Q-learning Agent Pseudo-code

The Q-learning algorithm was taken from page 131 of the second edition of 'Reinforcement Learning: An Introduction' by Richard S. Sutton and Andrew G. Barto, which can be found at http://incompleteideas.net/book/the-book.html and which is shown below in Figure 1. I have also rewritten this algorithm in Figure 2 so that my changes are a little easier to follow.

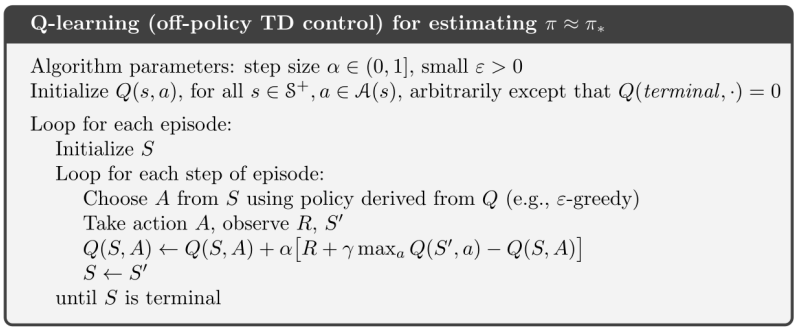

Figure 1: The original Q-learning algorithm.

Require: $\alpha \in (0, 1]$ and $\epsilon > 0$
1: $Q(s, a) \leftarrow 0, \quad \forall s, a$
2: **for all** episodes **do**
3: $S \leftarrow$ starting state
4: **repeat**
5: $A \leftarrow \epsilon$-greedy choice from $Q(S, \cdot)$
6: Take action A, observe R, S'
7: $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
8: $S \leftarrow S'$
9: **until** S is terminal
10: **end for**

Figure 2: The original Q-learning algorithm rewritten.

With the agent learning from the experiences (transitions) of a number of nodes in a graph, another loop needs to be added to iterate though each node of the graph. Because the next state of a node and its next best action can be dependant on the state of its neighbours, the action of each node should be carried out together before learning from any of these actions. Currently, this is achieved with two loops as shown in Figure 3. In the first loop (line 6), an action is selected for each node. Then all the actions are performed (line 9). After this, the second loop (line 10) learns from the transitions which resulted from each node's action.

Require: Learning rate: $\alpha \in (0, 1]$ and discount factor: $\gamma \in (0, 1]$
1: $Q(s, a) \leftarrow 0, \quad \forall s, a$
2: G is a graph of N nodes
3: **for all** episodes **do**
4: $S^t \leftarrow$ the N starting states of the nodes
5: **for** $t = 0$ **to** deadline **do**
6: **for all** $n \in G$ **do**
7: $A_n^t \leftarrow \epsilon$-greedy choice from $Q(S_n^t, \cdot)$
8: **end for**
9: Find S^{t+1}, R^t from S^t, A^t, G
10: **for all** $n \in G$ **do**
11: $Q(S_n^t, A_n^t) \leftarrow Q(S_n^t, A_n^t) + \alpha[R^t + \gamma \max_a Q(S_n^{t+1}, a) - Q(S_n^t, A_n^t)]$
12: **end for**
13: **end for**
14: **end for**

Figure 3: The adapted Q-learning algorithm.

The code for the main training loop is shown in Figure 4. Lines 6, 9 and 10 of Figure 3 corresponds to lines 7, 17 and 20 of Figure 4 respectively. Action selection is performed by the `QLearningAgent`'s `choose_epsilon_greedy_action` method shown in Figure 5. At the moment the node's state is made up of only the time step and the node's normalised fitness (the fitness after being normalised but before the monotonic function), which is why these are the two values passed to the method.

The environment's `run_time_step` performs the set action for each node and finally the agent learns from each node's experience using its `learn` method. This method is passed the time step and node's normalised fitness before and after the transition, as well as the action which it took and fitness resulting from this action (this is after the monotonic function). This fitness is used as a reward in the final time step. The code for this method is shown in Figure 6. This `learn` method calls a private `_update_q_table` calculates the equation on line 11 of the pseudo-code in Figure 3.

```
    for episode in range(1, train_env_config["episodes"]):
        if not episode % save_interval:
            agent.save(suffix=episode)

        # subsequent time steps
        for time in range(deadline):
            for node in range(num_nodes):
                # choose action to be taken by this node at this time
                action = agent.choose_epsilon_greedy_action(
                        time,
                        environment.get_node_fitness_norm(node, time),
                        )
                # set selected action
                environment.set_action(node, time, action)

            # run time step of environment
            environment.run_time_step(time)

            # learn from the last transition
            for node in range(num_nodes):
                agent.learn(
                        time,
                        environment.get_node_fitness_norm(node, time),
                        environment.get_node_action(node, time),
                        time +1,
                        environment.get_node_fitness_norm(node, time +1),
                        environment.get_node_fitness(node, time +1),
                        )


        # decay epsilon for the next episode
        agent.decay_epsilon()
        # generate a new fitness function and reset for the next episode
        environment.generate_new_fitness_func()
        # resets all environment traces to zero
        # and sets new random initial positions
        environment.reset()
```

Figure 4: The main training loop from train.py (lines 58 – 94).

```
    def choose_epsilon_greedy_action(self, time, current_fitness_norm):
        """Returns either the best action according the Q table
        or a random action depending on the current epsilon value.
        """
        current_state = self._find_state(time, current_fitness_norm)
        # choose action
        if random.rand() <= self.epsilon:
            action = random.choice(self.possible_actions)
        else:
            action = self.possible_actions[
                np.argmax(self._q_table[current_state])
            ]
        return action
```

Figure 5: QLearningAgent.choose_epsilon_greedy_action from agents/qlearning.py (lines 76 – 88).

```
    def learn(self, prior_time, prior_fitness_norm,
              chosen_action, post_time, post_fitness_norm, post_fitness):
        """Learns from a transition."""
        prior_state = self._find_state(prior_time, prior_fitness_norm)
        post_state = self._find_state(post_time, post_fitness_norm)

        # each node only receives a reward at the deadline
        if post_time == self.deadline:
            reward = post_fitness
        else:
            reward = 0

        self._update_q_table(prior_state, chosen_action, post_state, reward)
```

Figure 6: QLearningAgent.learn from agents/qlearning.py (lines 94 – 106).

```
    def _update_q_table(self, prior_state, action_num, post_state, reward):
        """Updates the Q table using the given transition."""
        # the action index of the best action in the post transition state.
        post_best_action_idx = np.argmax(self._q_table[post_state])
        td_target = reward \
            + self.discount * self._q_table[post_state][post_best_action_idx]

        action_idx = self.action_num2idx[action_num]
        td_delta = td_target - self._q_table[prior_state][action_idx]

        self._q_table[prior_state][action_idx] += self.learning_rate * td_delta
        self._update_count[prior_state] += 1
```

Figure 7: QLearningAgent._update_q_table from agents/qlearning.py (lines 58 – 69).