COMP 4106 - Assignment 3

Joshua Harris - 101091864

Michael Ting - 101068936

April 14, 2021

Implementation Description

There are a few core functions which are used in the td_qlearning class to produce our optimal qvalue. These are the init function, qvalue, policy, rewardAt, qValueSetter, maxAllActions, and actionPossible. The init function is the main one, the rest either retrieve the result or help the init calculate and optimize towards the Q-value.

Init Function

```
for i in range(len(self.csvInput)): # loop through trajectories
    state = self.csvInput[i][0]
    action = self.csvInput[i][1]
    qv=0.0
    if i < (len(self.csvInput) - 1):
        if state in  qvalues.keys():
            if action in qvalues[state].keys() and self.actionPossible(state, action): # if state and actions already exsists
                    qv = self.qvalue(state,action) + alpha*(self.rewardAt(state) + gamma*self.maxAllActions(self.csvInput[i+1][0]) - self.qvalue(state,action))  # calculate Q value
                    self.qValueSetter(state,action,qv)  # add them to exsiting list

            elif self.actionPossible(state, action):  # state exsits but action doesn't
                    qv = self.qvalue(state,action) + alpha*(self.rewardAt(state) + gamma*self.maxAllActions(self.csvInput[i+1][0]) - self.qvalue(state,action))  # calculate Q value
                    qvalues[state][action] = [qv]
        else: # state not in exist already
            qvalues[state] = {}
            for acts in allActions:
                if self.actionPossible(state,acts):     # make all relevant actions that is possible in that squre
                    qvalues[state][acts] = [0.0]
                if acts == action:
                    qv = self.qvalue(state,action) + alpha*(self.rewardAt(state) + gamma*self.maxAllActions(self.csvInput[i+1][0]) - self.qvalue(state,action))  # calculate Q value
                    qvalues[state][action] = [qv]
```

The init function is responsible for the setup and calculation of the q-values for each state action pair. Our setup is a nested dictionary of qvalues, setup as a dictionary of states, with action q-value pairs. This will later let us return and retrieve any desired state-action pair value. The init function finds the optimal value by looping through all trajectories, if our state doesn't exist already, we make a new state in our dictionary and loop through the possible actions and calculate our q-value. If it already exists, we check if we have state and actions already and we recalculate and update the q-value for the actions, if not, we add and calculate the actions.

Qvalue and Policy Functions

```
def qvalue(self, state, action):     #this is a getter
    if state in qvalues.keys() and action in qvalues[state].keys():
        return max(qvalues[state][action])
    else:
        return 0

def policy(self, state):    # state is a string representation of a state
    # Examines all the actions for that state, returns maxmim Q value for a the state action pair ( dependent on Q value)
```

These functions can be called to retrieve the qvalue and policy respectively for a given state, or state action input.

RewardAt, qValueSetter, maxAllActions, actionPossible functions

```
def rewardAt(self, state):  #Returns the reward at a given state
    reward = 0
    for s in state[1:]:
        reward += int(s)
    return -reward

def qValueSetter(self, state, action, qval):  #add new q value to list for that action. not sure if implementation right
    qvalues[state][action].append(qval)
    return

def maxAllActions(self, state):
    highest= 0.0
    if state in qvalues.keys():
        for actionss in qvalues[state]:  # for actions in  in this states qvalues
            highestAct= max(qvalues[state][actionss])  # get the max vaalue
            if highestAct > highest:
                highest = highestAct

    return highest
```

```
def actionPossible(self,state, action):
    possible = False
    square = int(state[0])
    if square == 1:
        if action == "D" or action == "C":
            possible = True
    elif square == 2:
        if action == "R" or action == "C":
            possible = True
    elif square == 4:
        if action == "L" or action == "C":
            possible = True
    elif square == 5:
        if action == "U" or action == "C":
            possible = True
    elif square == 3:
        if action == "R" or action == "C" or action == "L" or action == "D" or action == "U" :
            possible = True
    else:
        print("not valid action")
    return possible
```

All the remaining functions are helper functions for the init function. rewardAt() returns a reward for a given state. qValueStter() adds a new q value to list for a given action. maxAllActions() returns the maximum from a set of state actions. actionPossible returns if a given action is possible or not, soif in square one, only D and C are possible actions.

Implementation Questions

1. What type of agent have you implemented?

   The agent we have implemented is a goal based agent, the goal is solely to clean the floor. Since we have preconditioned data, we don't have any other goals besides clearing the floor and optimising towards that goal.

2. Suppose there is a state-action pair that is never encountered during a trajectory through state space. What should the Q-value be for this state-action pair?

   The Q-value would be 0 since we assume a starting value of 0 and never update it.

3. For some cases (even with a long trajectory through state space), the optimal Q-value for a particular state-action pair will not be found. Explain why this might be the case.

   The optimal Q-value might never be found in the case that all the squares are perpetually clean and never turned dirty, the AI will never optimise towards an optimal Q-value since the AI is never moving and is in its terminal state. Another potential case where we might not find an optimal Q-value could be if there were infinite states, since

our Q-value is only able to apply across a discrete (limited) trajectory space, we would never be able to store all the output and calculate it.

4. In the test cases provided, the trajectories through state space were generated using random policy. Describe a different strategy to generate trajectories compare it to using a random policy.

    A different policy strategy could be a repeating pattern of visiting each square in our graph and cleaning. This would be starting at square one, down then clean, then going to square two left then clean, then back to the middle, and repeating until we're all the way around our graph back to tile 1.