|  |  |
| --- | --- |
| SWARM SOFTWARE TASK DOCUMENTATIONS - 2022 |  |
|  |  |
| NAME: MITADRU DATTA  ROLL: 21AE10024 | See the source image |

# TASK 1 ON REINFORCEMENT LEARNING

# Policy: First of all, “**intelligence” in RL is the capacity of the agent to select the appropriate strategy in relation to its goals. A policy is, therefore, a strategy that an agent uses in pursuit of goals**. The policy dictates the actions that the agent takes as a function of the agent’s State and the Environment.

# In a Markov Decision Process, which consists of States(S), Action(A), Probability of transition(P) and corresponding Reward(R), a policy(π) is the overall suggested sequence of actions (which generally leads to some positive Reward R > 0).

# If the policy directly refers to the *specific* action, the agent should take at a state, then it is deterministic policy. The more general case is that of a stochastic policy where the policy assigns probabilities to every action in every state, like

# P (state1 || action1) =0.4

# An example of probability distribution of a 4-action stochastic policy (sum of the bars is unity as expected)

# 

* Value Function: This represents the value for the agent to be in a particular state.
* The state value function describes the expected return from a given state. So, .
* The action value function of a state is the expected return if the agent chooses action a according to a policy .

So, .

This value function thus helps in the assessment of the quality of different policies.

* Bellman Equation: A fundamental property of the value functions is that they satisfy some recursive relations known as Bellman equation.

V(s)= MAXa(R(s,a)+ γV(s’))

s => current state || s’ => next state || V=> value || R=> reward

γ => Discount Factor

Discount factor(γ): determines how much the agent cares about rewards

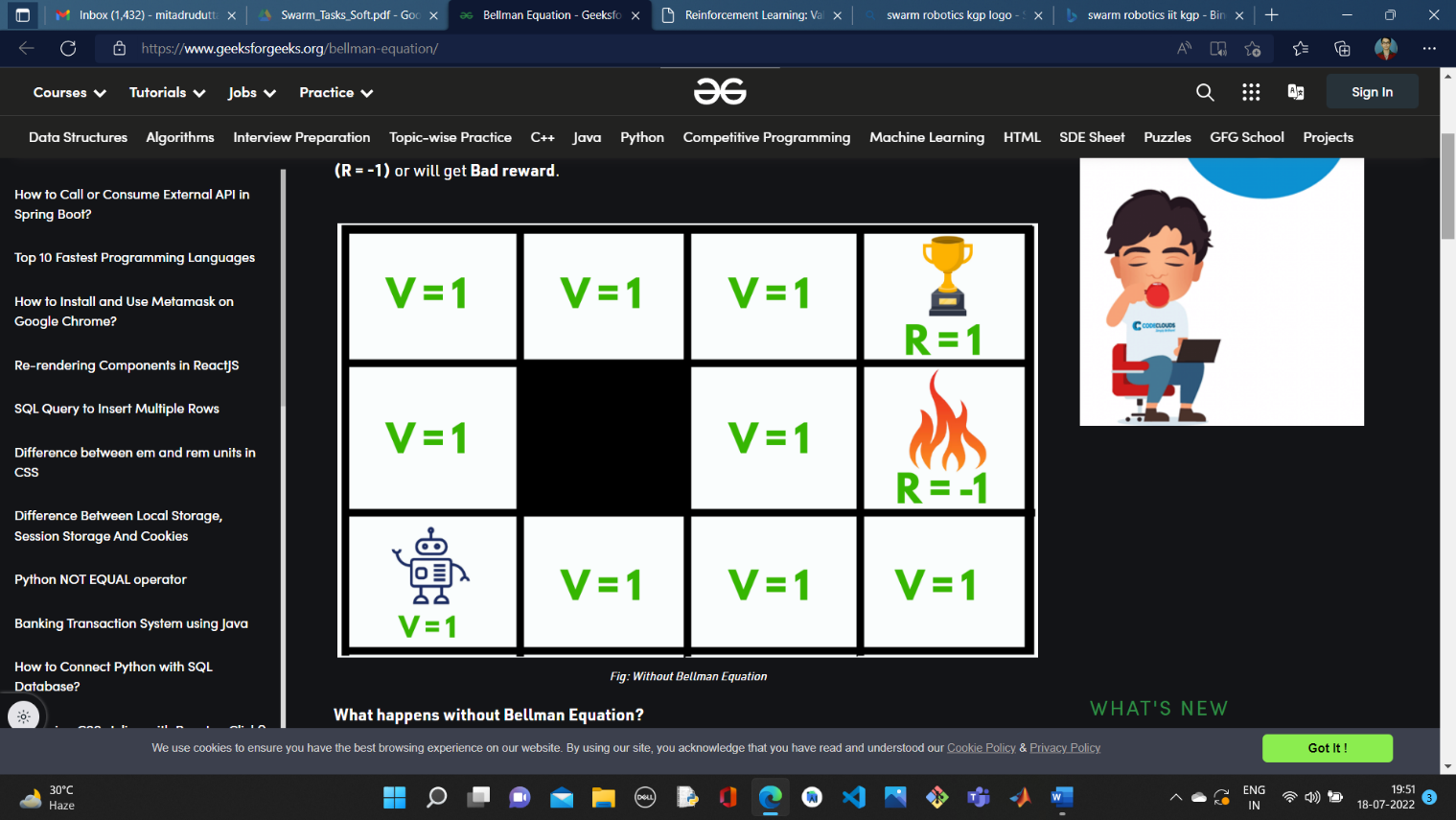
in the distant future relative to those in the immediate future. It has a

value between 0 and 1. Lower value encourages short–term rewards

while higher value promises long-term reward.

An example of the usefulness of Bellman Equation:

Let us consider this maze problem where the agent wants to reach the trophy avoiding the fire.

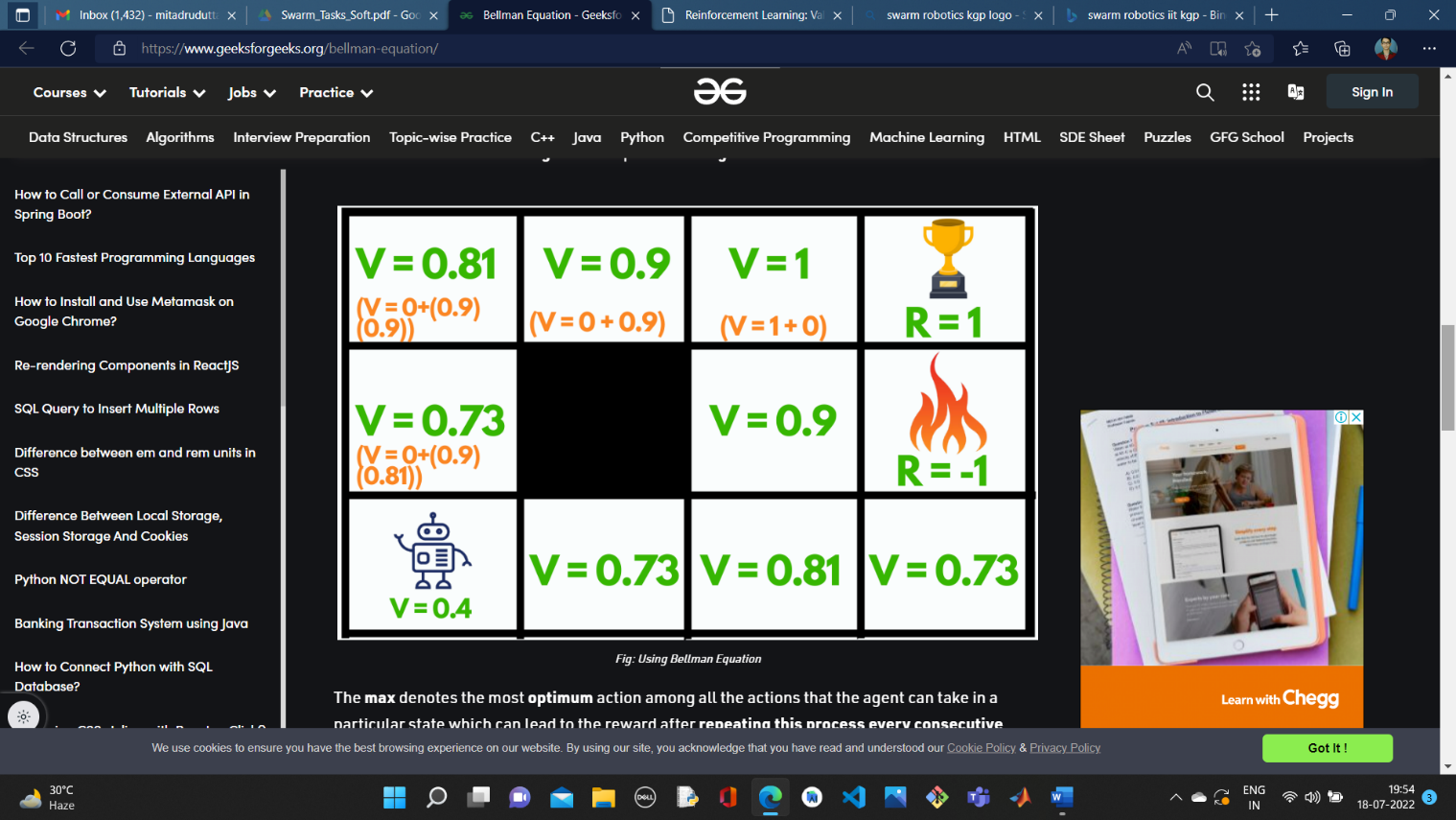


*Image source : Geeks for Geeks*

The above picture shows the condition without the Bellman Equation. Initially, we will give our agent some time to explore the environment and let it figure out a path to the goal. As soon as it reaches its goal, it will back trace its steps back to its starting position and mark values of all the states which eventually leads towards the goal as V = 1.

The agent will face no problem until we change its starting position, as it will not be able to find a path towards the trophy state since the value of all the states is equal to 1. So, to solve this problem we should use Bellman Equation.

After using the Bellman Equation:



So, after making such a plan our agent can easily accomplish its goal by just following the increasing values.

* Temporal Difference Learning:

Temporal difference (TD) learning refers to a class of [model-free](https://en.wikipedia.org/wiki/Model-free_(reinforcement_learning)) [reinforcement learning](https://en.wikipedia.org/wiki/Reinforcement_learning) methods which learn by [bootstrapping](https://en.wikipedia.org/wiki/Bootstrapping_(statistics)) from the current estimate of the value function. These methods sample from the environment. TD methods adjust predictions to match later, more accurate, predictions about the future before the final outcome is known.

The tabular TD (0) method:

It estimates the [state value function](https://en.wikipedia.org/wiki/Reinforcement_learning#Algorithms_for_control_learning) of a finite state Markov Decision Process (MDP) under a policy {\displaystyle \pi }.

The algorithm starts by initializing a table {\displaystyle V(s)}  arbitrarily, with one value for each state of the MDP. A positive [learning rate](https://en.wikipedia.org/wiki/Learning_rate) {\displaystyle \alpha } is chosen. We then repeatedly

evaluate the policy {\displaystyle \pi }, obtain a reward**r** {\displaystyle r}and update the value function for the old state using the rule:

is known as the TD target.

* Q- Learning and SARSA techniques:

Q-Learning technique is an Off Policy technique and uses the greedy approach to learn the Q-value. SARSA technique, on the other hand, is an On Policy and uses the action performed by the current policy to learn the Q-value.  
This difference is visible in the difference of the update statements for each technique:

Q-Learning:

SARSA:

Here, the update equation for SARSA depends on the current state, current action, reward obtained, next state and next action. This observation lead to the naming of the learning technique as SARSA stands for State Action Reward State Action which symbolizes the tuple (s, a, r, s’, a’).

Having these knowledge, now I write a well commented and markdowned code on jupyter notebook. Please find the jupyter notebook with proper outputs.

THE PROBLEM STATEMENT:

Consider the 15 × 10 grid world shown in Figure 1. We want to move our agent, denoted by the light blue square, to the goal which is shown as the green square. However there are certain obstacles in the world, denoted by dark grey squares which our agent cannot cross. The agent can move in four

directions, that is there are four actions possible in each state, A = up, down, right, left, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. The agent also incurs a reward of R = −1 for each step it takes, except when the step leads to the goal, in which case it gets a reward R = 0. Also take the discount factor γ = 1, that is consider the whole reward of the future states while calculating action values.

The agent has to face another adversary, that is a strong gale. The areas affected by the wind areshaded in blue and the direction is specified by the arrows. When the agent moves in the affected region the resultant next states are shifted rightwards by the “wind,” the strength of which varies from row to row (denoted by the number of arrows). For example, if you try to move up in the wind, your resultant motion with diagonally up and right. Also note that the wind is not constant and only affects our agent 80% of the time.

THE INITIALIZATIONS

states=[]

learning\_rate=0.001

e\_value=0.2

discount\_factor = 1

state\_values = [['X' if [j,i] in world.obstacles else -1 for i in range(world.WORLD\_WIDTH)] for j in range(world.WORLD\_HEIGHT)]

state\_values

THE FUNCTION WHICH DETERMINES THE ACTION TO CHOOSE:

def whichAction(current\_state,e\_value):

    action = -1 # a random value

    max\_reward = -1 # the general case for the reward

    num = random.uniform(0,1) #selects a random floating point number in between 0 to 1

    final\_state=current\_state

    if num > e\_value :

        #performing a greedy action

        i=0

        for new\_action in possible\_actions:

            i+=1

            new\_state,new\_reward = world.step(current\_state,new\_action)

            if new\_reward> max\_reward:

                max\_reward,action = new\_reward,new\_action

            elif (i==4)and (max\_reward==-1): #if the greedy action makes no progress, we select a random value

                action = random.choice(possible\_actions)

        final\_state = world.step(current\_state,action)[0]

        while(final\_state == current\_state):

            # this while loop ensures that the steps where the agent stays in the same cell is skipped

            # this would improve space complexity

            action = random.choice(possible\_actions)

            final\_state = world.step(current\_state,action)[0]

    else:

        action = random.choice(possible\_actions)

        final\_state = world.step(current\_state,action)[0]

        while(final\_state == current\_state):

            # this while loop ensures that the steps where the agent stays in the same cell is skipped

            # this would improve space complexity

            action = random.choice(possible\_actions)

            final\_state = world.step(current\_state,action)[0]

    return action

A FUNCTION TO PERFORM THE ACTION :

def performAction(state,action):

    return world.step(state,action)

THE MODEL TRAINING FUNCTION:

def training(current\_state,training\_iter):

    print("Starting state is ",current\_state)

    for i in range(training\_iter):

        if current\_state != world.GOAL : #unless goal reached

            #accessing the global values from the initializations cell

            global states

            global learning\_rate

            global e\_value

            global state\_values

            new\_action = whichAction(current\_state,e\_value) #selects new action

            new\_state = performAction(current\_state,new\_action)[0] #performs the new action

            states.append(new\_state)

            #printing the performed action

            print("In iteration number ",i+1,": ")

            print("Now the agent has reached ",new\_state," by taking the action ",new\_action)

            current\_state = new\_state

        else: # if goal reached

            print("GAME OVER!")

            #backtracing the values

            reverse\_states = list(reversed(states))

            reward = 0

            state\_values[world.GOAL[0]][world.GOAL[1]] =reward

            for state in reverse\_states:

                if state==world.GOAL:

                    continue

                print("updating ",[state[0],state[1]],end=" ")

                prev\_state = reverse\_states[reverse\_states.index(state)-1]

                # applying the temporal difference formula TD(0)

                # v(s)⇐v(s)+α(r+γv(s')-v(s))

                reward = state\_values[state[0]][state[1]]

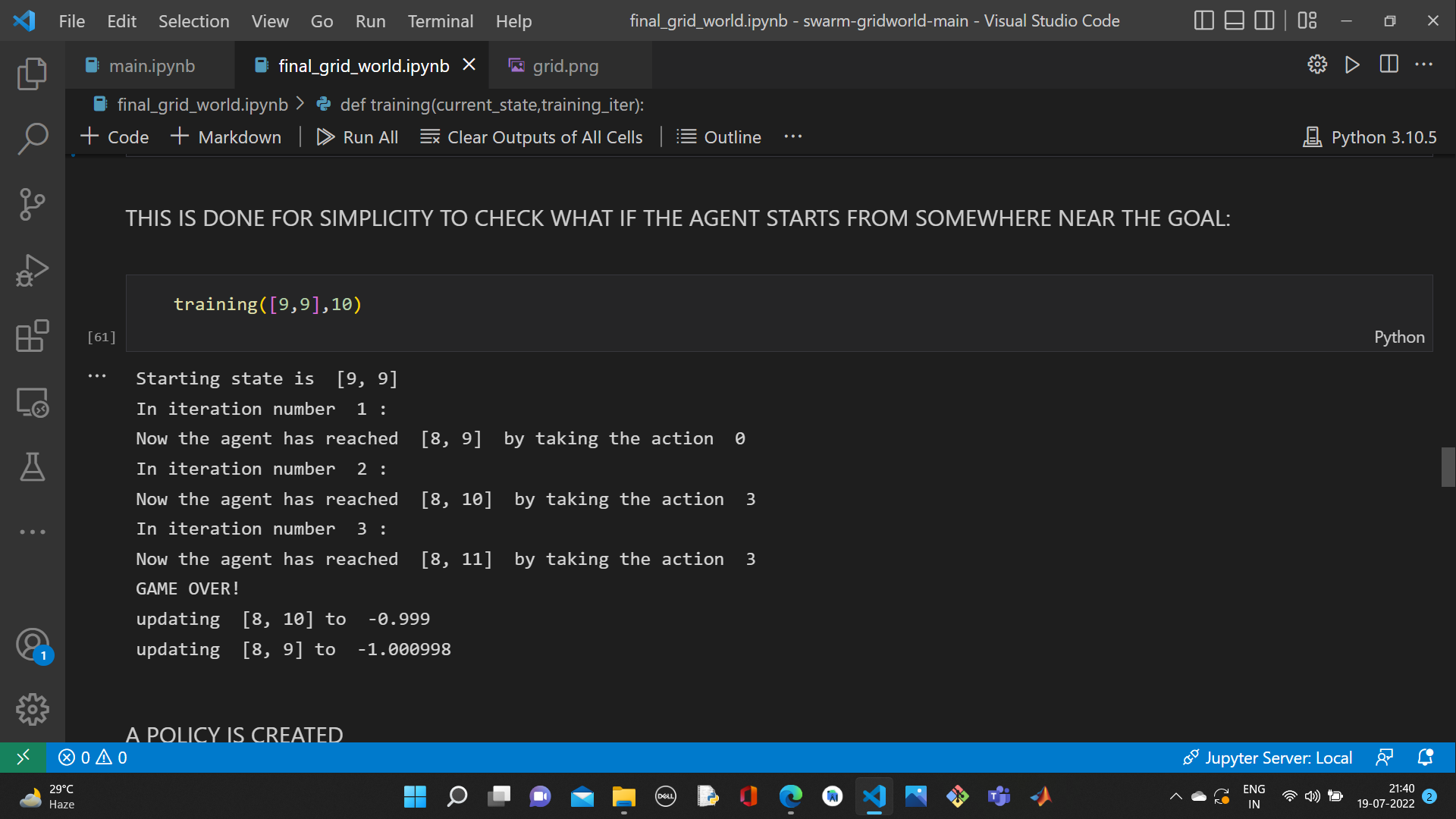
                + learning\_rate\*(reward + discount\_factor\*state\_values[prev\_state[0]][prev\_state[1]] - state\_values[state[0]][state[1]])

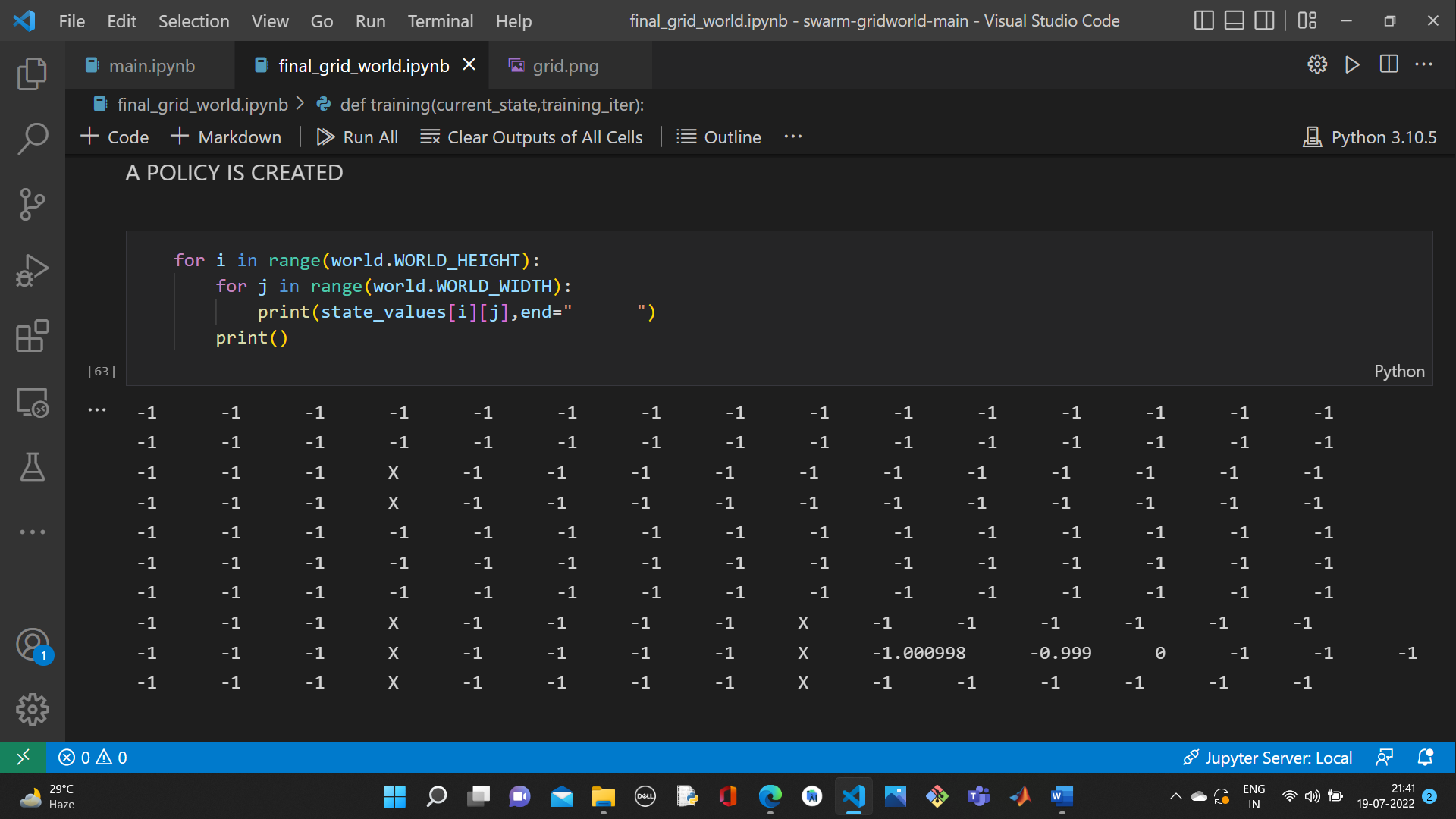
                state\_values[state[0]][state[1]] = reward

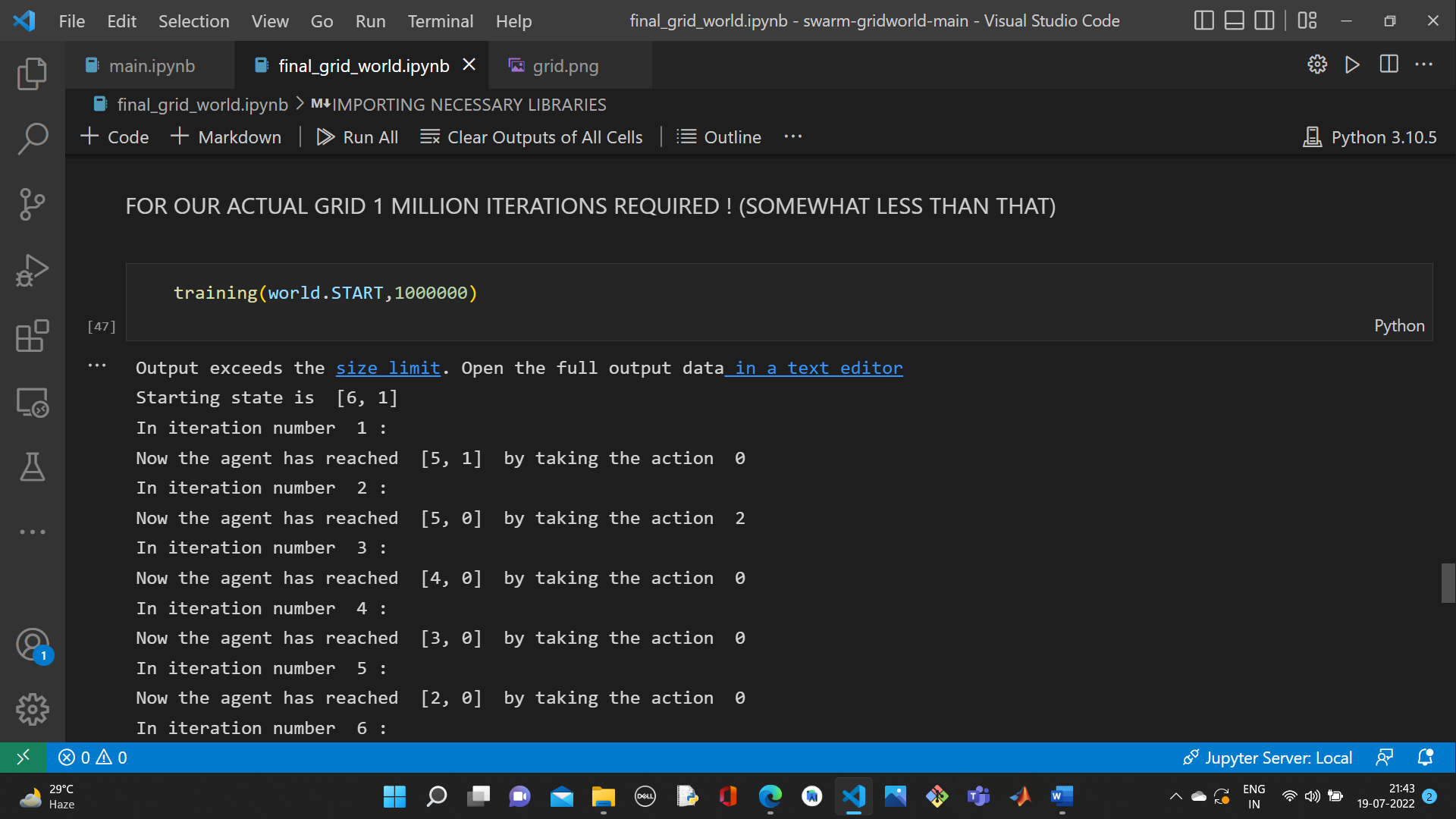
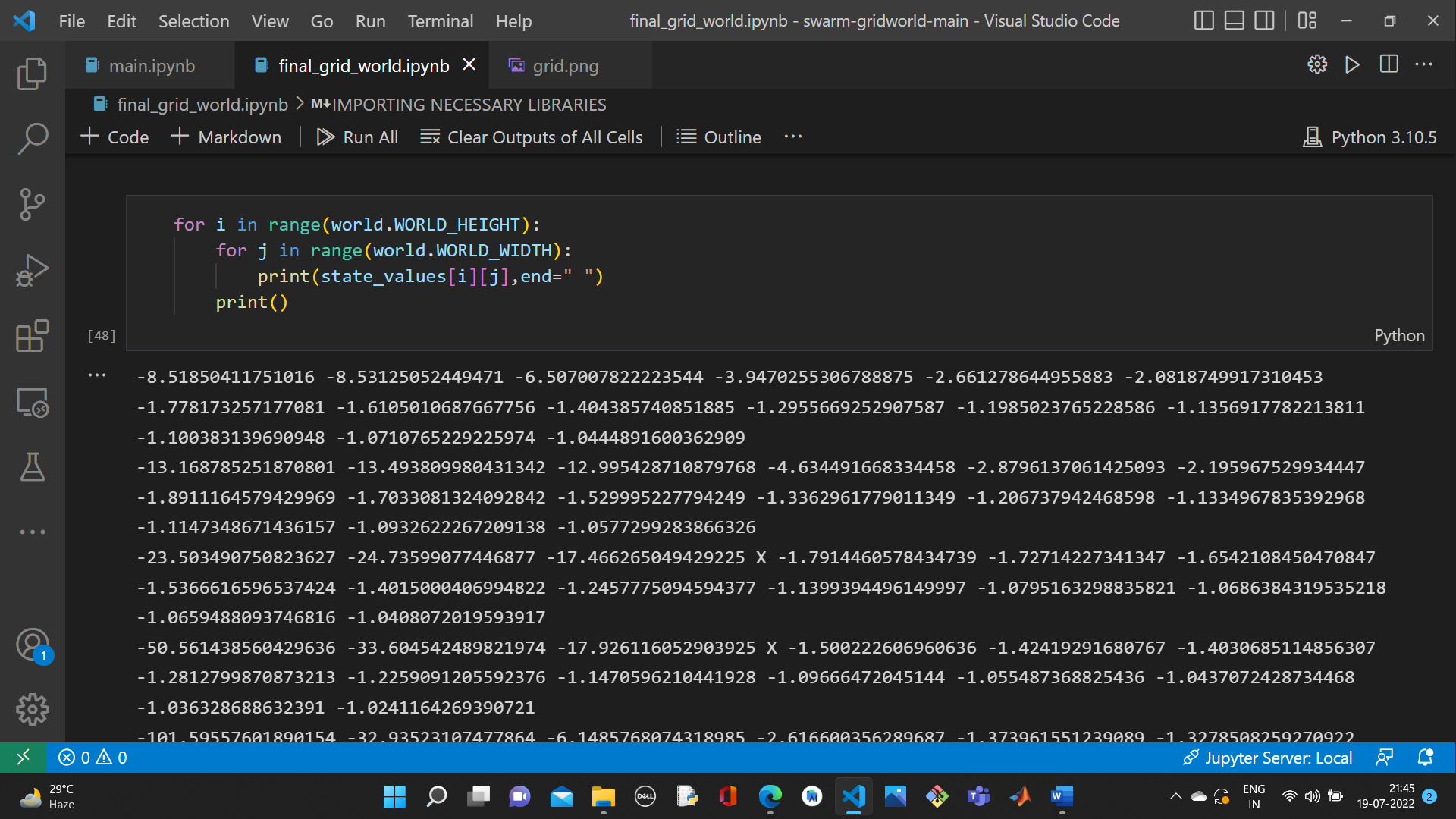
                print("to ",reward)

            states = [] # RESET

            break





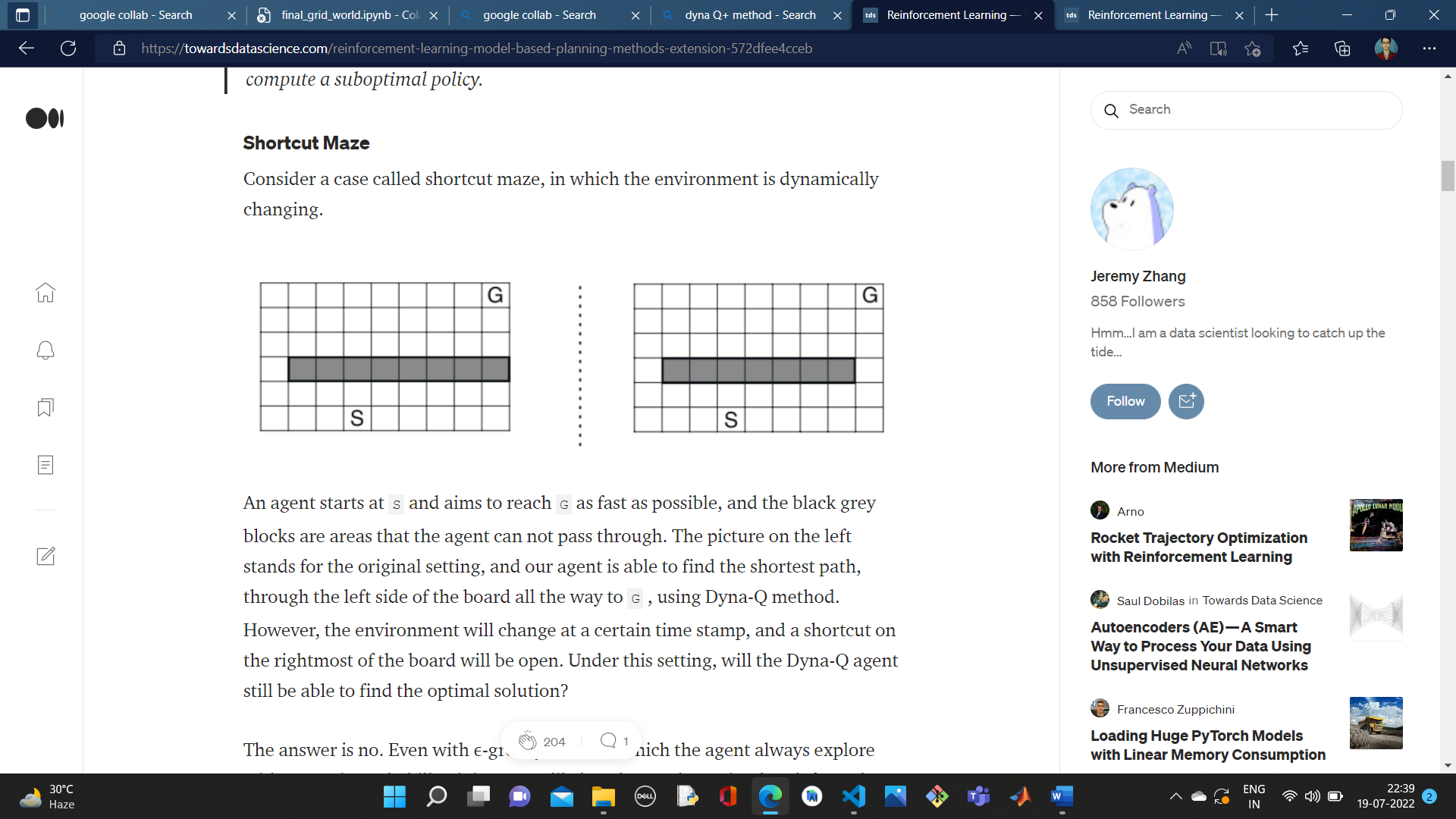


* Dyna-Q method: This method is same as the earlier mentioned Q learning by performing a greedy action except that it records a model of the environment(by assuming deterministic) and updates the Q value n times where n is a predefined parameter.

**We may think it as **repeating what the agent has experienced** several times in order to reinforce the learning process.**

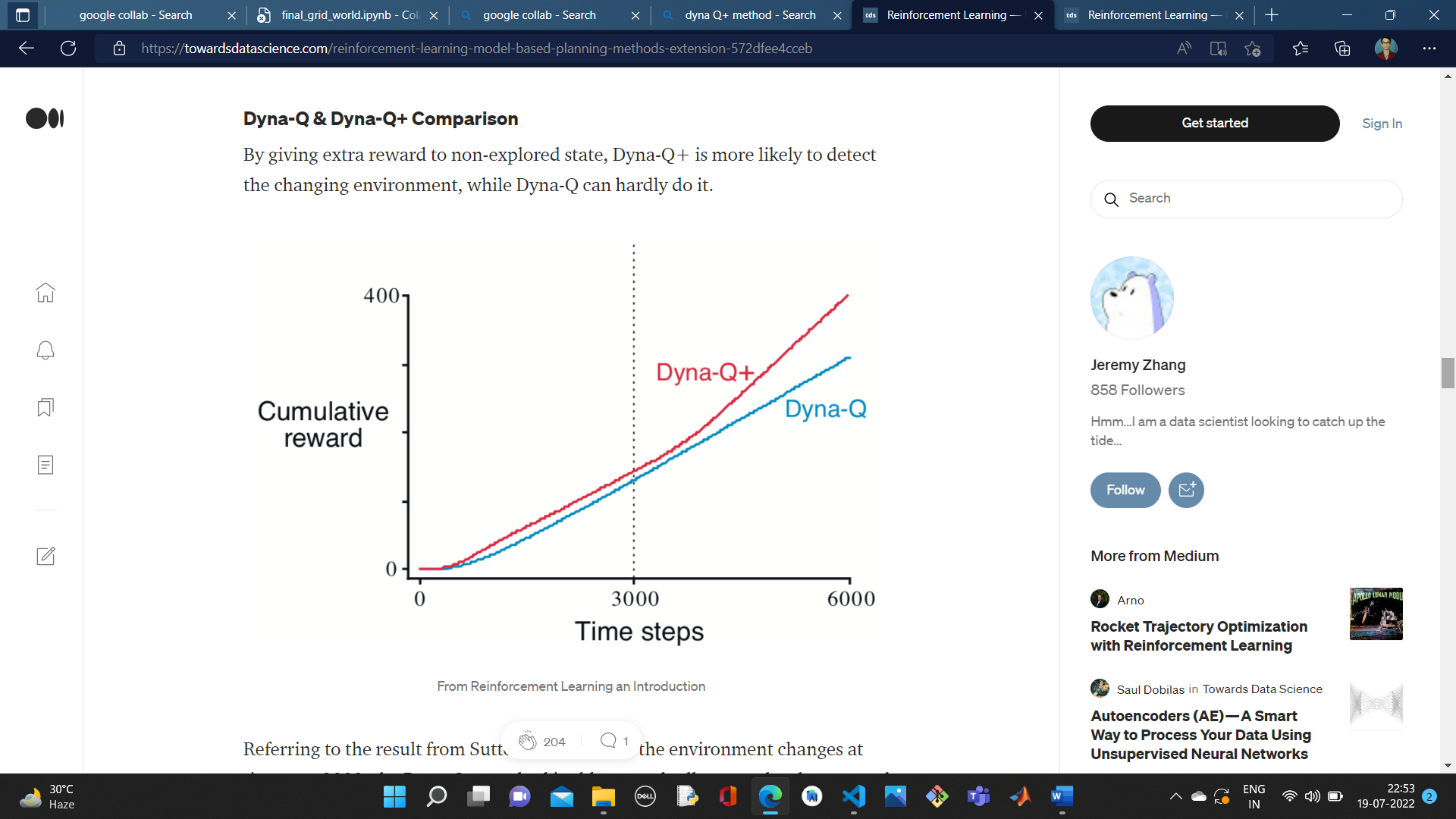
**In a nutshell, it is :**

* **Learning a model by experience**
* **Fully trust the model and apply it to reinforce the value function.**

**But what if the environment is dynamically changing with time?**

An agent starts at S and aims to reach G as fast as possible, and the black grey blocks are areas that the agent cannot pass through. The picture on the left stands for the original setting, and our agent is able to find the shortest path, through the left side of the board all the way to G, using Dyna-Q method. However, the environment will change at a certain time stamp, and a shortcut on the rightmost of the board will be open. Under this setting, the Dyna-Q agent will not be able to find an optimal path. Even with e-greedy method, it is very unlikely that the agent will explore the shortcut path.

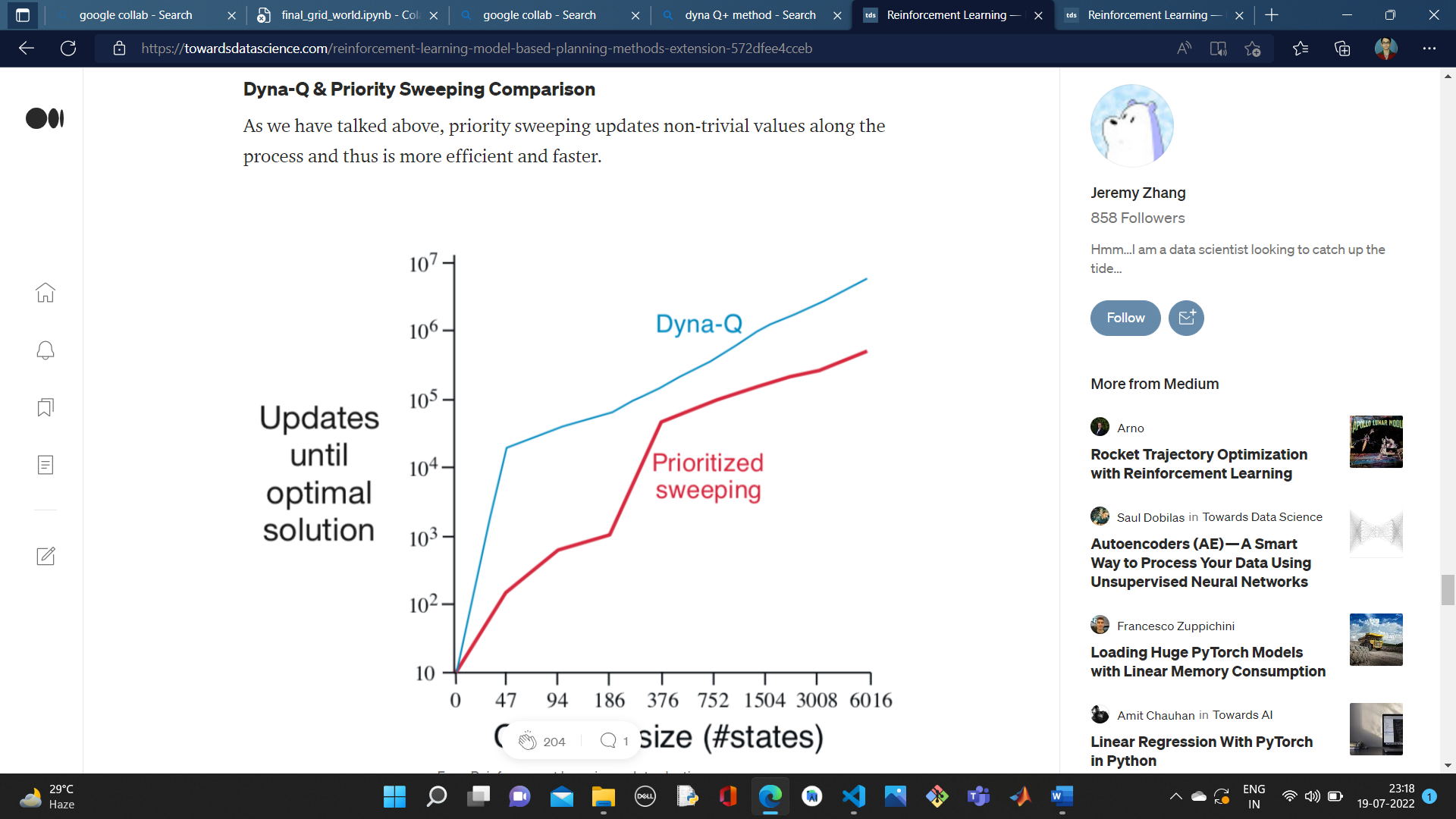
* Dyna-Q+ method: To solve the above problem, a simple essence is to keep the agent to be able to explore new states to fit the changing environment.

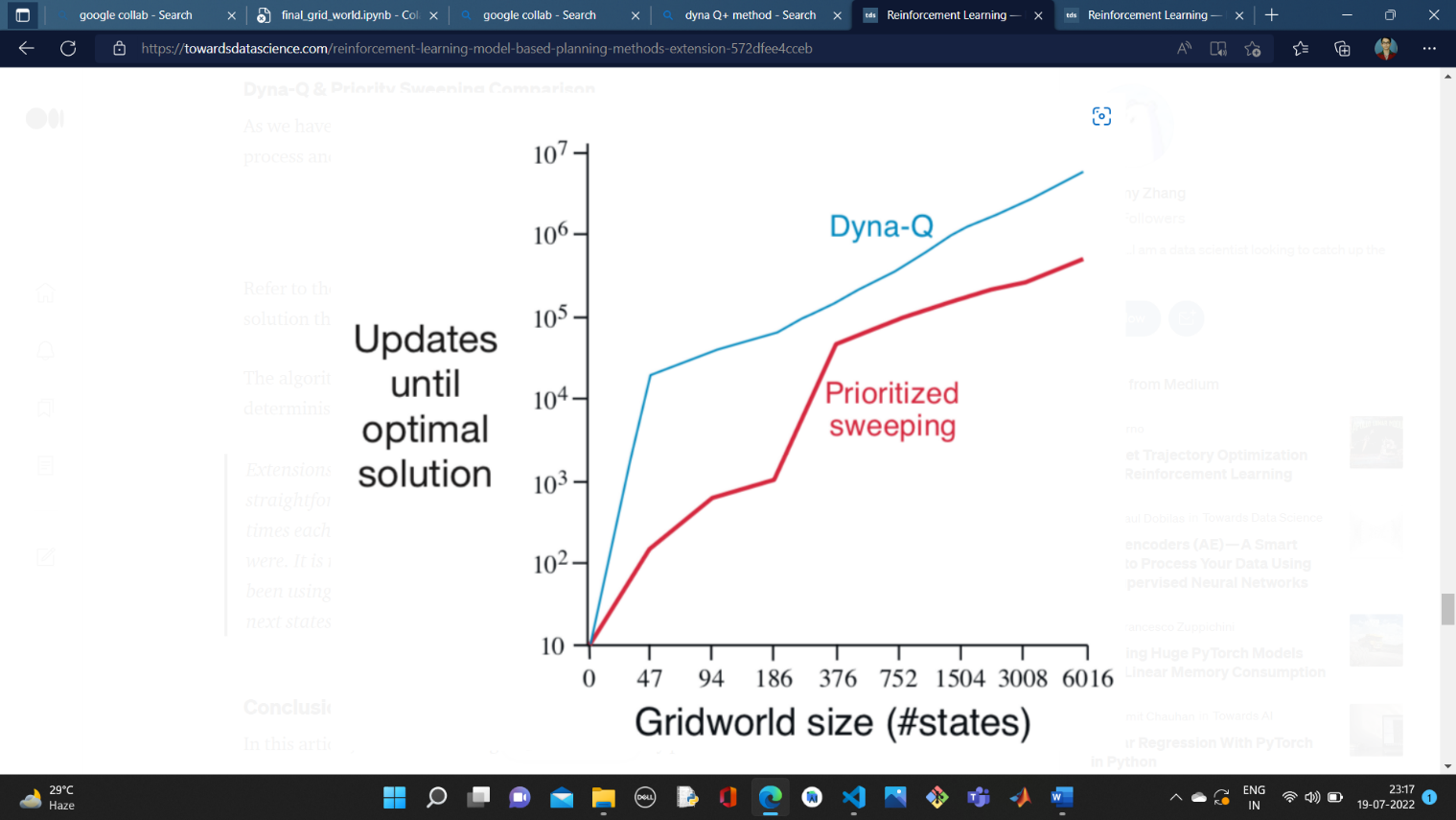
**To encourage behavior that tests long-untried actions, a special “bonus reward” is given on simulated experiences involving these actions.** In particular, if the modelled reward for a transition is r, and the transition has not been tried in τ time steps, then planning updates are done as if that transition produced a reward of r + κ\*sqrt(τ), for some small κ.

*Image source: Towards Data science.com*

* Priority Sweeping: It is worth noticeable that **in the planning stage, there are actually many invalid updates**, especially at the beginning phase when the Q functions of all states and actions are 0 and rewards along the way are also 0.

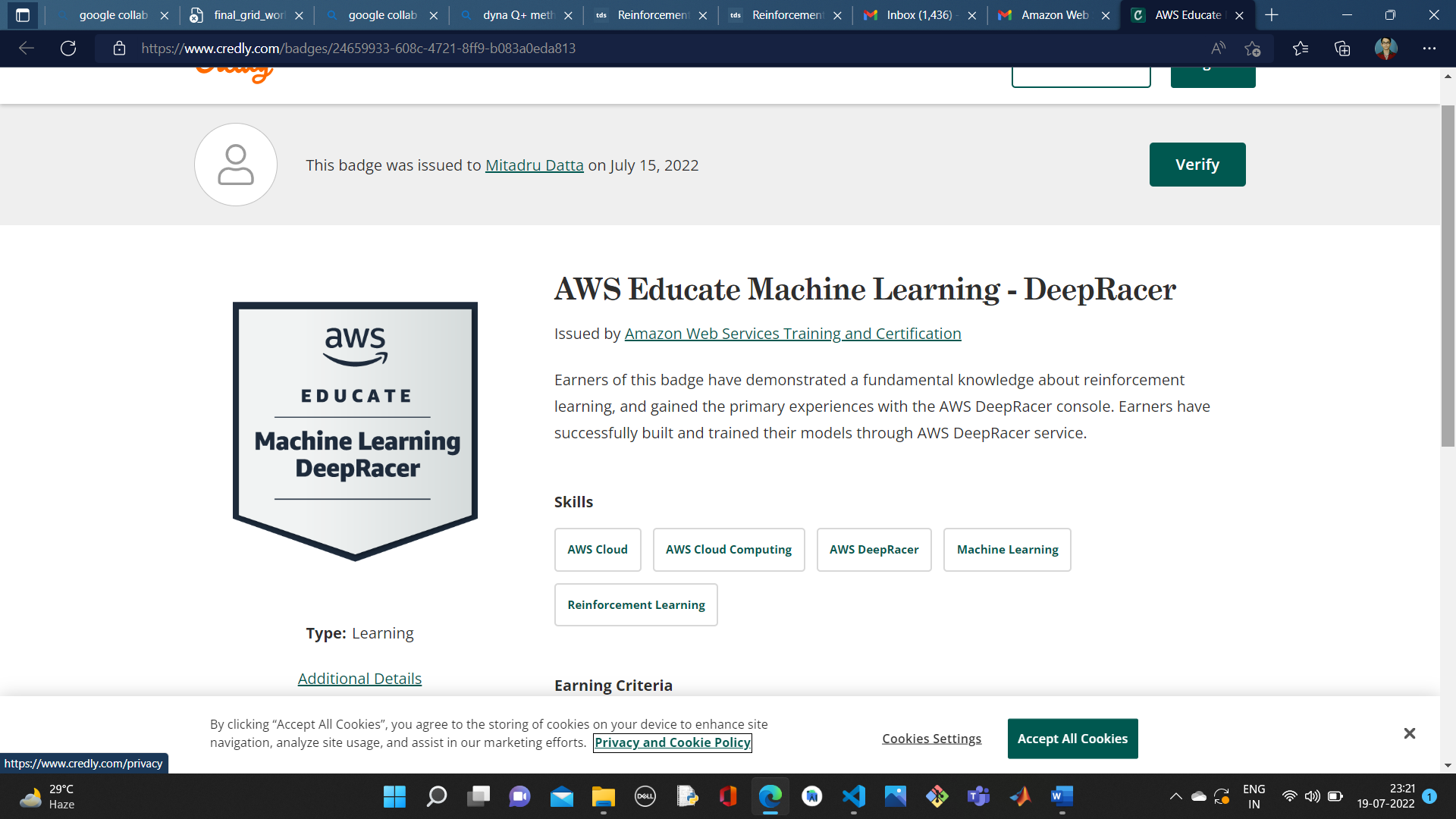
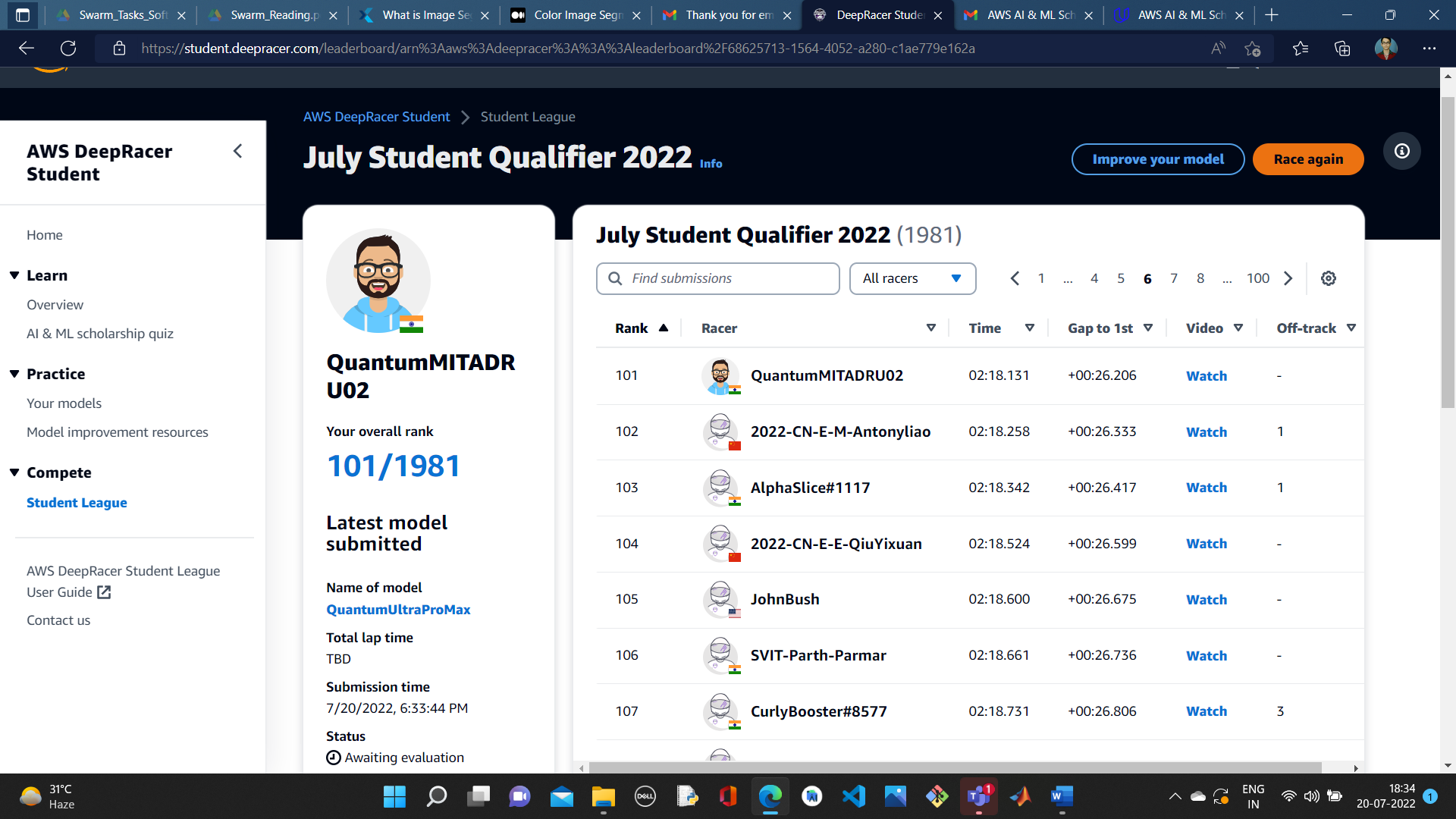
Priority sweeping focuses on updating non-zero values in the planning stage. The intuition is since many updates are 0s, We are able to only update values that are higher than a certain threshold, thus make the learning faster.

In this algorithm, an additional parameter theta is used, only those temporal difference greater than theta are added in the priority queue.



*Image Source: Towards Data Science.com*

My other experiences with Reinforcement Learning:

* AWS Educate DeepRacer Badge
* Global rank 101 in AWS Deep Racer Students League