INF8250AE : Reinforcement Learning
Assignment Number 1

September 25, 2024

Fida CHERNI

Aragorn's Trials: The Quest to Find Middle-earth's True Champion

---

Problem Statement

In the twilight of the Third Age, as shadows deepen over Middle-earth, a dire threat looms over all free peoples. Sauron, the Dark Lord of Mordor, has regained much of his former strength, and his malice spreads across the lands like a rising tide. The fate of the world depends on the courage and strength of its heroes.

Aragorn, son of Arathorn, rightful heir to the throne of Gondor, must identify the hero with the highest probability of successfully completing the most dangerous and crucial quests. The true potential of each hero is unknown and must be inferred from their performance in various trials.

As Aragorn's advisor, your task is to devise a strategy using multi-armed bandit (MAB) algorithms to help identify the greatest hero. You must balance exploration, to discover hidden strengths, and exploitation, to maximize the power of known talents.

- **Heroes and Quests:**
  - A fellowship of N heroes, including Men, Elves, Dwarves, and Hobbits, will face the trials.
  - Each hero's true prowess is represented by a success probability p_i, unknown initially, reflecting their chances of success in any given quest.
  - Each quest is a Bernoulli trial, where a success or failure is drawn based on p_i.
- **Quests and Outcomes:**
  - In each round, Aragorn must choose one hero to undertake a quest.
  - After each quest, the success or failure is observed and used to guide future decisions.
  - The goal is to identify the hero with the highest estimated success probability by the end of the trials.

Question 1: Completing the Heroes Class

Problem Description

The `Heroes` class has been partially implemented. Your task is to complete the `attempt_quest` method to fulfill the following requirements:

1. Update the total number of quests and successes for the given hero.
2. Return the reward of the quest: return 1 for a success and 0 otherwise.

Example Setup

We will work with the following setup:

```
heroes = Heroes(total_quests=3000, true_probability_list=[0.35, 0.6, 0.1])
```

- `total_quests=3000` specifies the total number of quests to simulate.
- `true_probability_list=[0.35, 0.6, 0.1]` defines the true success probabilities for each hero. The first hero has a 35% chance of success, the second hero has a 60% chance, and the third hero has a 10% chance.

Response 1

The implementation of the `attempt_quest` method in the `Heroes` class is done in the `heroes.py` file. The description of the code:

```
def attempt_quest(self, hero_index):

    Parameters:
    hero_index (int): Index of the hero attempting the quest.

    Returns:
    int: 1 if the quest is successful, 0 otherwise.

    success = np.random.rand() < self.heroes[hero_index]['true_success_probability']
    self.heroes[hero_index]['total_quests'] += 1
    if reward:
        self.heroes[hero_index]['total_success'] += 1
    return reward
```

Question 2: Implementing the Epsilon-Greedy Method

Problem Description

The task is to implement the Epsilon-Greedy method in the `eps_greedy.py` file. The method should select a random action with probability ϵ after that we have the selection of the action with the highest estimated reward with probability $1-\epsilon$ and finally we have an update of the action-value estimates incrementally based on the observed rewards.

Response 2

The Epsilon-Greedy method was implemented in the `eps_greedy.py` file as follows:

```
def eps_greedy(heroes, eps, init_value):

    Parameters:
    heroes (Heroes): Instance of the Heroes class.
    eps (float): Exploration probability.
    init_value (float): Initial value estimates for each hero.

    Returns:
    Rewards, average returns, total regret, and optimal action counts.
```

The method selects a random action with probability ϵ and the best-known action with probability $1-\epsilon$. It updates the reward estimates using an incremental formula based on the observed rewards.

Figure: Performance of Epsilon-Greedy Method

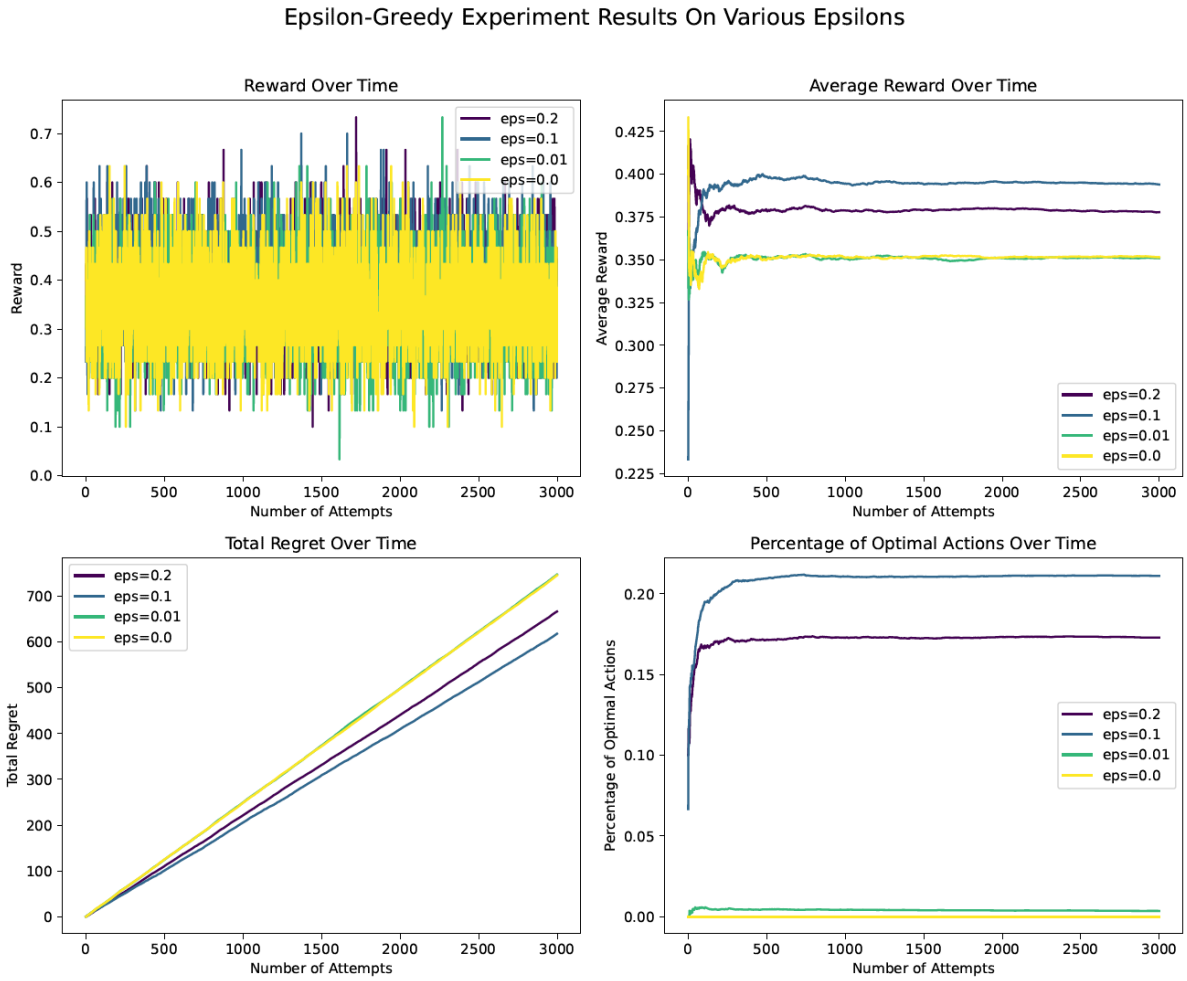

Figure 1: Performance of the Epsilon-Greedy method with various ϵ values.

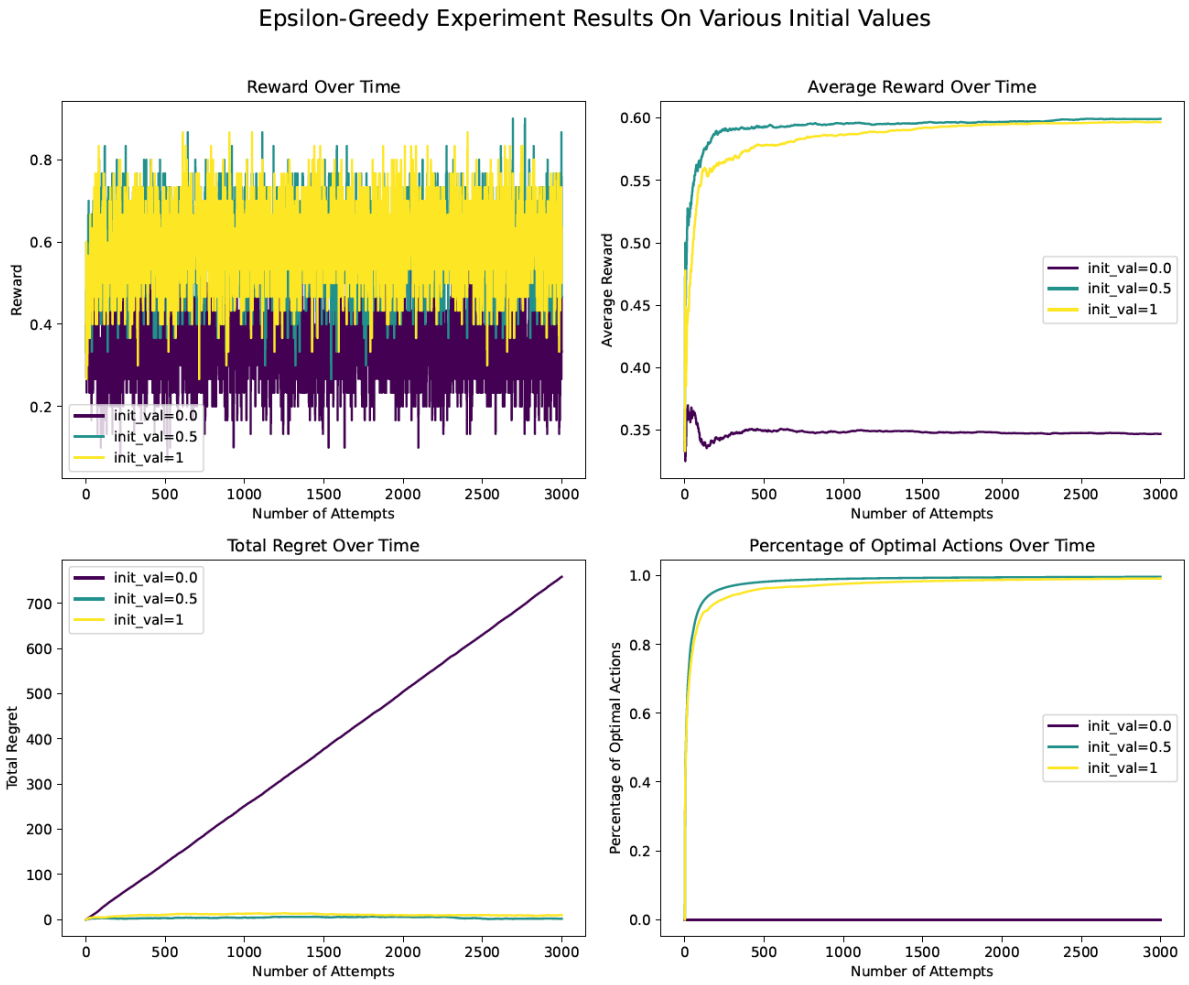

Figure 2: Impact of different initial values on the Epsilon-Greedy method.

Analysis

The figure above shows the performance of the Epsilon-Greedy method with different values of ϵ.

- We notice an increase of in expected reward as time goes to infinity and the agent seem to be learning which actions to pick in order to maximize gain.
- At the begining the 4 different ratios seem to induce the same behavior despite a little difference in around the 100th time step where it can be caused by difference in how actions are selected at that time if more than one action has the same optimal action value.
- The greedy way($\epsilon = 0$) seems to be faster at the begining to find optimal action but epsilon-greedy methods soon catch up and do a better performance-$\epsilon = 0.1$ is the

optimal one, The greedy method improved slightly faster than the other methods at the very beginning, but then leveled at t a lower level it achieved a reward average-per-step of only about 0.35, compared with the best possible of about 1.54

- The ϵ=0.1 greedy algorithm explores more than a greedy algorithm (which is stuck in suboptimal actions) and it is expected because we know in advance that this method will select the optimal action. The method mostly exploits the best-known action, leading to high rewards and low regret.

- 0.01 greedy algorithm seem to be slower to perform compared with 0.1 greedy version as it takes more time for more exploration 0.01 greedy algorithm seem to over-perform the 0.1 greedy version. ϵ greedy are expected to over-perform greedy algorithm since they explore more and they have more chances to see beyond what greedy sees.

- $\epsilon = 0.2$ and $\epsilon = 0.1$ show a much higher percentage of optimal actions, approaching 100%, as they stick to the known best action with higher variability, indicating more exploration and a mix of exploration and exploitation and show lower regret, indicating better long-term performance in which $\epsilon = 0.1$ shows the highest average reward over time, stabilizing around 0.4 and converges to the highest percentage of optimal actions, showing an effective balance between exploration and exploitation while $\epsilon = 0.0$, Minimal exploration, resulting in consistently low rewards and has the lowest average reward due to no exploration and thus limited learning that's mean almost no exploration, resulting in a low percentage of optimal actions as the algorithm cannot learn effectively, So $\epsilon = 0.1$ performs best overall, with the highest average reward and lowest regret.

- Initial values significantly affect the rewards. Higher initial values like 0.5 and 1.0 lead to better overall rewards, as they encourage more exploration, while initial value 0.0 leads to lower rewards due to lack of initial optimism. In term of average reward, higher initial values converge to a higher average reward around 0.6, showing the benefit of optimistic initial estimates. The initial value of 0.0 stabilizes around 0.35, indicating limited exploration. In term of total regret, initial value 0.0 results in high regret as the algorithm fails to explore and identify optimal actions effectively. Higher initial values lead to much lower regret as they quickly identify and exploit the best options. Finally, in term of percentage of optimal actions Over Time, higher initial values 0.5 and 1.0 rapidly achieve almost 100% optimal action selection, while the initial value of 0.0 lags significantly, indicating poor action choices.

The results indicate that a small amount of exploration improves performance, while too much or too little reduces efficiency and that an optimal initial values promote better exploration and faster convergence to optimal actions.

Question 3: Implementing the Upper-Confidence-Bound (UCB) Method

Problem Description

The goal of this question is to complete the implementation of the UCB method in the `ucb.py` file. The method should define the optimal reward and optimal hero index based on the true success probabilities. After that, we have to implement the UCB action-selection strategy, where the chosen action maximizes the upper confidence bound and finally return the following lists:

- `rew_record`: A list of rewards received at each attempt.
- `avg_ret_record`: The running average of rewards at each attempt.
- `tot_reg_record`: The cumulative regret at each attempt.
- `opt_action_record`: The percentage of times the optimal hero was selected at each attempt.

Response 3

The UCB method was completed in the `ucb.py` file as follows :

```
def ucb(heroes, c, init_value=0.):

    Parameters:
    heroes (Heroes): Instance of the Heroes class.
    c (float): Exploration coefficient.
    init_value (float): Initial value estimates for each hero.

    Returns:
    Rewards, average returns, total regret, and optimal action counts.

    Initialization of variables:
    values = [init_value] * len(heroes.heroes) : Initial value estimates
    rewards = list()
    avg_rewards = list()
    total_regret = list()
    optimal_action_count = list()
```

The UCB Method was implemented with exploration coefficient c and the values were updated based on observed rewards and the number of times each action was selected.

Figures: Impact of Different c Values on UCB Method

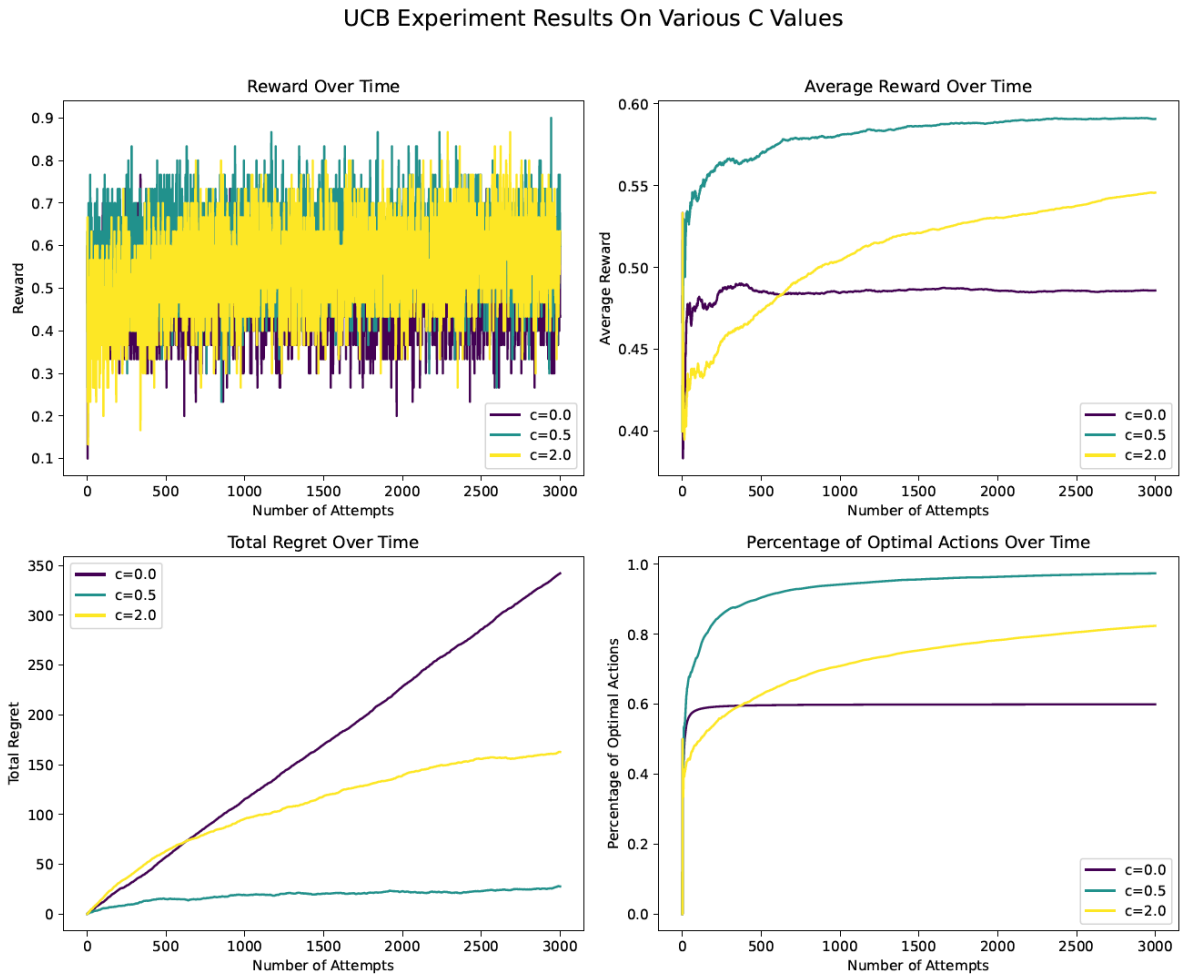

Figure 3: Performance of the UCB method with different c values (0.0, 0.5, 2.0).

Analysis

- The rewards fluctuate significantly for all c values. With $c = 0.5$, the rewards are more consistent and higher compared to $c = 0.0$ and $c = 2.0$. A high c value like 2.0 leads to increased variability due to excessive exploration.
- For the average reward, $c = 0.5$ has the highest average reward, stabilizing around 0.58. This shows that a moderate c value allows for effective balance between exploration and exploitation. $c = 2.0$ starts with high exploration but stabilizes at a lower average reward of around 0.5. $c = 0.0$ has the lowest average reward, as it relies solely on initial estimates without any exploration.

- The total regret for $c = 0.0$ is significantly higher than for the other values. This is because it does not explore at all. $c = 2.0$ shows a moderate level of regret, while $c = 0.5$ has the lowest total regret, indicating it finds and exploits the optimal action effectively.
- For the percentage of optimal actions over Time, $c = 0.5$ converges to almost 100% optimal action selection, demonstrating its effectiveness in finding the best option. $c = 2.0$ also approaches a high percentage but at a slower rate. $c = 0.0$ struggles to reach a high percentage of optimal actions, as it never explores to find the best option.

A moderate c value balances exploration and exploitation well, leading to high rewards and low regret. A high c value causes too much exploration, reducing efficiency and a lower c value results in poor performance due to lack of exploration.

Question 4: Implementing the Boltzmann Method

Problem Description

The objective of this question is to complete the implementation of the Boltzmann method in the `boltzmann.py` file. The method should complete the implementation for the boltzmann_policy method that gets an array and temperature value τ and returns an index sampled from the Boltzmann policy, we have to implement the Boltzmann action-selection strategy, where the probability of choosing an action is proportional to the exponential of the estimated value of the action divided by the temperature τ and finally to return the following lists:

- `rew_record`: A list of rewards received at each attempt.
- `avg_ret_record`: The running average of rewards at each attempt.
- `tot_reg_record`: The cumulative regret at each attempt.
- `opt_action_record`: The percentage of times the optimal hero was selected at each attempt.

Response 4

The Boltzmann method was completed in the `boltzmann.py` file as follows :

```
def boltzmann_policy(values, tau):

    Parameters:
    values (List[float]): Estimated values for each action.
```

```
    tau (float): Temperature parameter.

    Returns:
    int: Index of the selected action based on Boltzmann distribution.

    Calculate softmax probabilities:
    softmax_probabilities = np.exp(values / tau) / np.sum(np.exp(values / tau))

    Sample an action based on softmax probabilities:
    action = np.random.choice(len(values), p=softmax_probabilities)
    return action
```

The Boltzmann policy was implemented, which uses the softmax function to convert value estimates into probabilities. The actions were selected based on these probabilities, with higher values having a greater chance of being selected.

Figures: Impact of Different τ Values on Boltzmann Method

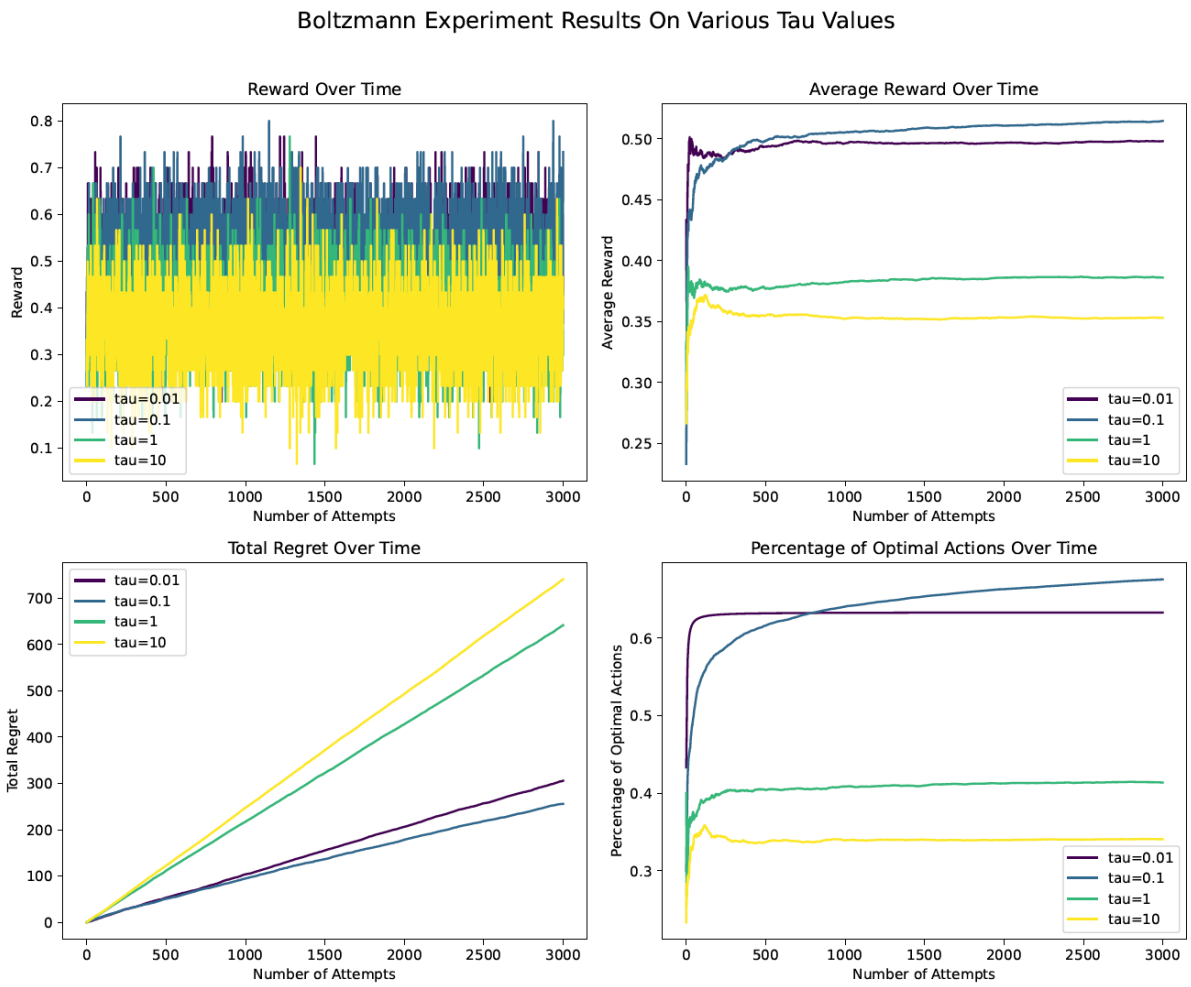

Figure 4: Performance of the Boltzmann method with different τ values (0.01, 0.1, 1, 10).

Analysis

- The rewards show different patterns depending on the τ value. For very high values like $\tau = 10$, the reward is low, indicating too much exploration. Lower values like $\tau = 0.01$ and $\tau = 0.1$ lead to more stable rewards over time, suggesting effective exploitation of known good actions.
- For the Average Reward, $\tau = 0.1$ achieves the highest average reward around 0.5. This shows that very low τ values favor exploitation and result in consistent

performance. $\tau = 0.01$ also performs well, reaching an average reward of around 0.48. Higher τ values like 1 and 10 show lower average rewards due to excessive exploration.

- The total regret is significantly lower for small τ values like 0.01 and 0.1. This indicates that these settings allow the algorithm to quickly identify and exploit the best actions. High τ values lead to high regret, as the algorithm keeps exploring even when the best actions are known.

- For the percentage of Optimal Actions, $\tau = 0.01$ and $\tau = 0.1$ reach a high percentage of optimal actions, indicating effective learning. $\tau = 10$ struggles to find the optimal action consistently, as evidenced by a low percentage of optimal choices. $\tau = 1$ performs moderately but does not achieve as high a percentage as lower τ values.

Very low τ values perform best, balancing exploration and exploitation effectively. High τ values lead to too much exploration, resulting in lower rewards and higher regret. A τ value of 1 offers a middle ground but is less efficient compared to smaller τ values.

Question 5: Implementing the Gradient Bandits Method

Problem Description

The goal of this question is to complete the implementation of the Gradient Bandit method in the `gradient_bandit.py` file. The method is to define the optimal reward and optimal hero index based on the true success probabilities by implementation of the Gradient Bandit action-selection strategy, ensuring the logits are updated correctly both when using the average return as a baseline and when not using a baseline and to return the following lists:

- `rew_record`: A list of rewards received at each attempt.
- `avg_ret_record`: The running average of rewards at each attempt.
- `tot_reg_record`: The cumulative regret at each attempt.
- `opt_action_record`: The percentage of times the optimal hero was selected at each attempt.

Response 5

The Gradient Bandits method was completed in the `gradient_bandit.py` file as follows:

```
def gradient_bandit(heroes, alpha, baseline=True):

    Parameters:
    heroes (Heroes): Instance of the Heroes class.
    alpha (float): Learning rate.
    baseline (bool): Whether to use the average reward as a baseline.

    Returns:
    Rewards, average returns, total regret, and optimal action counts.

    Initialize preference and running average reward:
    preferences = [0.0] * len(heroes.heroes)
    avg_reward = 0.0
```

Figures: Impact of Different α Values on Gradient Bandits Method

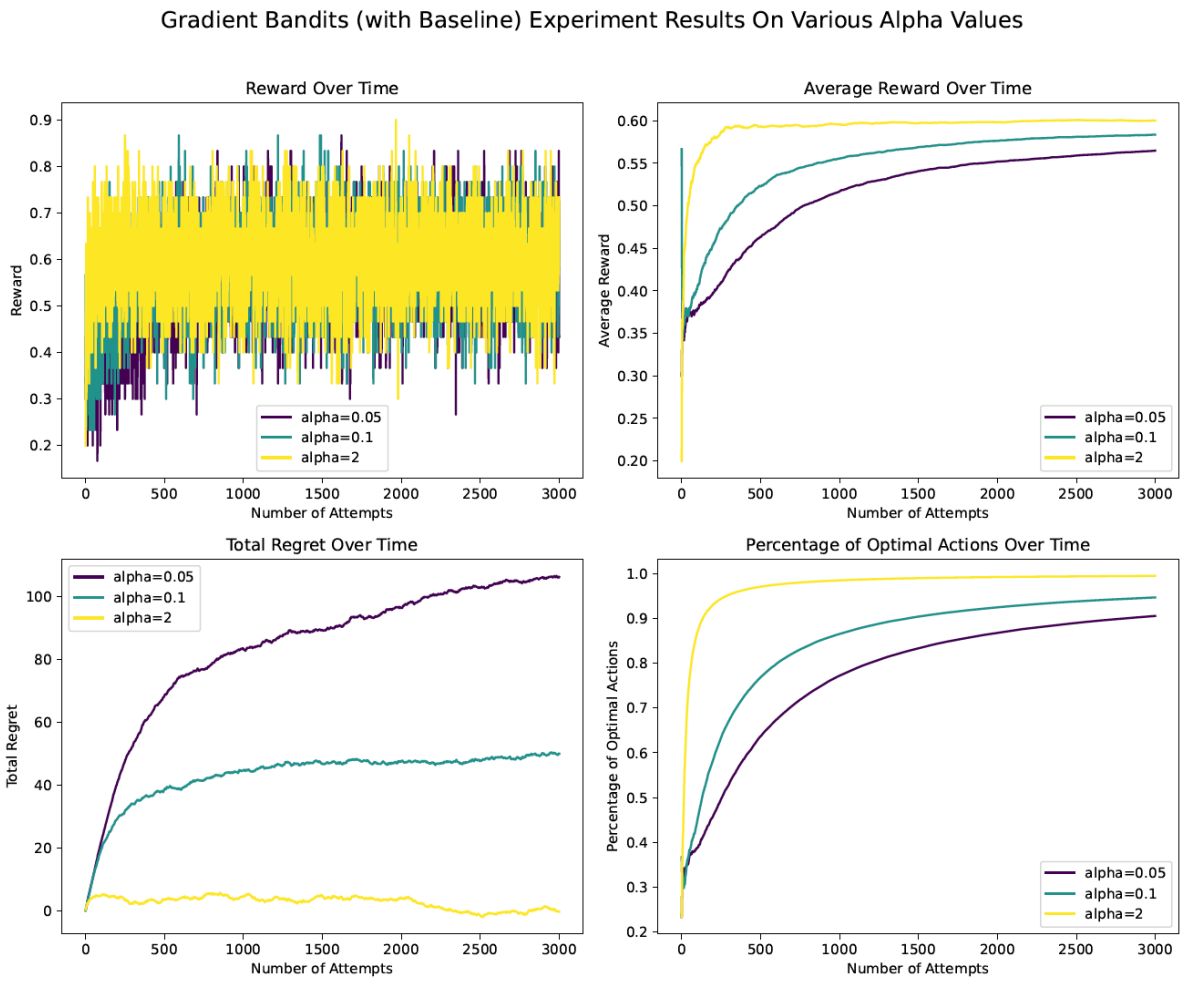

Figure 5: Performance of the Gradient Bandits method with different α values and baseline.

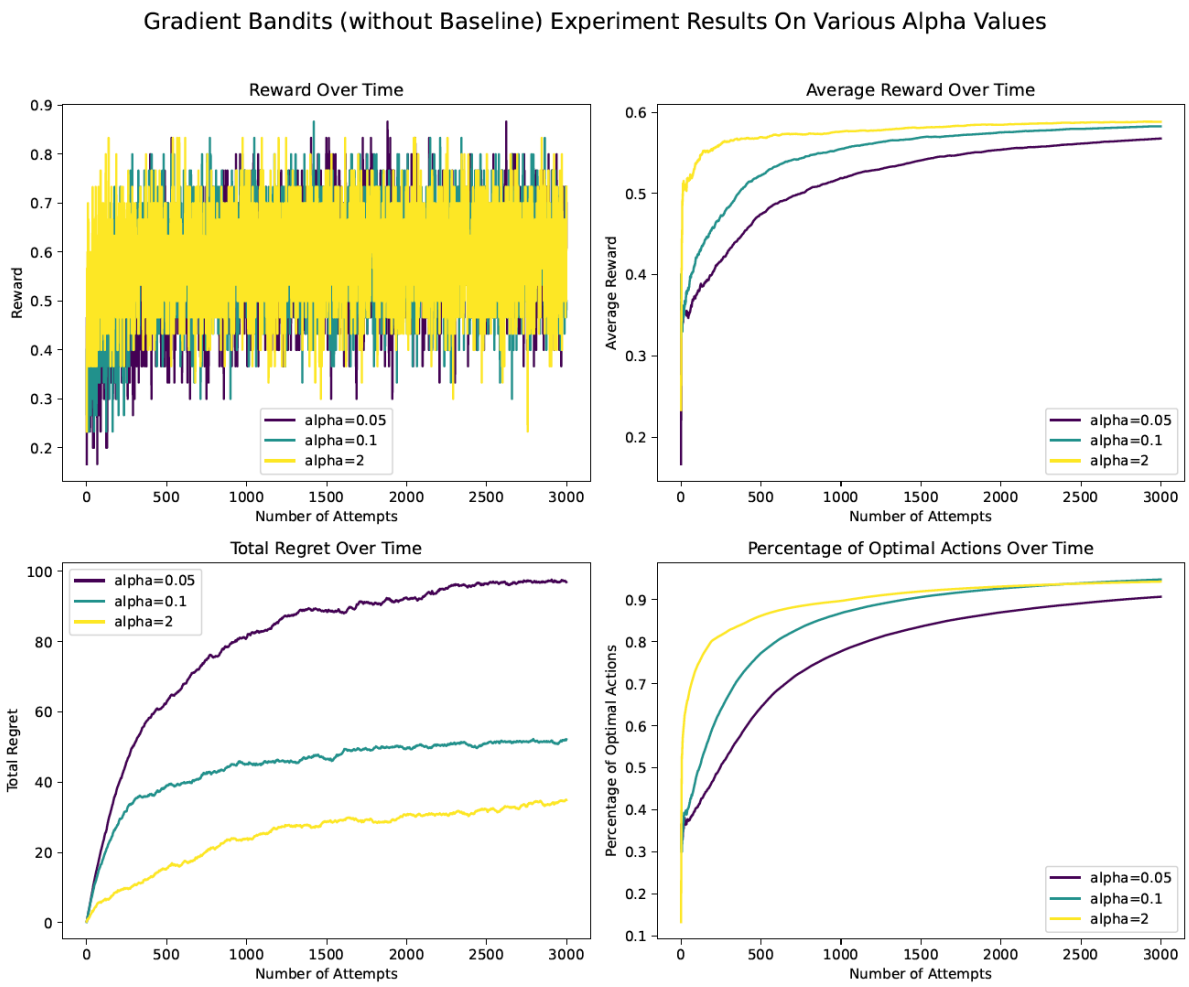

Figure 6: Performance of the Gradient Bandits method with different α values without baseline.

Analysis

- The use of a baseline leads to more stable rewards across all α values. $\alpha = 2$ shows the best performance with the most stable and highest rewards, indicating faster learning.
- Without Baseline, higher α values such as 2 result in more stable rewards over time, indicating faster convergence. Lower α values like 0.05 and 0.1 show more variability in rewards, reflecting slower adaptation.
- $\alpha = 2$ reaches an average reward of around 0.6 showing the benefit of using a baseline to stabilize learning. Lower α values also perform better compared to the no-baseline scenario, reaching higher average rewards.

- Without Baseline, $\alpha = 2$ achieves the highest average reward, stabilizing around 0.55, indicating effective learning. Lower α values take longer to converge and stabilize at lower average rewards.
- The total regret is significantly lower for all α with Baseline values compared to the no-baseline scenario, indicating that the baseline helps reduce suboptimal choices.
- Higher α values have lower total regret without Baseline, showing quicker convergence to the optimal actions. Lower α values exhibit a higher cumulative regret due to slower adaptation.
- $\alpha = 2$ reaches almost 100% optimal action selection very quickly, highlighting the advantage of using a baseline. Lower α values also achieve better performance than without a baseline, though they still converge slower than $\alpha = 2$.
- Without Baseline, $\alpha = 2$ reaches the highest percentage of optimal actions quickly, showing effective exploitation. Lower α values take longer to reach optimal performance.

Using a baseline significantly improves stability and performance across all α values, with $\alpha = 2$ showing the best results in terms of average reward, regret, and percentage of optimal actions.
In other words, without Baseline, higher learning rates like $\alpha = 2$ are effective for quickly learning the optimal actions but can cause instability. Lower α values adapt more slowly and have higher regret.
So, Using a baseline with a higher learning rate like $\alpha = 2$ is the most effective strategy, ensuring fast and stable convergence to optimal actions, and we gonna try another α value in the next question to analysis the sensibility of varying parameters.

Question 6: Ultimate Showdown: Tuning Parameters and Comparing Methods

Problem Description

The goal of this question is to optimize and evaluate the performance of various multi-armed bandit methods using the `compare.py` script. Specifically, we need to experiment with different parameters for each of the methods studied, select the most effective parameters for each method based on experimental results and finally, compare the performance of the different methods to determine the best strategy for Aragorn to choose the optimal hero for his quests.

Response 6

The following modifications were made in the `compare.py` file to fine-tune the parameters for each method: we tested different parameters for each method like ϵ values for ϵ-Greedy, c values for UCB, τ values for Boltzmann, and α values for Gradient Bandits and for each method, the parameter yielding the highest total reward or lowest regret was selected as the optimal parameter. We have chosen a random parameters for each algo :

- Epsilon-Greedy Method:
  - $\epsilon = 0.3$
  - `init_val` $= 0.0$
- Upper Confidence Bound (UCB) Method:
  - $c = 3$
  - `init_val` $= 0.0$
- Boltzmann Method:
  - $\tau = 0.2$
  - `init_val` $= 0.0$
- Gradient Bandits Method:
  - $\alpha = 1.55$
  - `use_baseline` $=$ True

Figure: Performance Comparison of Multi-Armed Bandit Methods

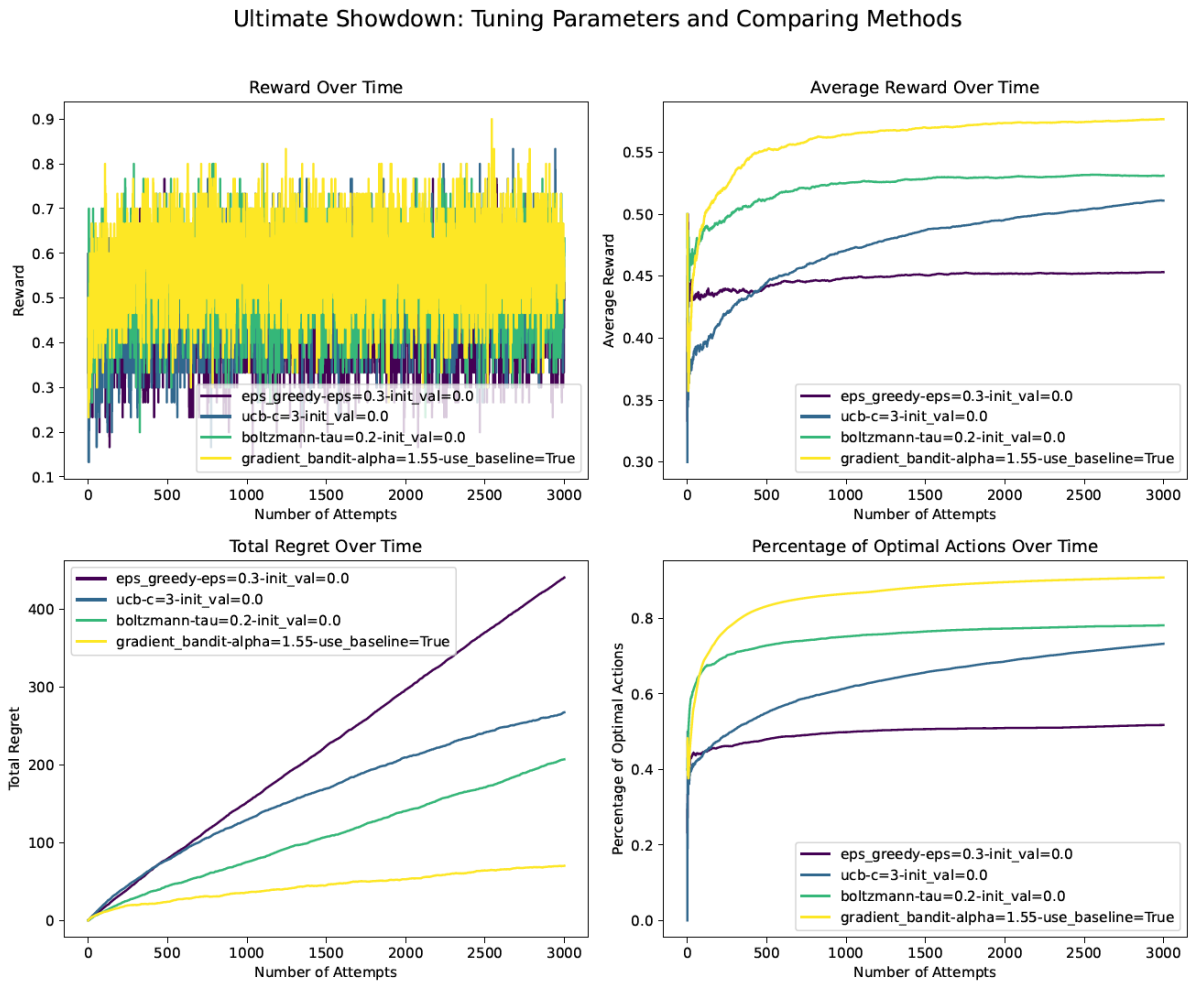

Figure 7: Comparison of different multi-armed bandit methods based on their optimal parameters.

Analysis

- The Gradient Bandit method with $\alpha = 1.55$ and baseline enabled consistently achieves the highest rewards over time, showing stable performance. The UCB method with $c = 3$ also performs well but with slightly more fluctuation. The Epsilon-Greedy method with $\epsilon = 0.3$ shows lower rewards, indicating insufficient exploration. The Boltzmann method with $\tau = 0.2$ has intermediate performance, balancing exploration and exploitation but not as effectively as the Gradient Ban-

dit.

- The Gradient Bandit method reaches the highest average reward, stabilizing around and more than 0.55. This shows that using a baseline with a well-tuned learning rate leads to efficient learning. The UCB method also converges to a high average reward but remains below the Gradient Bandit. The Boltzmann and Epsilon-Greedy methods take longer to stabilize and reach lower average rewards, indicating slower adaptation to the optimal action.

- The total regret is lowest for the Gradient Bandit method, indicating that it quickly identifies and exploits the optimal action. The UCB method shows moderate regret, while the Boltzmann method has higher regret due to its exploratory nature. The Epsilon-Greedy method has the highest regret, as it balances exploration and exploitation less effectively than the others.

- The Gradient Bandit method reaches nearly 90% optimal actions quickly, showing its effectiveness in selecting the best action consistently. The UCB method also performs well, reaching about 75% optimal actions. The Boltzmann method stabilizes around 65%, while the Epsilon-Greedy method struggles to reach 50%, indicating less effective exploration.

The Gradient Bandit method with the tuned $\alpha = 1.55$ and baseline enabled outperforms the other methods in all metrics, including average reward, total regret, and percentage of optimal actions. The UCB method with $c = 3$ is the next best, followed by the Boltzmann method with $\tau = 0.2$. The Epsilon-Greedy method with $\epsilon = 0.3$ shows the least effective performance, highlighting the importance of proper parameter tuning and strategy selection.

Upper Confidence Bounds in Multi-Armed Bandits (MABs)

Problem Statement

Consider the multi-armed bandit problem where at each stage $t = 1, \cdots, T$:

- The learner chooses an arm from a finite set A.

- Each arm $a \in A$ generates a random reward $r_t(a) \in [0, 1]$ with mean $E[r_t(a)] = \mu_a$.

Let $A_t \in A$ be the arm chosen at time t. The performance of a bandit algorithm A is commonly measured using regret, which quantifies the expected difference between

the accumulated reward obtained by always choosing the optimal arm and the reward obtained by following the algorithm. Formally, the regret after T rounds is defined as:

$$R_T(A) := \max_{a \in A} \mathbb{E}\left[\sum_{t=1}^{T} r_t(a)\right] - \mathbb{E}\left[\sum_{t=1}^{T} r_t(A_t)\right]$$

Let a^* denote the optimal arm, and define $N_T(a) := \sum_{t=1}^{T} \mathbb{I}\{A_t = a\}$ as the number of times arm a has been pulled after T rounds. Note that $N_T(a)$ depends on the realization of the rewards, so it is a random variable.

Question (a): Prove the Regret Formula

We need to Prove:

$$R_T(A) = \sum_{a \neq a^*} E[N_T(a)]\Delta_a,$$

where:

- $\Delta_a = \mu_{a^*} - \mu_a$ is the difference between the optimal arm's mean reward and arm a's mean reward.
- $E[N_T(a)]$ is the expected number of times arm a is selected.

Response (a) :

1. The regret measures the difference between the total reward we would have received by always choosing the optimal arm a^* and the total reward obtained by following the algorithm A.

   $$R_T(A) = E\left[\sum_{t=1}^{T} \mu_{a^*} - \sum_{t=1}^{T} r_t(A_t)\right]$$

   This lead to:

   $$R_T(A) = \sum_{t=1}^{T} \mu_{a^*} - \sum_{a \in A} E[N_T(a)]\mu_a,$$

   where $E[N_T(a)]$ is the expected number of times arm a was chosen.

2. Rearrange the regret formula:

   $$R_T(A) = \sum_{a \in A} E[N_T(a)](\mu_{a^*} - \mu_a).$$

   We have that $\mu_{a^*} - \mu_{a^*} = 0$, so we can ignore the term for $a = a^*$.

3. We sum only over optimal arms $a \neq a^*$:

   $$R_T(A) = \sum_{a \neq a^*} E[N_T(a)]\Delta_a.$$

Question (b): UCB Bound on Probability

Consider the UCB algorithm. It selects an arm at each round according to:

$$A_t = \arg\max_{a \in A} \text{ucb}_a(t),$$

where:

$$\text{ucb}_a(t) = Q_t(a) + \sqrt{\frac{2\log(1/\delta)}{N_t(a)}},$$

and $Q_t(a)$ is the empirical mean reward of arm a.
Remarking that $N_t(a) \leq t$, show that:

$$P\left(\mu_{a^*} \geq \min_{t=1,\ldots,T} \text{ucb}_{a^*}(t)\right) \leq T\delta.$$

Response (b) :

1. For all arms $a \in A$ and all $\epsilon \geq 0$:

$$P\left(\mu_a \geq Q_t(a) + \epsilon\right) \leq \exp\left(-\frac{t\epsilon^2}{2}\right).$$

   This gives us an upper bound on the probability that the true mean reward μ_a exceeds the empirical mean reward $Q_t(a)$ by more than ϵ.

2. We have to bound the Probability for UCB, so we need to substitute $\epsilon = \sqrt{\frac{2\log(1/\delta)}{N_t(a)}}$, we get:

$$P\left(\mu_a \geq Q_t(a) + \sqrt{\frac{2\log(1/\delta)}{N_t(a)}}\right) \leq \delta.$$

   Hence, for arm a^*, the probability that $\mu_{a^*} \geq \text{ucb}_{a^*}(t)$ is less than or equal to δ.

3. Using the union bound over $t = 1, \ldots, T$, we have:

$$P\left(\exists t \text{ such that } \mu_{a^*} \geq \text{ucb}_{a^*}(t)\right) \leq T\delta.$$

   Therefore, the probability that μ_{a^*} is greater than the minimum of the UCB estimates over time is bounded by $T\delta$.

With high probability, the mean of the optimal arm is not overestimated by the UCB algorithm. It helps us understand how the UCB algorithm confirms that the probability of not selecting the optimal arm decreases over time, leading to lower regret.