Programming Assignment 1: CS 747

Richeek Das : 190260036

3rd September 2021

Task: 1

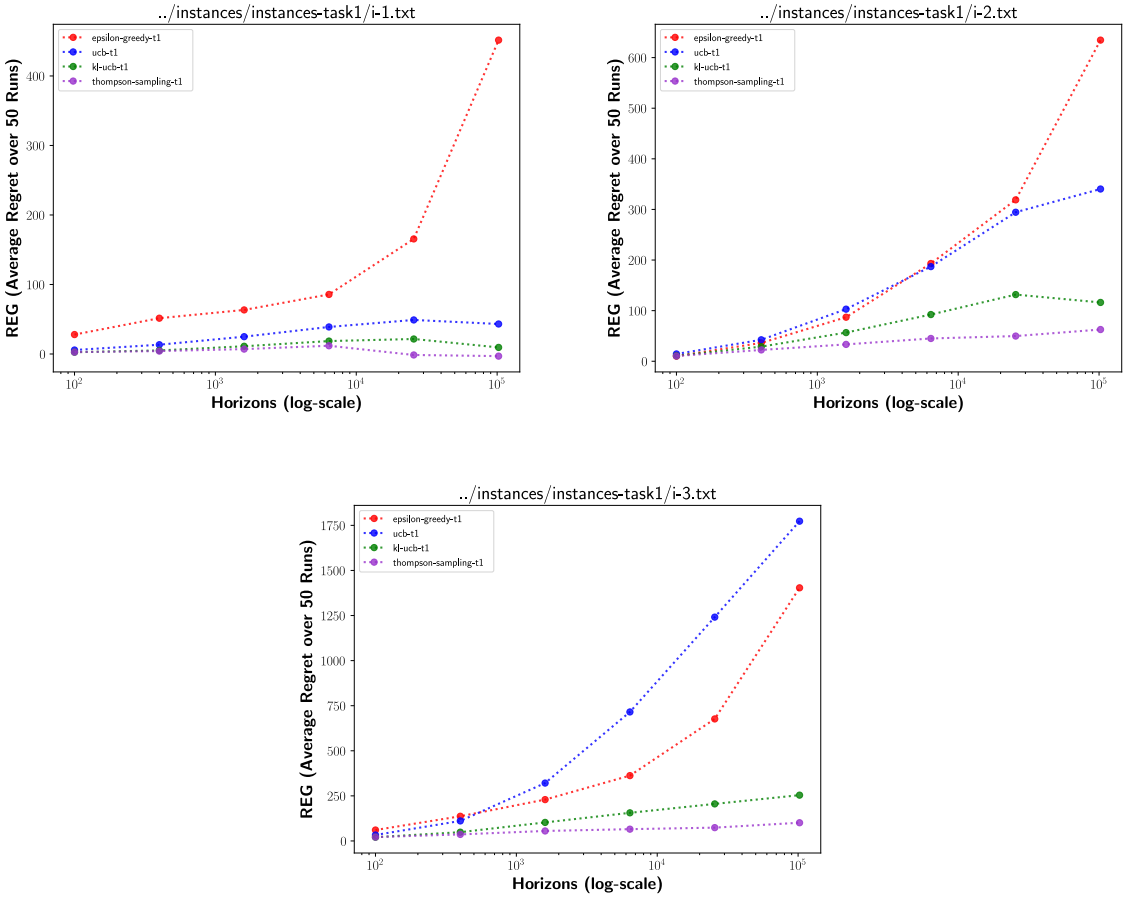

Figure 1: **Average Regret vs Horizons** for each of the 4 bandit planning algorithms

- **epsilon-greedy-t1**: In this function we implement the $\varepsilon - 3$ algorithm described in class. There are no wild assumptions for this one. We initialise the empirical means to 0. For every run in the horizon we pick an arm at random with probability ε and we pick the arm with highest empirical mean with probability $1 - \varepsilon$.
- **ucb-t1**: In this function we take the default value **c** to be 2. Here we initialise the empirical means to 0 and initial number of samples for each arm to 1 (we do this to make sure valid values of the confidence bound). Rest follows same as described in class.
- **kl-ucb-t1**: To implement this function we create two subroutines, namely: **kl_divergence**

and **find_q**. **kl_divergence** finds the KL Divergence of the two input variables. Here we handle any corner cases ($x \in 0, 1$ or $y \in 0, 1$). **find_q** function finds the max value of **q** such that the KL-bound is honoured. Owing to the monotonic nature of KL-Divergence, we perform binary search to find the value of **q**.

- **thompson-sampling-t1**: In this implementation we initialise successes to 0 and failures to 1 (to get a valid beta distribution). For sampling from the beta distribution we use the **numpy.random.beta** function from the NumPy library. For every run in the horizon we sample from the beta distribution for each arm and pull the arm which gives the highest value.

Results: In the results of **instance-1** we see, what we would naturally expect. **epsilon-greedy** algorithm shows a much higher regret than the rest. This is partially because **epsilon-greedy** continues to sample randomly with a pretty high probability. So it ends up sampling the arm with **0.4** probability a lot. To this end, other algorithms learn the difference between 0.4 and 0.8 arms pretty quickly and are able to achieve a sub-linear regret.

Again in **instance-2**, we see the regret of **epsilon-greedy** moving up exponentially (note that x-axis is in log-scale). This means the regret we observe in **epsilon-greedy** is sort of linear. In this case, **ucb-t1** seems to grow linearly wrt to the log-scale x-axis. This is consistent with it achieving sub-linear regret but with high constant factor.

In **instance-3**, the regret of **ucb-t1** overtakes that of **epsilon-greedy**. But we can still notice that **ucb-t1** is linear wrt to the log-scale and **epsilon-greedy** is exponential wrt the log-scale. This means the regret of **ucb-t1** increases linearly but with a high constant factor. If we had run our algorithms for a larger horizon, we should expect a higher regret from **epsilon-greedy**. As expected **thompson-sampling** and **kl-ucb** achieve the best regrets in all of the instances.

Task: 2

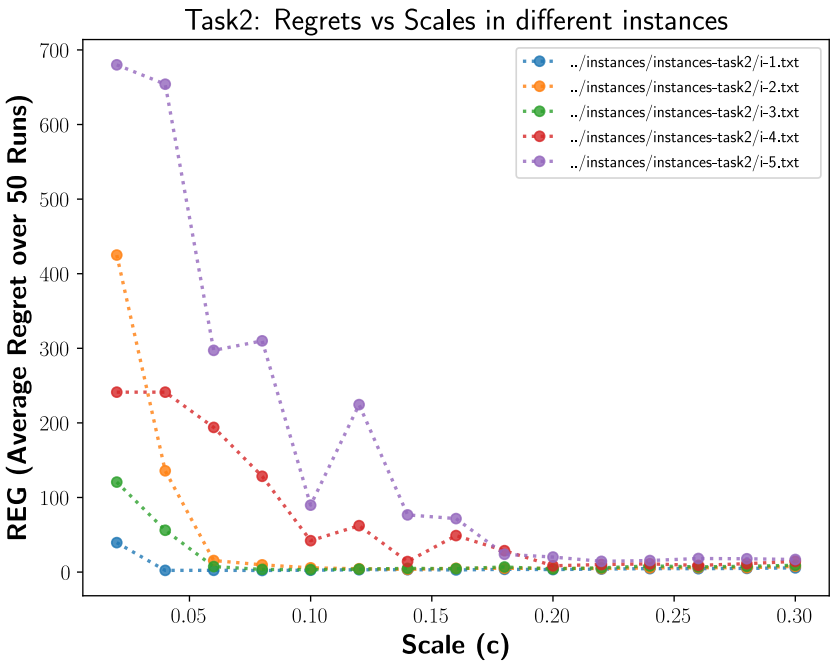

Figure 2: Regrets vs Scales for different instances

(1) the value of c that gave the lowest regret for each of the five instances:

Instances	$i-1$	$i-2$	$i-3$	$i-4$	$i-5$
Best **c**	0.08	0.14	0.10	0.20	0.22

Table 1: Table of instances and the scales with achieves the lowest averaged regret

(2) what trend is observed across the instances, along with an explanation:

Intuitively speaking the instances get harder to solve in this order: $i-5 > i-4 > i-3 > i-2 > i-1$. For example in $i-1$ we see, the arms having a true probability of 0.2 and 0.7. Here the huge difference in true probability makes it easier to learn the best arm with small exploration. Hence a small **c** of 0.08 was enough to to get the best regret. On the other hand in $i-5$ the arms had a true probability of 0.6 and 0.7. The empirical means we observe by sampling these two arms will be very close. This makes it much harder to decide which one has the highest true mean without much exploration. Hence in this case we observe the best regret with a much higher **c** of 0.22.

Task: 3

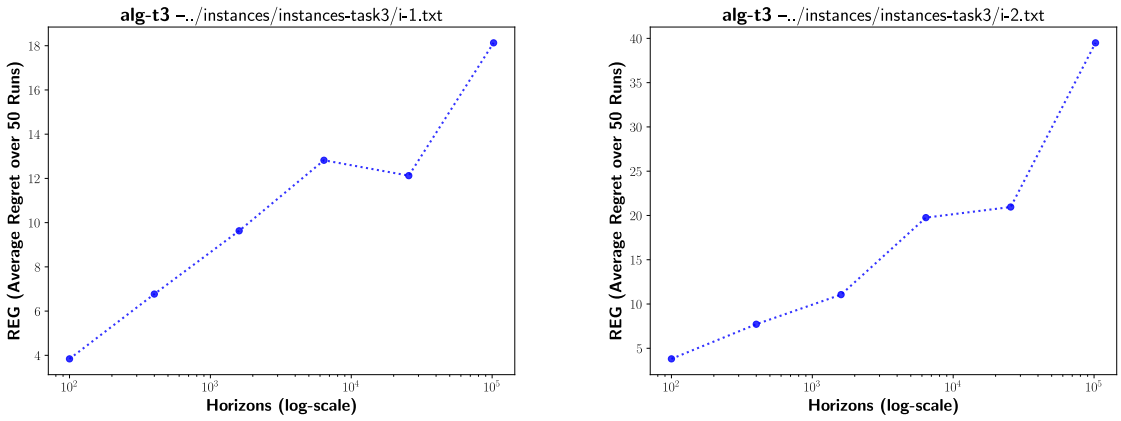

Figure 3: Average Regret vs Horizons for **alg-t3** which was specifically designed to handle finite reward support

Calculating max-expected-regret: To find the maximum expected regret, we find the arm which gives the max expected reward in each run, i.e $\arg\max(probabilities \times rewardSupport^T)$. Once we have this, we find the max expected reward by multiplying this with the Horizon.

Algorithm proposed: To solve the problem if finite reward support, we extend the Thompson Sampling algorithm. In Thompson Sampling algorithm we pick the arm which gives the highest value when sampled from its corresponding beta distribution. We extend the same by using the Dirichlet distribution (i.e. the Multivariate Beta Distribution). In the [0, 1] reward case, we keep two variables: **successes for 1** and **failures for 0**. In this case, we associate a variable for each value in the reward support. We keep a count of each reward we receive. This makes it a k-length list **for each arm**. Now if we have k possible rewards in the support, we sample from a k-category Dirichlet distribution, with these reward counts as the distribution concentration parameters. Once we are done sampling from each arm for a particular run, we decide the arm to pull based on a similar metric as in the binary case.

```
armToPull = np.argmax(np.array(armweights) @ np.array(rewardList))
```

where, `armweights` are the values sampled from the Dirichlet distribution. Here we weigh each sample we receive from the Dirichlet distribution, on the basis of the reward. We pick the arm which gives the highest expected reward!

Intuitively this works, because just like in the case of Beta distribution we try to learn the expected number of successes and failures, here we try to learn the expected number counts for each value in the reward support.

Task: 4

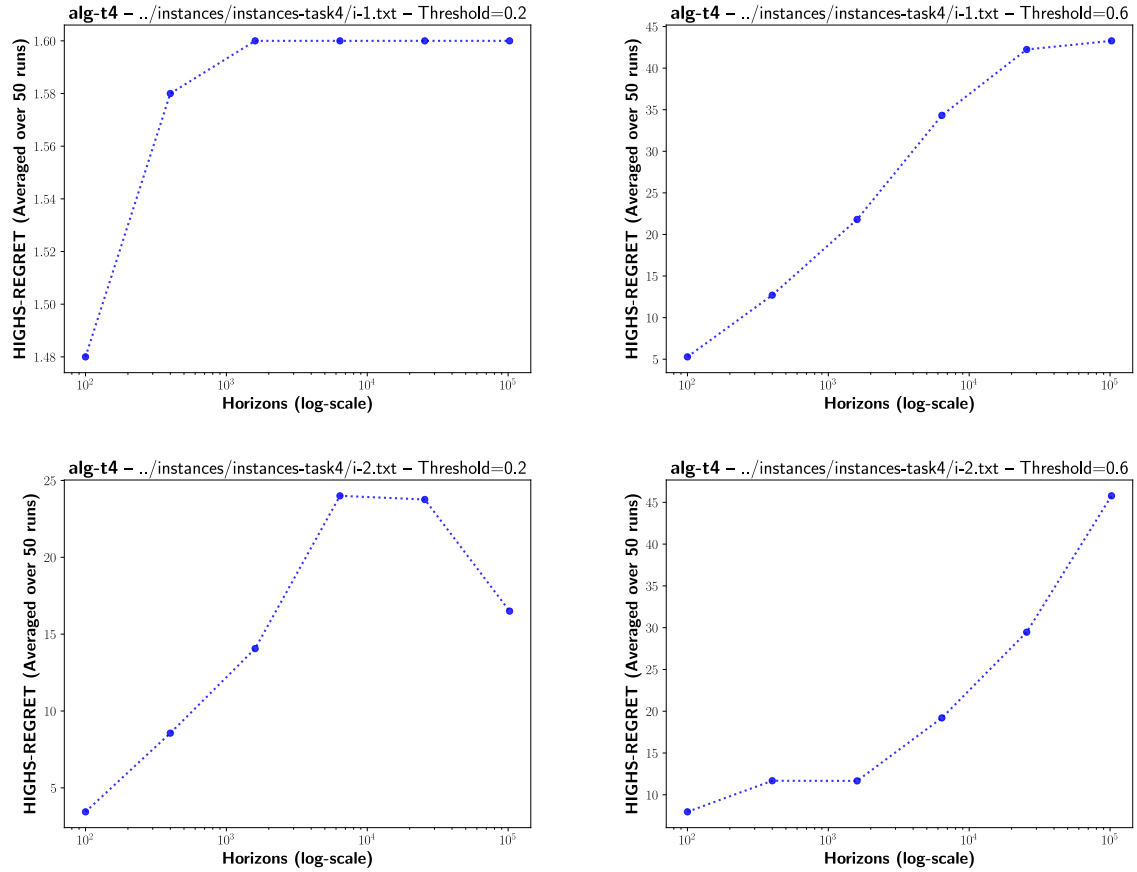

Figure 4: Average regret of not crossing threshold (HIGHS-REGRET) vs Horizons over the 2 instances and 2 thresholds

In this task we modify/adapt our algorithm from Task 3. We use the same strategy of using a Dirichlet distribution to track the expectation of each value in the reward support. We keep the counts of each reward from the reward support. For each arm, we have a k-length list of reward counts, where k is the size of finite reward support. In each run, for each arm we sample from a dirichlet distribution with the reward counts for that particular arm as the concentration parameters. To decide the arm to pull, we find the arm which we expect to give the highest number of rewards above the given threshold.

```
armToPull = np.argmax(np.array(armweights) @ threshedRewardList)
```

where `armweights` are the weights we receive from the Dirichlet distribution for every run and `threshedRewardList` is the finite reward support with binary thresholding. Say for example the support is [0, 0.25, 0.50, 0.75, 1] and the threshold is 0.6. The `threshedRewardList` is now [0, 0, 0, 1, 1]. This promotes the arms which have higher probability of getting a reward more than the given threshold.

Intuitively this works, because just like in the case of Beta distribution we try to learn the expected number of successes and failures, here we try to learn the expected number counts for each value in the reward support. And with the `threshedRewardList`, we can view each arm as bernoulli instances! So this method works for the same reasoning Thompson Sampling works.