Programming Assignment 1

Geet Sethi (23B2258)

September 11, 2025

1 Task 1

1.1 Upper Confidence Bound (UCB) Algorithm

- Keep track of the number of times each arm is pulled, the empirical means and the total number of pulls.
- To choose an arm to pull:
  - First K rounds: pull each arm once
  - Then select the arm with highest UCB value
- Finally, update the mean of the arm that was pulled according to the reward received: $\hat{\mu}_{t+1} = \frac{n-1}{n}\hat{\mu}_t + \frac{1}{n}r_t$

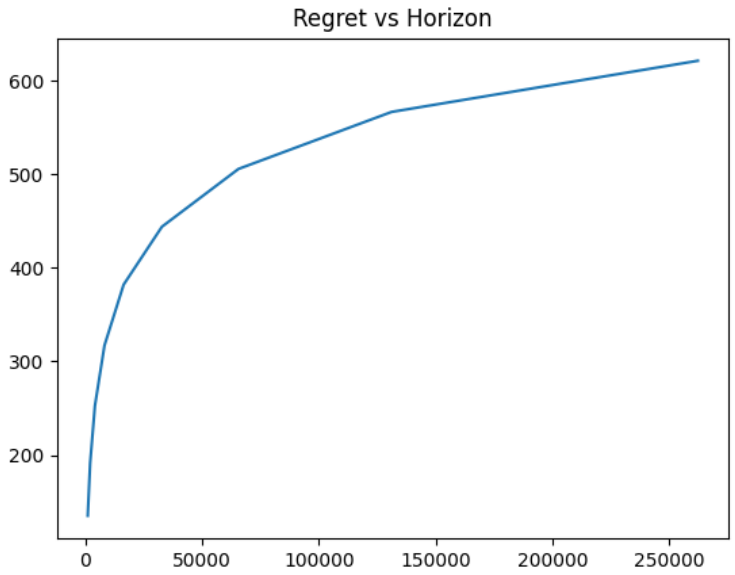

Figure 1: Regret vs Horizon for UCB

1.2 KL-UCB Algorithm

- Keep track of the number of times each arm is pulled, the empirical means and the total number of pulls.
- To choose an arm to pull:
  - First K rounds: pull each arm once

- Use binary search with initial bounds as $[\hat{\mu}_t, 1]$ to find the KL-UCB using the formula: $\text{KL-UCB}_i(t) = \max\{q : \text{KL}(\hat{\mu}_i, q) \leq \frac{\ln t + c \ln \ln t}{n_i}\}$. We continue to shrink the bounds until either the difference between the bounds becomes less than ϵ (which we choose) or until max iterations have been completed.
- Then select the arm with highest KL-UCB value

- Finally, update the mean of the arm that was pulled according to the reward received: $\hat{\mu}_{t+1} = \frac{n-1}{n}\hat{\mu}_t + \frac{1}{n}r_t$

1.2.1 Helper Functions

- **KL Divergence Calculation**: The `kl_div(p, q)` function computes the KL divergence between two Bernoulli distributions with numerical stability
- **KL-UCB Upper Bound Calculation**: The `calc_kl_ucb()` function uses binary search to find the upper confidence bound

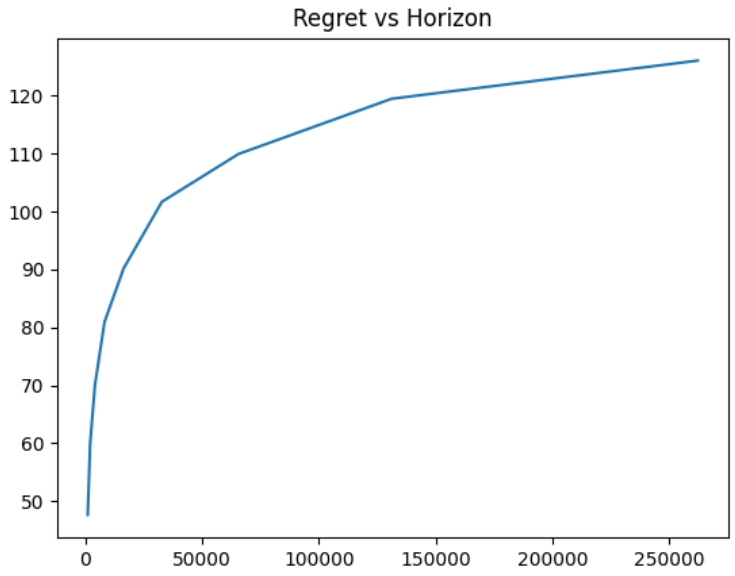

Figure 2: Regret vs Horizon for KL-UCB

1.3 Thompson Sampling Algorithm

- Keep track of the number of times each arm is pulled and the number of times the pull resulted in a success for each arm.
- To choose an arm to pull:
  - First K rounds: pull each arm once
  - Sample a value from the beta distribution with parameters $1 + s_t$ and $1 + f_t = 1 + u_t - s_t$ for each arm
  - Then select the arm with highest sampled value from the above step
- Finally, update the number of the successes of the arm that was pulled according to the reward received: $s_{t+1} = s_t + r_t$

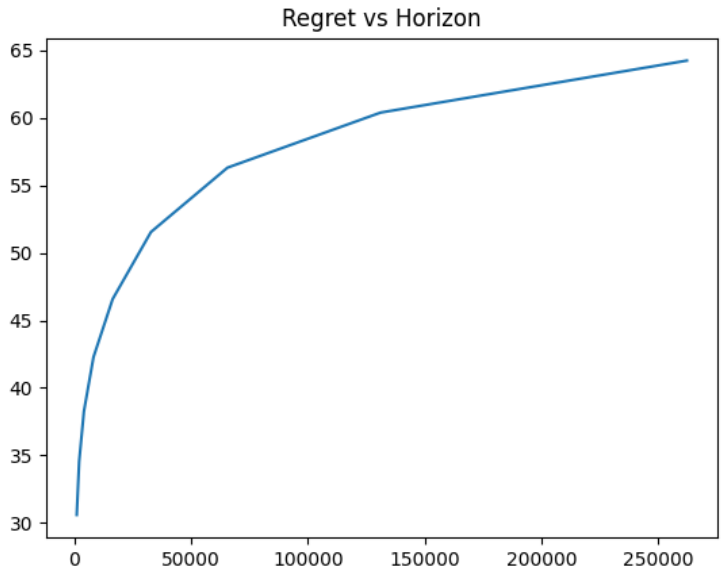

Figure 3: Regret vs Horizon for Thompson Sampling

2 Task 2

The problem is a variant of the multi-armed bandit problem but with an important distinction that we want to minimize the regret as much as possible in a small time horizon itself.

The core idea is to minimize expected time-to-termination. To achieve this we should prioritize doors that have the highest probability of breaking in the shortest time. This requires balancing two competing factors:

1. **High damage rate**: Doors with higher μ_i deal more damage on average
2. **Low remaining health**: Doors closer to breaking require less additional damage

2.1 Algorithm

- Keep track of the number of times each door is hit and their empirical means.
- To choose a door to pull:
  - First K rounds: pull each door once
  - Then select the door with lowest expected number of pulls required ($= \arg\min_i \frac{\text{health}[i]}{\sqrt{\hat{\mu}_i}}$) before it breaks
- Finally, update the mean and health of the door that was pulled according to the reward received: $\hat{\mu}_{t+1} = \frac{n-1}{n}\hat{\mu}_t + \frac{1}{n}r_t$

2.2 Mathematical Foundation

It is pretty simple to understand why a simple average gives a decent estimate of the mean. The square root scaling $\sqrt{\mu}$ in the denominator accounts for the variance in Poisson distributions, as the standard deviation of Poisson(μ) is $\sqrt{\mu}$.

Other approaches I tried (which didn't seem to give good results) are to follow the approach of Thompson Sampling but using a Gamma distribution to represent the belief distribution of the mean of each door instead of the beta distribution or use the KL-UCB like metric to get a *score* of the door and choose a door according to min moves required ($health/score$).

2.3 Computational Efficiency

The algorithm has:

- **Time Complexity**: $O(K)$ per step for selecting the door
- **Space Complexity**: $O(K)$ for storing reward sums, hit counts and healths per door

This makes it suitable for the constraint of up to 30 doors.

3 Task 3

3.1 Overview

The optimization leverages the observation that KL-UCB naturally exhibits long sequences of consecutive pulls of the same arm. Another thing to note is that this behaviour of KL-UCB becomes more pronounced as the algorithm converges to the optimal arm.

The standard implementation has a computational complexity of $O(K \times I)$ per time step, where:

- K is the number of arms
- I is the number of binary search iterations

This results in a total complexity of $O(T \times K \times I)$ over T time steps, making it computationally expensive for large horizons or many arms.

3.2 Optimization

The optimized algorithm implements a **batched approach**.

- We start with calculating the KL-UCB values for each arm and pull the arm with the highest value
- In the next timestep, we re-calculate the KL-UCB values for that arm and the one which was right behind it in the last iteration. If the arm that was pulled last time is no longer the arm with the highest value, we restart from the 1st step. But if it is, now we pull it for 1.5 times = 1 time.
- We continue doing the above and the batch size keeps increasing by a factor of 1.5 until the arm's KL-UCB value drops from the top spot.

This adaptive strategy ensures:

- Early exploration uses small batches (1-2 pulls) to maintain exploration capability
- Later exploitation uses larger batches (4, 6, 9, 13...) to maximize computational savings

The above aligns with how KL-UCB tends to behave as the algorithm starts to converge!

Another important point to note is that instead of recalculating the KL-UCB values for all arms after the completion of the batch, we only compute the KL-UCB values for the top 2 arms from the last calculation. This reduces the computational cost from $O(K \times I)$ to $O(2 \times I)$ per decision step.

3.3 Algorithm Pseudocode

3.4 Performance Analysis

3.4.1 Standard KL-UCB

- **Per step**: $O(K \times I)$ where $I \approx 25$ (binary search iterations)
- **Total**: $O(T \times K \times I)$

Algorithm 1 Optimized KL-UCB Algorithm

Initialize: counts $\leftarrow \mathbf{0}$, successes $\leftarrow \mathbf{0}$, batch_size $\leftarrow 1.0$
for $t = 1$ to T **do**
  if pulls_remaining_in_batch > 0 **then**
    pulls_remaining_in_batch $\leftarrow$ pulls_remaining_in_batch $- 1$
    return current_best_arm {Batch execution: O(1)}
  end if
  if any arm has not been pulled **then**
    Pull unexplored arm
    return arm index
  end if
  if kl_ucbs is not initialized **then**
    Compute UCB indices for all arms
    current_best_arm $\leftarrow \arg\max_i$ kl_ucbs$[i]$
  else
    Recalculate UCB for current_best_arm and second-best arm
    current_best_arm $\leftarrow \arg\max_i$ kl_ucbs$[i]$
  end if
  batch_size $\leftarrow$ batch_size $\times 1.5$
  pulls_remaining_in_batch $\leftarrow$ batch_size $- 1$
  return current_best_arm
end for

3.4.2 Optimized KL-UCB

- **Batched steps**: $O(1)$ (no UCB calculations)
- **Decision steps**: $O(2 \times I)$ (only 2 arms recalculated)
- **Total**: $O(T_{\text{batch}} \times 1 + T_{\text{decision}} \times 2 \times I)$

where $T_{\text{batch}} \gg T_{\text{decision}}$ in typical scenarios.

3.4.3 Regret Analysis

Similar to standard KL-UCB, this algorithm also follows all the GLIE rules and hence gives optimal regret and also closely follows the behaviour of standard KL-UCB like pulling the same arm again and again.

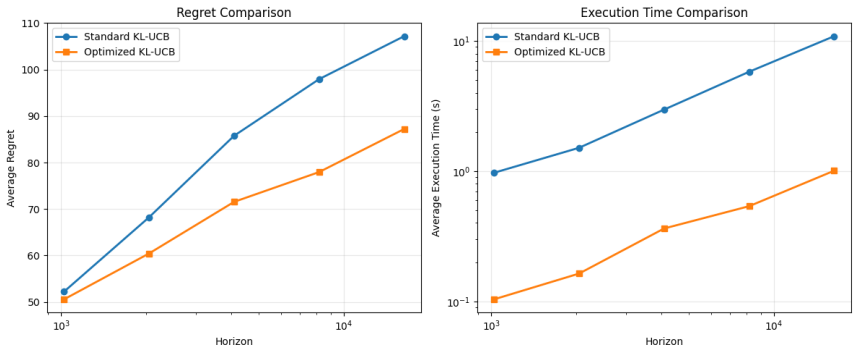

Figure 4: Comparison between Standard and Optimized KL-UCB

4 Bonus Task

One way to achieve exactly same trajectories is to employ a similar technique as the previously described algorithm but instead of just exponentially increasing the batch size, we calculate an optimal batch size in which we are sure that no other arm's KL-UCB value will not increase above the value of the arm that we have chosen.

We can calculate the exact future time t_j at which any other arm j's KL-UCB value will become equal to the value of the currently chosen arm, kl_ucb_{max}. We can do this by solving for t_j in the following equation:

$$N_j(t-1) \cdot d(\hat{\mu}_j(t-1), ucb_{max}) = \log(t_j) + c\log(\log(t_j))$$

We can ignore the $\log(\log(t))$ term for this calculation so as to simplify the calculation and speed up the algorithm further.

This t_j is the **earliest possible time** that arm j could become the leader. Before time t_j, we are **guaranteed** that $UCB_j(t') < ucb_{max}$ for all $t' < t_j$ and hence we can keep pulling that arm until t_j.