q(s', a')$. But in Q-learning, the Q-value of the next state-action pair is obtained by finding the maximum action for the next-state Q-value, thus Q-learning method doesn't need the parameter next action a'. The agent learning process and result can be seen in Figure 5.

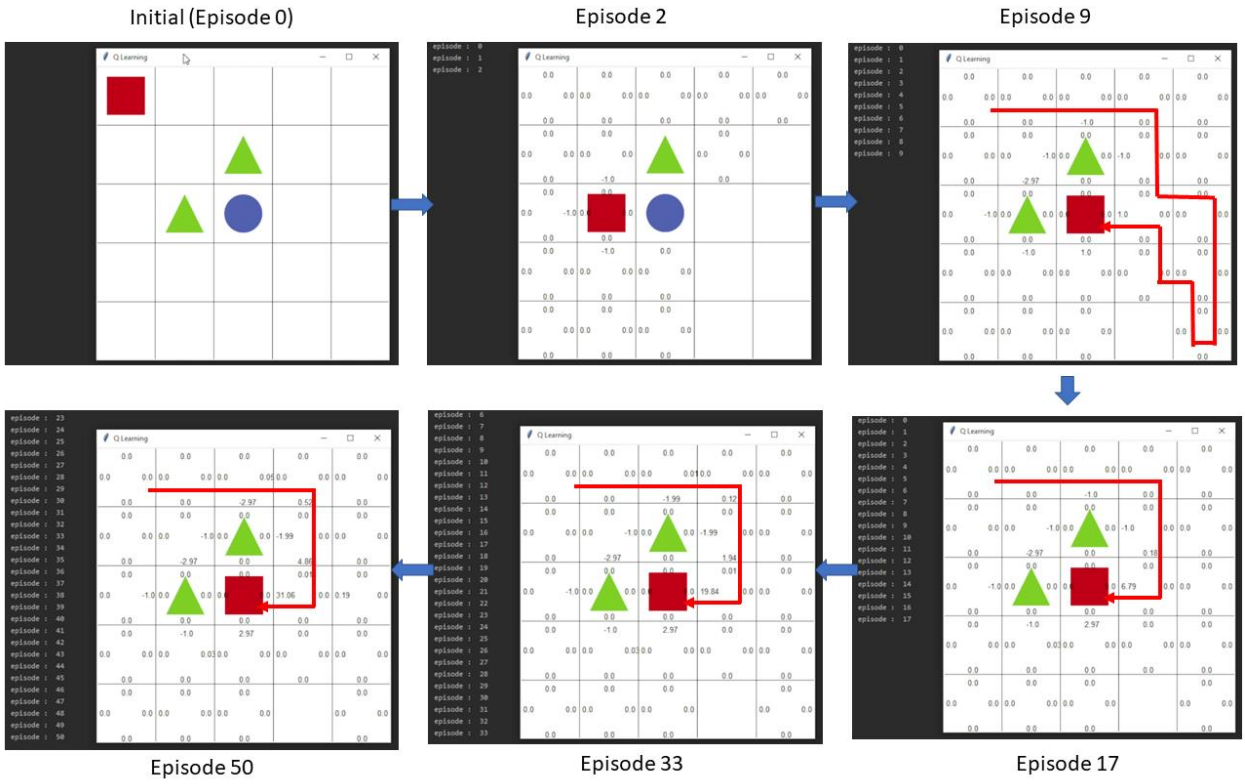

Figure 5. Q-learning Method for Grid World Experiment.

We can see from Figure 5 that, same with the other method, the Q-learning agent also does poorly in the early episode, as shown in the Episode 2 image where the agent hits an obstacle. As the episode goes on however, just like the other methods, the Q-learning agent gets smarter. We can see that by Episode 9, the agent managed to reach the goal, although the route isn't the optimal route. But from there, the agent explored some other states and managed to get more information, which eventually leads the agent into finding the optimal route. The agent was able to find the optimal route and consistently use that route to reach the goal by Episode 17. We can also see that the agent, unlike in the SARSA method, doesn't get stuck in a certain area of the world, where by Episode 33, and even to 50, the agent still able to use the optimal route to the goal.