HW #2

Fateme Noorzad

810198271

Question 1:

An important idea in reinforcement learning is to give much more credit to rewards with higher values. It means the actions followed by larger rewards to occur in more trials then the ones with less amount.

To make a judgment about what does a large amount of reward mean, a reference must be set. A wise choice for the reference rewards is the expected value (average value) of previous given rewards.

Comparing the instant rewards with reference rewards is a good way for finding out how large instant reward is.

For reinforcement comparison learning these two formulas hold :

$$p_{t+1}(a_t) = p_t(a_t) + \beta[r_t - \bar{r}_t]$$

$$\bar{r}_{t+1} = \bar{r}_t + \alpha[r_t - \bar{r}_t]$$

which beta is given 0.9 and alpha is 0.1. (defined in problem)

In each iteration action preference is updated based on the difference of its instant reward and the average reward of previous rewards until trial t.

But the point is this algorithm doesn't have a method for finding the action values. So other methods of action evaluation is used. One of them is soft-max and parameter τ (temperature) is set as 1 :

$$\pi_t(a) = p[a_t = a] = \frac{\exp(p_{t-1}(a))}{\sum_b \exp(p_{t-1}(b))}$$

This will help us find the probability of selecting action a on the t^{th} play. In other words in this method the action values for updating probability of choosing an action is updated using an always updated average reward.

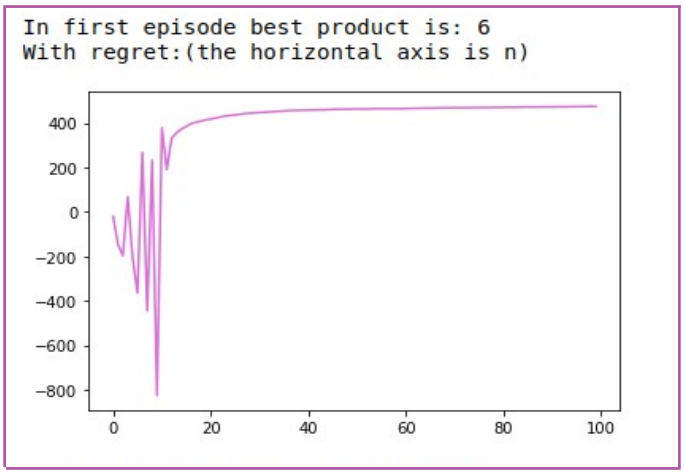
So the reinforcement comparison method explained and formulated above will help us update the probabilities and action preferences while Soft-max policy will help us find the action values.

The initial values of action preference can all be set to zero.

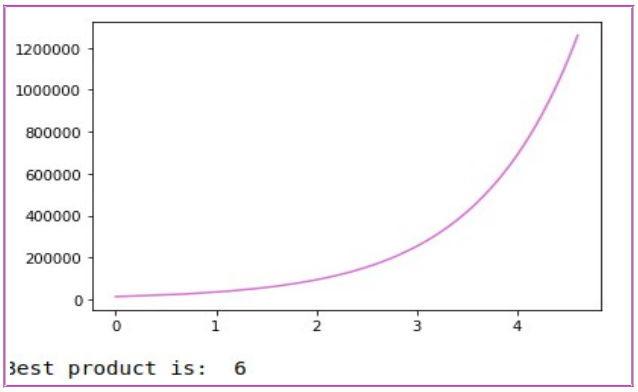
Code explanation :

For coding the Reinforcement Comparison a class called “RLComp” is created. As been said this algorithm doesn’t have a part for choosing an arm. In the last page Soft-Max policy was introduced as a way of this goal in each trial. But in code both ε-Greedy and Soft-Max are implemented.

Soft-Max Implementation : It was said that temperature is 1 for this algorithm. But since the value of each product can become a very large number resulting in overflow in exponential function, temperature is set as a larger number and is fixed. Although this helps us in solving the overflow problem, exact convergence can’t happen in only one episode. One episode containing 100 trials results in both product 2 and product 6 and other products as well. If we find regret in this way, it’ll be like the bottom plots:



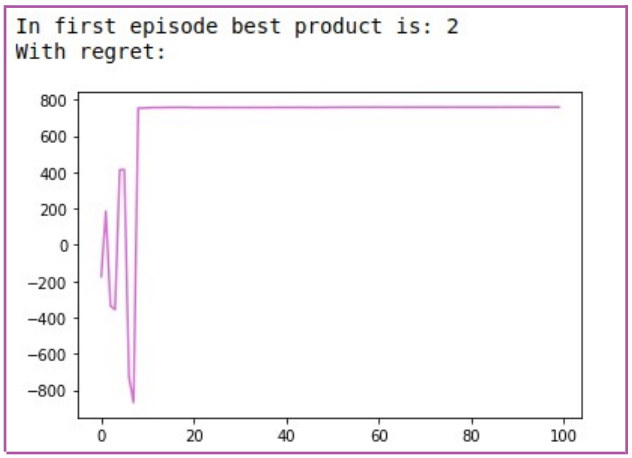
But if the learning process includes 100 episodes and the most selected product is chosen through all the episodes, a more converged result can be found.



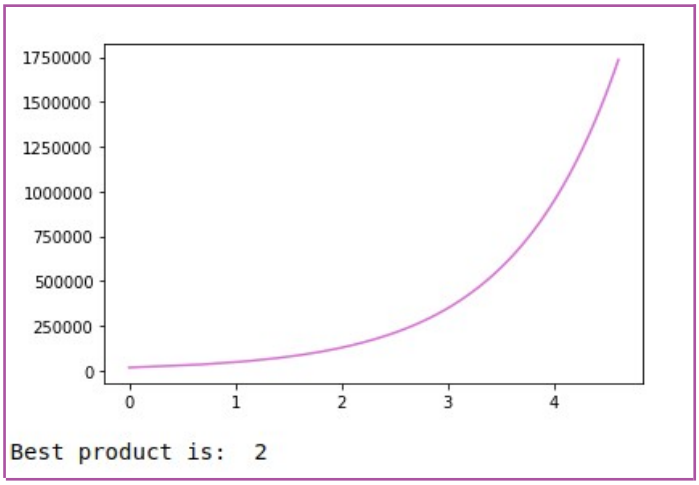
The above plot is the expected value of regret through all 100 episodes. As can be seen it shows a better convergence and one thing to note is that if you run the code in attached file the result is more converged. (most of the time product 2 and 6 which logically are better are chosen)

For running the attached code with soft-max policy type soft-max in the initialization of algorithm.

ε-Greedy Implementation : For running code with this algorithm just type ep-greedy in the initialization of algorithm. Same as last part both one episode and 100 episode are calculated. In the case of one episode regret is found :



And for 100 episodes the average regret is :



Again in here a more converged result can be seen. (Meaning if you run the code for some number of times you'll see product 2 or 6 most of the time as the best product, which in the case of one episode the diversity of answer is more.)

So to sum up best products are product 2 and 6.

Relationship between α and β: Since preference of a product is a function of β, we want β to be large enough in order to make the preference of a product with bigger reward more than it already was. It means if in a trial the instant reward of a product be bigger than the average reward, it must have a bigger preference. (as a result a bigger probability of being chosen) But for α the exact opposite story holds. Meaning we don't want our average to be very dependent to instant reward. So that our reference for how big rewards are be almost the same through all trials.

These statements are based on what this algorithm is about. In this algorithm we wanted to give a higher credit to an arm with a bigger reward so that it be chosen in many more times that the other ones with smaller rewards. And we said for finding how big or small a reward is we need a measurement which was the average of rewards. So for making these assumptions happen α must be less than β. Or in better words α be a smaller value and β a larger one.

About greedy policy: In this learning problem, actually the first arm is chosen completely random. But after that based on the choosing algorithm (Soft-Max or ε-greedy) arms are selected. But a closer look at the chosen arms show that, after 2-3 trials the algorithm chooses an arm till the end of the trial. This means if we have a prior knowledge about the rewards each arm provides (Not even a very accurate knowledge) a greedy policy can be helpful and can converge to the same results as above. But this

must be noted that the convergence is under one condition, which is the prior knowledge. (Wrong prior knowledge leads to wrong result.)

Question 2:

In this question 3 type of taxis are used. First taxis information are gathered in a list called taxiProp. In this list besides taxis name, the probability of not finding a driver and traveler's wasted time is included.

Traveler's wasted time is calculated based on finding or not finding a driver. If a driver is found, wasted time is zero. Otherwise wasted time is a number chosen from a normal distribution with mean and variance given based on each company.

For each detective wasted time is found based on how he\she reacts to finding or not finding a taxi.

Codes can be found in the file included in project folder and results are printed there too. Important point that needs to be mentioned here is that, since Poirot has a more logical way for choosing a taxi company, his wasted time is usually less then the other 2. Miss Marple has a logical choice about not changing taxi company when a driver is found, but has a bad choice when it comes to not finding a driver. She should've given a higher probability to changing the company like Poirot. Sherlock on the other hand has given a logical choice for changing taxi company when no driver is found, but has a wrong choice for changing a company when he had no trouble finding a driver. This makes Poirot's time less than the other two.

In one of the runs the results are :

```
Hercule Poirot's total wasted time is:  598.8726120917065
Miss Marple's total wasted time is:  634.5770951358805
Sherlock Holmes's total wasted time is:  642.940884460504
```

In order to make a better and more converged view of their total wasted time, these trials are done 10 times more and their average is found. If we do it, all the above explanation will be shown in a better way. The results are :

```
Hercule Poirot's total wasted time is:  3187.957844521454
Miss Marple's total wasted time is:  3376.1472489365296
Sherlock Holmes's total wasted time is:  3564.679878572034
```

Question 3:

For answering this question, 2 parts must be explained. The first part of my code is a class called UCB2, which as its name shows is a class with some attributes to implement a UCB2 algorithm. The next part is implementation of 3 ways Sara can go to work and using the first part's class for finding the best one.

First I want to explain what UCB2 algorithm is and then want to bring some reasons for why implementing this algorithm with a class is easier.

UCB2 is an attempt to improve UCB1. This improvement is actually about reducing regret with a trade-off of a more complicated algorithm. Same as UCB1 this algorithm uses a new definition for utility function. Utility function is made up of 2 parts. One of them is the expected value of rewards, which is shown by Q (called q value in code) and the other one is called bonus.

In UCB2 each iteration is broken into epochs. Epochs have varying size and are calculated using a function called τ. In each epoch arm *i is* selected to maximize :

$$Q_i + \sqrt{\frac{(1+\alpha)\ln(en/\tau(r_i))}{2\tau(r_i)}}$$

Which τ(r) is calculated as:

$$\tau(r) = \lceil (1+\alpha)^r \rceil$$

Q_i is the expected value of rewards, α is a number between 0 and 1 and *e* Euler's constant.

After selecting the maximizing arm, it's played exactly :

$$\tau(r_i+1) - \tau(r_i) = \lceil (1+\alpha)^{r_i+1} - (1+\alpha)^{r_i} \rceil$$

times before ending the epoch and starting a new iteration with selecting a new arm.

Should note that r_j is a counter indicating how many epochs arm *j* has been selected.

In the above equation α is a parameter which influences learning rate. More analysis about it will be discussed.

So a big difference between UCB1 and USB2 is this epochs. In UCB1 the selected arm is played only once which as was said is not like that here.

Based on the complication of saving parameters like number of total epochs and r_j for arm j, a class having them all as their attributes is the best choice. Although using

functions or even no function is ok too, but a class with attributes mentioned makes a cleaner and more understandable code.

For UCB2 class 6 attributes are defined. Alpha's importance will be discussed. Other one is pathsCount which is the number of times each path is chosen. Qvalues are the expected value of rewards (path cost and rewards) . PathsRetakeCount is a list containing all the times a path in chosen in an epoch. CurrPath as its name shows the current path and allRetakes is the sum of all the times a path is chosen including the times in epochs.

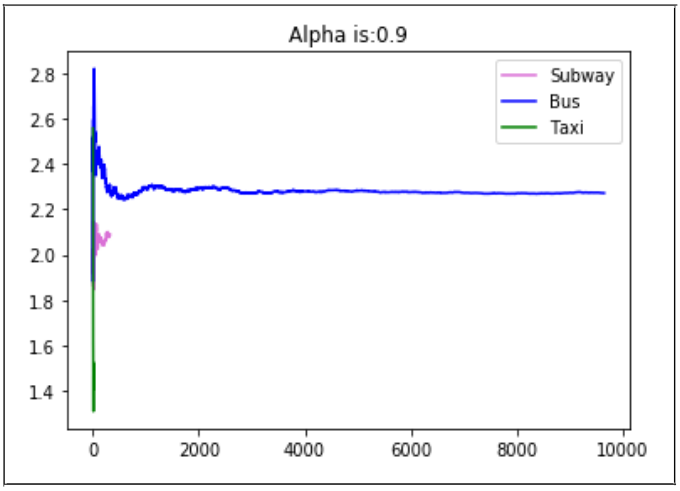
Calculating τ and bonus is based on the mentioned equations. Function setPath() is changing the attribute currPath into path which is the input of the function. Then increases PathsRetakeCount of that path. For updating allRetakes the above equation is used except one point should be mentioned. Max function is used here for when the substraction is less than 1. This may never happen because ceil() function is used. But for any cases that didn't work, max is there to fix the issue.

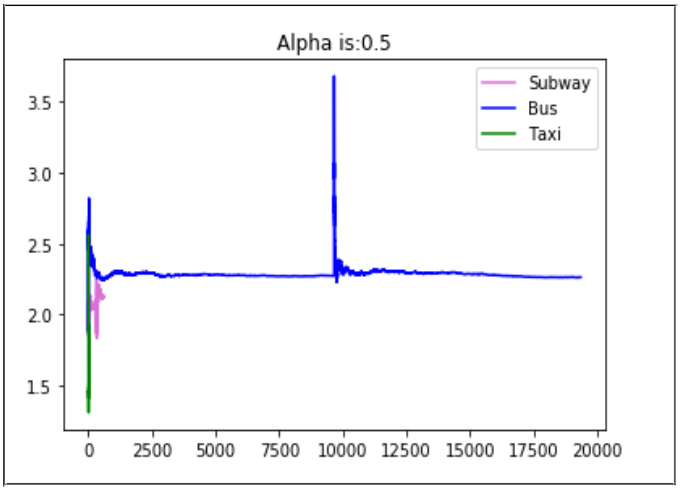
In this algorithm same as USB1 first all arms are played once. Here arms are paths which means here all path are taken once. This is taken care by setInitValue() function. If all the path are taken once then it should be made sure that a path is not chosen again after an epoch is finished meaning a path should be chosen based on maximizing the utility function. This point is checked by endEpoch(). Then utility function is found and the argument with the biggest utility function is selected as the path.

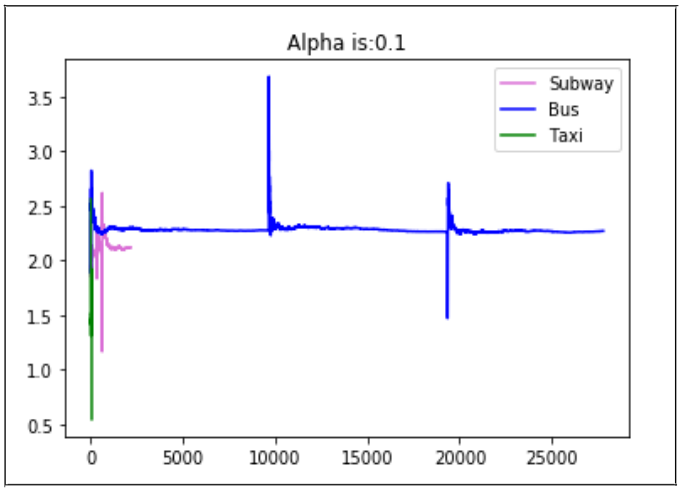
After all these calculations(!) q values must be updated which is by :

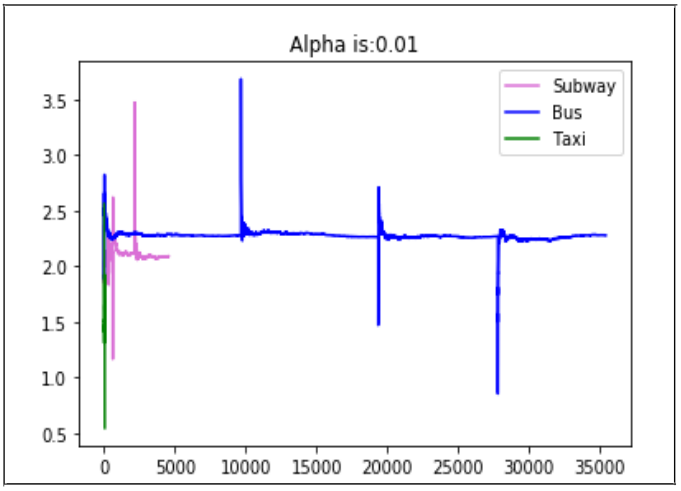
$$Q = Q + \frac{1}{n+1}(reward - Q)$$

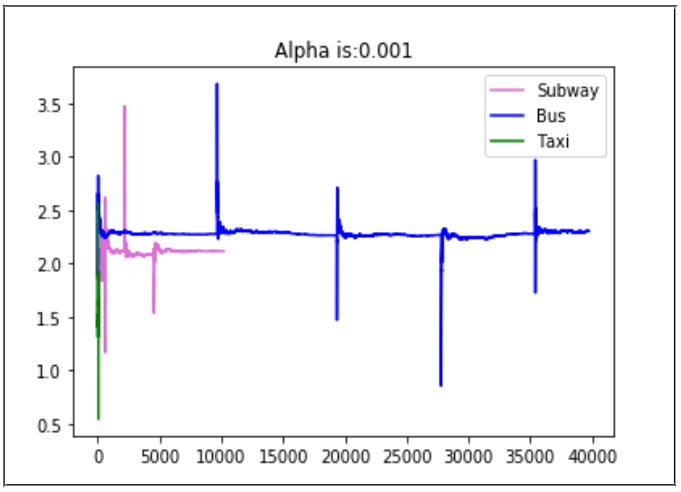
Next part of the code is about defining the cost and delay in each path. Then based on these values outcome of Sara's journeys(!) are calculated. Then the function findBestWay() tries to find the best path for Sara. In this part 6 different values for α is set. First one is 0.9 which is a value close to 1. The last one is 0.001 which tries to show a smaller number. As simulations show, as α decreases, for convergence more trials are needed. So a slow decaying α is actually the best one. Because in regret equation for this algorithm, choosing small or large value for it both has negative points and so it's a trade-off.


Alpha is:0.9
Subway
Bus
Taxi
2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
0
2000
4000
6000
8000
10000


Alpha is:0.5
Subway
Bus
Taxi
3.5
3.0
2.5
2.0
1.5
0
2500
5000
7500
10000
12500
15000
17500
20000


Alpha is:0.1
Subway
Bus
Taxi
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
5000
10000
15000
20000
25000





The above are the results of learning for different path and different values of α.

Question 4:

As said an agent's action's rewards is a linear combination of the actual rewards and agent's confidence about the action. So :

$$r(a_i) = \alpha\, r(a_i) + \beta\, p(a_i)$$

When $\alpha >> \beta$ and with $p(a_i)$ being a number between 0 and 1, there's not a huge difference between this policy of giving a reward and when $\beta=0$. But when $|\alpha|$ and $|\beta|$ are close the difference between these 2 ways will be more.

When $\beta > 0$ the agent gets a bigger reward for the action it thought has a bigger probability of happening then when there's no confidence parameter in rewards equation. So as the learning goes on, it becomes more and more confident about its choice and as a result will reach to the best action sooner. Since here there are only 2 actions, as the probability of one action increases, the other one decreases resulting in a smaller reward and making the learning process faster. Actually the agent gets more and more confident about the action so will start the greedy policy sooner resulting in a faster convergence to optimum action.

When $\beta < 0$ the agent gets a smaller reward than what it expected for both arms. But since for the action with bigger p a bigger amount is subtracted, it may cause the reward for a better action to be less than the worse action. This leads to a wrong conclusion as the optimum action. But if it doesn't happen, meaning if the rewards are big enough, even a $\beta < 0$ can lead to a faster conclusion.

When $\beta >> \alpha$ the reward of each action is mostly a function of agent's beliefs. So the action with a bigger value for p will get a bigger reward.