CS747: Programming Assignment 1

Aayush Rajesh
200070001

1 Task 1

1.1 UCB

The code for the give_pull and get_reward member functions for this algorithm are listed below.

```
def give_pull(self):
    # START EDITING HERE
    return np.argmax(self.values + self.bonus)
    # END EDITING HERE

def get_reward(self, arm_index, reward):
    # START EDITING HERE
    self.counts[arm_index] += 1
    self.curr_time += 1
    # Update probability estimate for pulled arm
    n = self.counts[arm_index]
    value = self.values[arm_index]
    new_value = ((n - 1) / n) * value + (1 / n) * reward
    self.values[arm_index] = new_value
    # Calculate Exploration Bonus for all arms
    for i in range(self.num_arms):
        self.bonus[i] = Explore_Bonus(self.curr_time, self.counts[i])
    # END EDITING HERE
```

At any point of time, we choose that arm which has the highest *upper confidence bound* (UCB). This UCB is available in the form of an array given by `self.values + self.bonus`. The values stored in the corresponding indices of this array represent the UCB values of each arm. The array `self.values` contains the estimated probability of reward of each arm (initialized to 0), and is updated for the pulled arm in the get_reward function. The exploration bonus, stored in array `self.bonus`, is updated for all arms at each time step, due to it's time dependence. The updated exploration bonus values are calculated by the Explore_Bonus function, which has been defined below for convenience.

```
def Explore_Bonus(t, n_sample):
    return math.sqrt(2*math.log(t)/n_sample)
```

The dependence of regret on the horizon is plotted in Figure 1. The plot exhibits a logarithmic nature, which is as expected, since the UCB algorithm is known to achieve regret of $\mathcal{O}(\log T)$, where T is the horizon. However, we see that the algorithm still incurs significant regret, and as evident later, is not as optimal as some other algorithms.

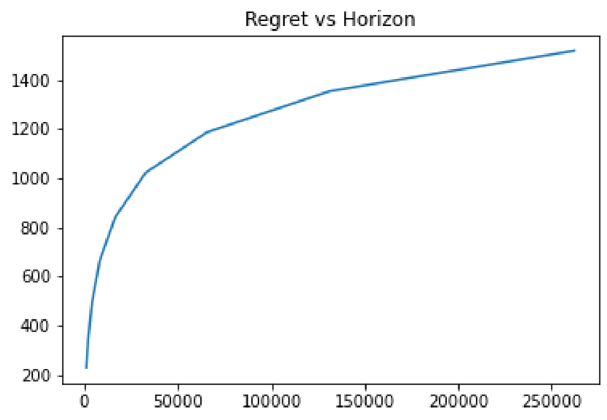

Figure 1: Regret vs Horizon for UCB algorithm

1.2 KL-UCB

The code for the `give_pull` and `get_reward` member functions for this algorithm are listed below.

```
def give_pull(self):
    # START EDITING HERE
    return np.argmax(self.confidence)
    # END EDITING HERE

def get_reward(self, arm_index, reward):
    # START EDITING HERE
    self.counts[arm_index] += 1
    self.curr_time += 1
    t = self.curr_time
    # Update probability estimate for pulled arm
    n = self.counts[arm_index]
    value = self.values[arm_index]
    new_value = ((n - 1) / n) * value + (1 / n) * reward
    self.values[arm_index] = new_value
    # Calculate q value satisfying inequality
    for i in range(self.num_arms):
        req_bound = (math.log(t) + 3*math.log(math.log(t)))/self.counts[i]
        self.confidence[i] = Find_Sol(self.values[i], req_bound)
```

At any point of time, we choose that arm which has the highest value of confidence, which is stored in the array `self.confidence`. The confidence is given by,

$$\text{Confidence} = \max\{q \in \left[\hat{p_a}^t, 1\right] : u_a^t \cdot KL(\hat{p_a}^t, q) \leq \ln(t) + c\ln(\ln(t))\} \tag{1}$$

where u_a^t and $\hat{p_a}^t$ are the number of times an arm a has been pulled, and it's estimated probability, respectively. The value of c taken in the code is $c = 3$. In order to compute the value of q, we have two helper functions defined as follows:

```python
def KL(p, q):
    return p*math.log(1e-9 + p/q) + (1-p)*math.log(1e-9 + (1-p)/(1-q))

def Find_Sol(low, val):
    # Running binary search for 10 iterations to find required point
    p = low
    iterate = 10
    high = 0.99
    for i in range(iterate):
        mid = (low+high)/2
        curr_val = float(KL(p, mid))
        if(abs(curr_val - val) < 0.01):
            return mid
        elif(curr_val > val):
            high = mid
            continue
        else:
            low = mid
            continue
    return mid
```

The function `KL` calculates the KL-divergence between p and q, and the function `Find_Sol` is used to calculate the value of q that satisfies equation 1. We use the bisection method iterated 10 times to compute q within a precision of two decimal places.

The performance of the KL-UCB algorithm is plotted in Figure 2. The logarithmic dependence of regret on horizon is clearly visible from the plot. On comparing the rough values on the plot with those obtained in Figure 1, we can see that the KL-UCB algorithm is far more efficient than the UCB algorithm, which is expected, since the former satisfies Lai and Robbins lower bound on regret, in theory. The exact values obtained on the plot is sensitive to our value of c, however, as long as $c \geq 3$, there is not much variation in performance.

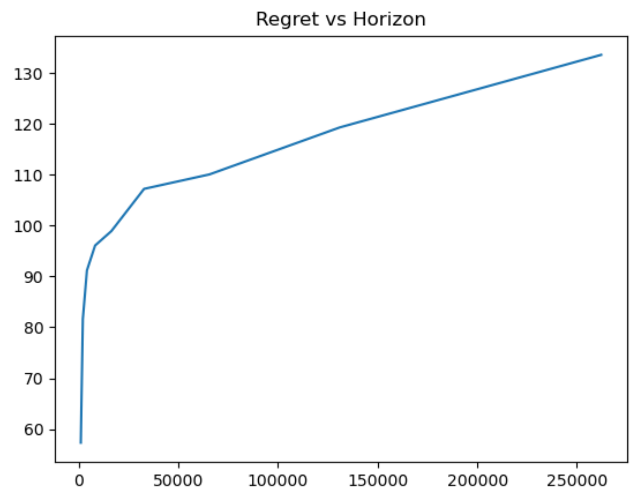

Figure 2: Regret vs Horizon for UCB algorithm

1.3 Thompson Sampling

The code for the `give_pull` and `get_reward` member functions for this algorithm are listed below.

```
def give_pull(self):
    # START EDITING HERE
    # Draw samples for each arm
    self.values = np.random.beta(self.success + 1, self.failure + 1)
    return np.argmax(self.values)
    # END EDITING HERE

def get_reward(self, arm_index, reward):
    # START EDITING HERE
    if(reward == 1):
        self.success[arm_index] += 1
    else:
        self.failure[arm_index] += 1
    # END EDITING HERE
```

At any point of time, we choose that arm which has the highest "belief", which is stored in the array `self.values`. The belief for each arm is updated at each time instant and is a randomly sampled value from a Beta distribution, with the parameters of the distribution being $\alpha = \text{success} + 1$ and $\beta = \text{failure} + 1$.

The plot depicting the variation in regret for different horizons is shown in Figure 3. We can see that for the horizon values for which we have analyzed the different algorithms, Thompson sampling performs the best, even better than KL-UCB. This lines up with the fact that Thompson sampling is an extremely good algorithm in practice. The graph does show a brief downward trend, which is not expected, but can be attributed to the nature of the random distribution instance.

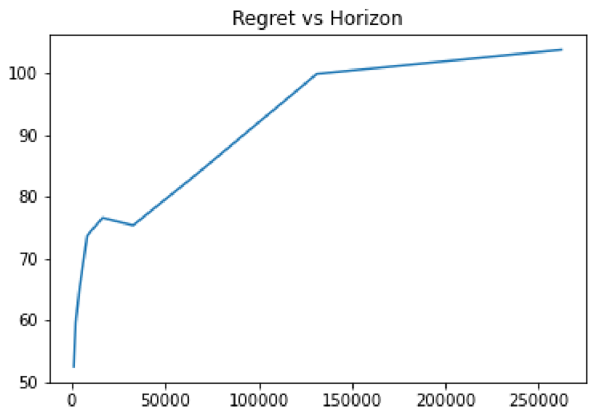

Figure 3: Regret vs Horizon for Thompson Sampling algorithm

2 Task 2

The code for the give_pull and get_reward member functions for this algorithm are listed below.

```
def give_pull(self):
    # START EDITING HERE
    best_arms = np.arange(self.num_arms)
    best_pulls = np.zeros(self.num_arms)
    # Exercising Thompson Belief batch_size number of times to determine pulls
    for i in range(self.batch_size):
        self.values = np.random.beta(self.success + 1, self.failure + 1)
        best_arm = np.argmax(self.values)
        best_pulls[best_arm] += 1

    return best_arms.astype(int), best_pulls.astype(int)
    # END EDITING HERE

def get_reward(self, arm_rewards):
    # START EDITING HERE
    for i in arm_rewards.keys():
        self.success[i] += np.sum(arm_rewards[i])
        self.failure[i] += len(arm_rewards[i]) - np.sum(arm_rewards[i])
    self.curr_time += self.batch_size
    # END EDITING HERE
```

The algorithm makes use of Thompson Sampling to determine the arms to pull, as well as the number of pulls, for each batch. For a batch, we compute the sampled value of the Beta distribution of each arm, `batch_size` number of times. The arm with the highest belief in each of these `batch_size` sampling instances, is allotted one pull. Therefore, in a batch, we pull each arm the number of times it had returned the highest belief value.

The optimality of this algorithm is governed by the fact that Thompson Sampling is a randomized algorithm. Since all our decisions made in a single batch are based on a fixed history, which is not updated until the end of the batch, deterministic algorithms would fail to work. This is because, a deterministic algorithm would always make the same decision based on a fixed history, leading to a single arm being pulled in the entire batch. Thompson Sampling on the other hand, makes a decision in a unique fashion each time, while ensuring that the decision is optimal based on the likelihood of the arm being optimal.

The variation of regret with the batch size is plotted in Figure 4. From the plot, a linear dependence is visible, which can be attributed to the fact that as the batch size increases, we receive feedback from the environment less often, forcing us to make many decisions based on a fixed past history. Therefore, we are not necessarily making optimal decisions during each batch, but instead making the best decision on the basis of the limited history available to us.

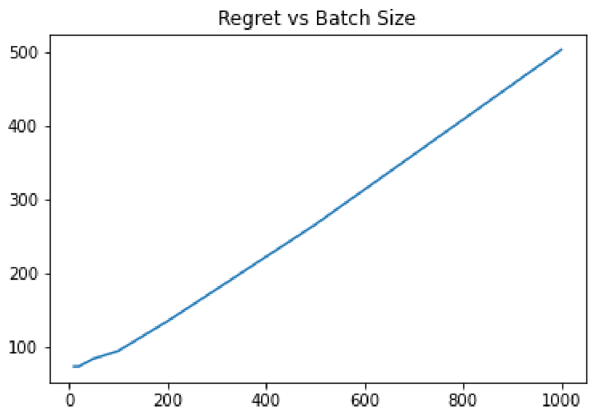

Figure 4: Regret vs Batch Size for implemented algorithm

3 Task 3

The code for the `give_pull` and `get_reward` member functions for this algorithm are listed below.

```
def give_pull(self):
    # START EDITING HERE
    eps = min(1, self.const/(self.curr_time + 1))
    # Implementing Modified Eps-Greedy
```

```
        if(np.random.random() < eps):
            return np.random.randint(self.num_arms)
        else:
            return np.argmax(self.values)
        # END EDITING HERE

    def get_reward(self, arm_index, reward):
        # START EDITING HERE
        self.counts[arm_index] += 1
        n = self.counts[arm_index]
        value = self.values[arm_index]
        new_value = ((n - 1) / n) * value + (1 / n) * reward
        self.values[arm_index] = new_value
        self.curr_time += 1
        # END EDITING HERE
```

The algorithm makes use of the GLIE-fied ϵ-greedy algorithm. The main motivation to use ϵ-greedy is the fact that the horizon is finite and comparable to the number of arms, therefore the conditions of GLIE (Greedy in the Limit with Infinite Exploration) are not satisfied, and any algorithm cannot give cannot give sub-linear regret on all bandit instances. The modified ϵ-greedy algorithm, however, performs decently in a finite-horizon setting. To determine the constant factor scaling the value of ϵ, I referred to the paper by Auer et al. (2002), referenced on the course webpage. The value of ϵ is varied as,

$$\epsilon_t = \min\left(1, \frac{25}{t}\right)$$

Therefore, at any time instant, with probability ϵ_t, we choose a randomly selected arm, otherwise, we choose the arm with the highest estimated probability of reward, which is stored in the array `self.values`.

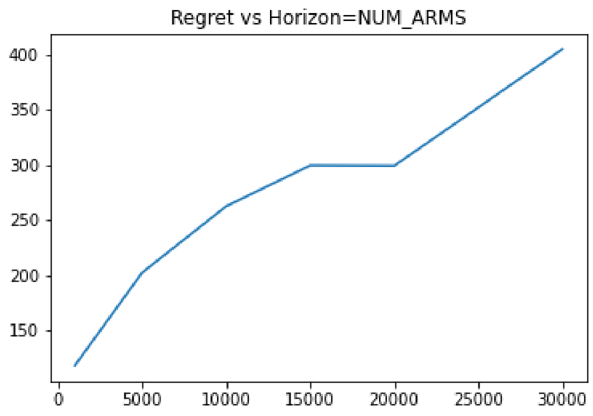

Figure 5: Regret vs Horizon for implemented algorithm

The plot showcasing the variation in regret with the horizon (which is equal to the number of arms) is showcased in Figure 5. From the plot it is evident that upto a large value of horizon (about 20000), the algorithm performs reasonably well, even going as far as providing what seems to be sub-linear regret. This is attributed to the fact that the modified ϵ-greedy algorithm balances finite exploration and finite exploitation to a good extent. However, at very large horizon values, the regret scales up linearly, due to the fact that over such large time scales, other algorithms like KL-UCB and Thompson Sampling, start proving to be more effective in their decisions.