Final Project of SI140A

Team member:

熊闻野 2023533141

王珺 2023511040

仲之宸 2023533131

Contribution: All three members are of equal contributions:

熊闻野：Work on Problem 1, 2, 3 of Part I, and Problem 1, 2, 3, 4 of Part II. He also contributed to the whole project frame.

王珺： Work on Problem 3, 4, 5 of Part I, and Problem 4, 5 of Part II.

仲之宸： Work on Problem 4, 6 of Part I, and Problem 4, 5 of Part II.

More detailed contribution can be viewed through our GitHub Repository: Bandit Learning

Part I: Classical Bandit Algorithms

We consider a time-slotted bandit system $(t = 1, 2, \ldots)$ with three arms. We denote the arm set as $\{1, 2, 3\}$. Pulling each arm $j (j \in \{1, 2, 3\})$ will obtain a random reward r_j, which follows a Bernoulli distribution with mean θ_j, i.e., $\mathrm{Bern}(\theta_j)$. Specifically,

$$r_j = \begin{cases} 1, & w \cdot p \cdot \theta_j \\ 0, & w \cdot p \cdot 1 - \theta_j \end{cases}$$

where $\theta_j, j \in \{1, 2, 3\}$ are parameters within $(0, 1)$. Now we run this bandit system for $N (N \gg 3)$ time slots. In each time slot t, we choose one and only one arm from these three arms, which we denote as $I(t) \in \{1, 2, 3\}$. Then we pull the arm $I(t)$ and obtain a random reward $r_{I(t)}$. Our objective is to find an optimal policy to choose an arm $I(t)$ in each time slot t such that the expectation of the aggregated reward over N time slots is maximized, i.e.,

$$\max_{I(t), t=1, \ldots, N} \mathbb{E}\left[\sum_{t=1}^{N} r_{I(t)}\right]$$

If we know the values of $\theta_j, j \in \{1, 2, 3\}$, this problem is trivial. Since $r_{I(t)} \sim \mathrm{Bern}(\theta_{I(t)})$,

$$\mathbb{E}\left[\sum_{t=1}^{N} r_{I(t)}\right] = \sum_{t=1}^{N} \mathbb{E}\left[r_{I(t)}\right] = \sum_{t=1}^{N} \theta_{I(t)}$$

Let $I(t) = I^* = \arg\max \theta_j$ for $t = 1, 2, \ldots, N$, then

$$\max_{I(t), t=1,\ldots,N} \mathbb{E}\left[\sum_{t=1}^{N} r_{I(t)}\right] = N \cdot \theta_{I^*}$$

However, in reality, we do not know the values of $\theta_j, j \in \{1, 2, 3\}$. We need to estimate the values $\theta_j, j \in \{1, 2, 3\}$ via empirical samples, and then make the decisions in each time slot. Next we introduce three classical bandit algorithms: ϵ-greedy, UCB, and TS, respectively.

(1). ϵ-greedy Algorithm ($0 \le \epsilon \le 1$)

Algorithm 1 ϵ-greedy Algorithm

Initialize $\widehat{\theta}(j) \leftarrow 0, \mathrm{count}(j) \leftarrow 0, j \in \{1, 2, 3\}$

for $t = 1, 2 \ldots, N$ **do**

$$I(t) \leftarrow \begin{cases} \arg\max_{j \in \{1,2,3\}} \widehat{\theta}(j) & w.p.\ 1-\epsilon \\ \text{randomly chosen from}\{1,2,3\} & w.p.\ \epsilon \end{cases}$$

$\mathrm{count}(I(t)) \leftarrow \mathrm{count}(I(t)) + 1$

$\widehat{\theta}(I(t)) \leftarrow \widehat{\theta}(I(t)) + \frac{1}{\mathrm{count}(I(t))}\left[r_{I(t)} - \widehat{\theta}(I(t))\right]$

end for

(2). UCB (Upper Confidence Bound) Algorithm

Algorithm 2 UCB Algorithm

for $t = 1, 2, 3$ **do**

$I(t) \leftarrow t$

$\mathrm{count}(I(t)) \leftarrow 1$

$\widehat{\theta}(I(t)) \leftarrow r_{I(t)}$

end for

for $t = 4, \ldots, N$ **do**

$$I(t) \leftarrow \arg\max_{j \in \{1,2,3\}} \left(\widehat{\theta}(j) + c \cdot \sqrt{\frac{2\log(t)}{\mathrm{count}(j)}}\right)$$

$\mathrm{count}(I(t)) \leftarrow \mathrm{count}(I(t)) + 1$

$\widehat{\theta}(I(t)) \leftarrow \widehat{\theta}(I(t)) + \frac{1}{\mathrm{count}(I(t))}\left[r_{I(t)} - \widehat{\theta}(I(t))\right]$

end for

Note: c is a positive constant with a default value of 1.

(3). TS (Thompson Sampling) Algorithm

Recall that $\theta_j, j \in \{1, 2, 3\}$ are unknown parameters over $(0, 1)$. From the Bayesian perspective, we assume their priors are Beta distributions with parameters (α_j, β_j).

Algorithm 3 TS Algorithm

Initialize Beta parameter $(\alpha_j, \beta_j), j \in \{1, 2, 3\}$

for $t = 1, 2 \ldots, N$ **do**

 # Sample model

for $j \in \{1, 2, 3\}$ **do**

Sample $\widehat{\theta}(j) \sim \mathrm{Beta}(\alpha_j, \beta_j)$

end for

 # Select and pull the arm

$$I(t) \leftarrow \arg\max_{j \in \{1,2,3\}} \widehat{\theta}(j)$$

 # Update the distribution

$$\alpha_{I(t)} \leftarrow \alpha_{I(t)} + r_{I(t)}$$
$$\beta_{I(t)} \leftarrow \beta_{I(t)} + 1 - r_{I(t)}$$

end for

Problems 1

Question

Now suppose we obtain the parameters of the Bernoulli distributions from an oracle, which are shown in the following table. Choose $N = 5000$ and compute the theoretically maximized expectation of aggregate rewards over N time slots. We call it the oracle value. Note that these parameters $\theta_j, j \in \{1, 2, 3\}$ and oracle values are unknown to all bandit algorithms.

Arm j	1	2	3
θ_j	0.7	0.5	0.4

Solution

Since each arm's parameter is known from the oracle, we need to choose the arm with the largest parameter to maximize the expectation of aggregate rewards over N time slots.

Given $\theta_1 = 0.7, \theta_2 = 0.5, \theta_3 = 0.4$, we have $\theta_1 > \theta_2 > \theta_3$. Thus, we choose arm 1 every time.

i.e.

$$\forall t, I(t) = I^* = \arg \max_{j \in \{1,2,3\}} \theta_j = 1$$

$$\theta_{I(t)} = \theta_1 = 0.7$$

Since $r_{I(t)} \sim \mathrm{Bern}(\theta_{I(t)})$,

$$E(r_{I(t)}) = \theta_{I(t)}$$

The maximum expected value is

$$\max_{I(t), t=1,2,\cdots,N} E[\sum_{t=1}^{N} r_{I(t)}]$$

$$= \max_{I(t), t=1,2,\cdots,N} \sum_{t=1}^{N} E[r_{I(t)}]$$

$$= N \cdot \theta_{I^*} = 5000 \times 0.7 = 3500$$

Therefore, with the given oracle parameters, the maximum expected value is 3500.

Problem 2

Question

2. Implement classical bandit algorithms with following settings:
   - $N = 5000$
   - ϵ-greedy with $\epsilon \in \{0.1, 0.5, 0.9\}$.
   - UCB with $c \in \{1, 5, 10\}$.
   - TS with $\{(\alpha_1, \beta_1) = (1, 1), (\alpha_2, \beta_2) = (1, 1), (\alpha_3, \beta_3) = (1, 1)\}$ and $\{(\alpha_1, \beta_1) = (601, 401), (\alpha_2, \beta_2) = (401, 601), (a3, b3) = (2, 3)\}$

Solution

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

class Bandit:
    def __init__(self, theta=[0.7, 0.5, 0.4]):
        self.theta = theta
        self.n_arms = len(theta)
        self.counts = np.zeros(self.n_arms)
        self.values = np.zeros(self.n_arms)

    def pull(self, arm):
        return np.random.binomial(1, self.theta[arm])
```

```
    def update(self, arm, reward):
        self.counts[arm] += 1
        n = self.counts[arm]
        value = self.values[arm]
        self.values[arm] = ((n - 1) / n) * value + (1 / n) * reward

class EpsilonGreedy(Bandit):
    def __init__(self, epsilon, theta=[0.7, 0.5, 0.4]):
        super().__init__(theta)
        self.epsilon = epsilon

    def select_arm(self):
        if np.random.random() < self.epsilon:
            return np.random.randint(self.n_arms)
        if np.all(self.values == self.values[0]):
            return np.random.randint(self.n_arms)
        return np.argmax(self.values)

    def modify_parameter(self, epsilon):
        self.epsilon = epsilon

class UCB(Bandit):
    def __init__(self, c, theta=[0.7, 0.5, 0.4]):
        super().__init__(theta)
        self.c = c

    def select_arm(self):
        for arm in range(self.n_arms):
            if self.counts[arm] == 0:
                return arm

        total_counts = sum(self.counts)
        ucb_values = self.values + self.c * np.sqrt(2 * np.log(total_coun
        return np.argmax(ucb_values)

    def modify_parameter(self, c):
        self.c = c

class ThompsonSampling(Bandit):
    def __init__(self, alpha=[1,1,1], beta=[1,1,1], theta=[0.7, 0.5, 0.4]
        super().__init__(theta)
        self.alpha = np.array(alpha)
        self.beta = np.array(beta)

    def select_arm(self):
        samples = [np.random.beta(self.alpha[i], self.beta[i]) for i in r
        return np.argmax(samples)

    def update(self, arm, reward):
        super().update(arm, reward)
        self.alpha[arm] += reward
        self.beta[arm] += (1 - reward)

    def modify_parameter(self, alpha, beta):
        self.alpha = np.array(alpha)
        self.beta = np.array(beta)
```

Problem 3

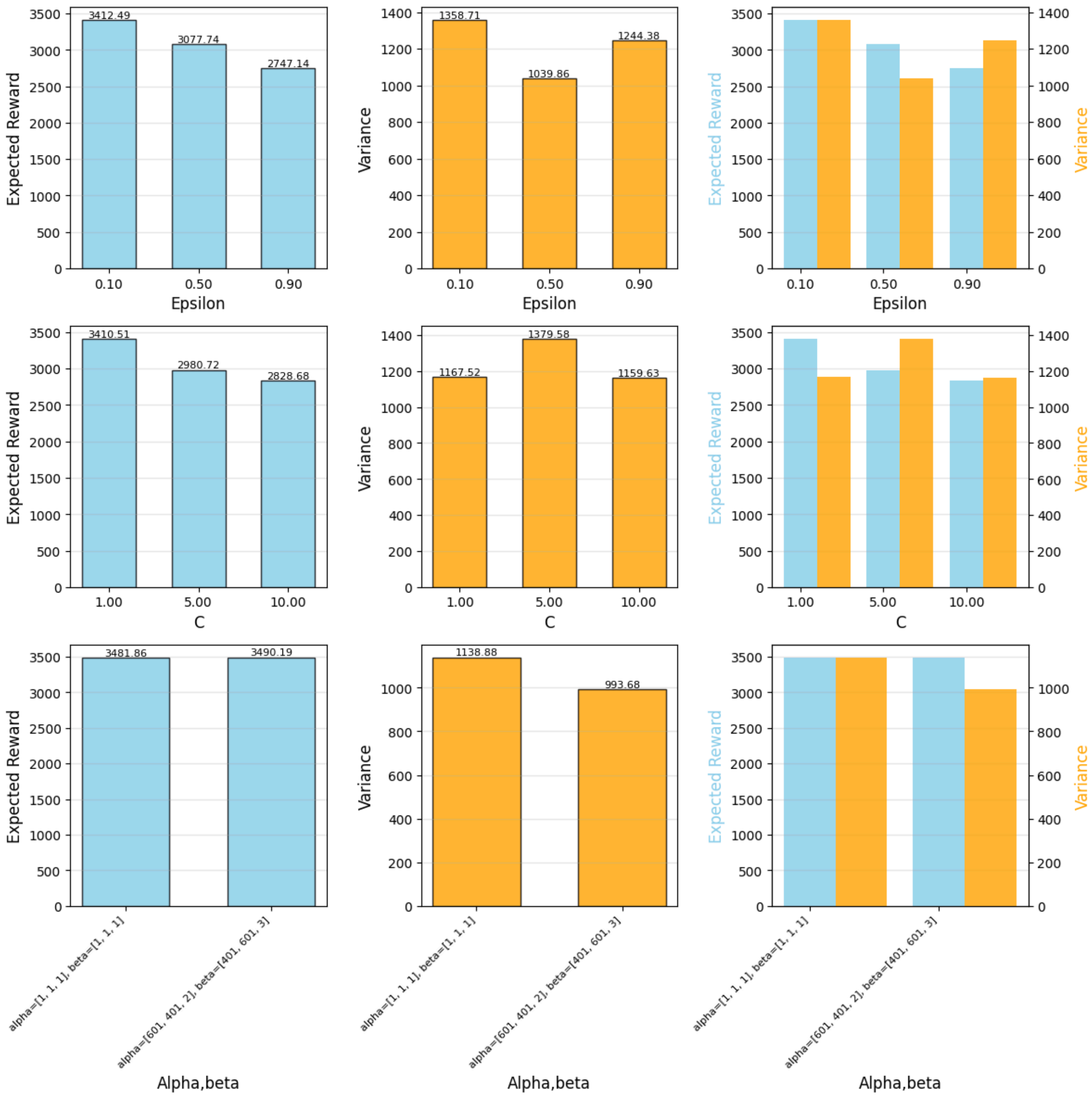
Each experiment lasts for N = 5000 time slots, and we run each experiment 200 trials. Results are averaged over these 200 independent trials.

In [5]:
```python
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
experiments = {
    'epsilon_greedy': [0.1, 0.5, 0.9],
    'ucb': [1, 5, 10],
    'ts': [
        ([1, 1, 1], [1, 1, 1]),
        ([601, 401, 2], [401, 601, 3])
    ]
}
results_rewards = {
    'epsilon_greedy': [],
    'ucb': [],
    'ts': []
}
results_variances = {
    'epsilon_greedy': [],
    'ucb': [],
    'ts': []
}
results_regrets = {
    'epsilon_greedy': [],
    'ucb': [],
    'ts': []
}

for key in ['epsilon_greedy', 'ucb', 'ts']:
    for value in experiments[key]:
        trial_rewards_list = []
        cumulative_rewards = 0
        cumulative_regrets = 0

        for _ in range(num_trials):
            if key == 'epsilon_greedy':
                bandit = EpsilonGreedy(epsilon=value)
            elif key == 'ucb':
                bandit = UCB(c=value)
            elif key == 'ts':
                bandit = ThompsonSampling(alpha=value[0], beta=value[1])

            trial_rewards = []
            for t in range(N):
                arm = bandit.select_arm()
                reward = bandit.pull(arm)
                bandit.update(arm, reward)
                trial_rewards.append(reward)
                cumulative_rewards += reward
                regret = max(bandit.theta) - bandit.theta[arm]
                cumulative_regrets += regret

            trial_rewards_list.append(np.sum(trial_rewards))

        mean_reward = cumulative_rewards / num_trials
```

```
        variance_reward = np.var(trial_rewards_list)

        results_rewards[key].append(mean_reward)
        results_variances[key].append(variance_reward)
        results_regrets[key].append(cumulative_regrets / num_trials)

print("epsilon = 0.1   reward: ", results_rewards['epsilon_greedy'][0], "
print("epsilon = 0.5   reward: ", results_rewards['epsilon_greedy'][1], "
print("epsilon = 0.9   reward: ", results_rewards['epsilon_greedy'][2], "
print("c = 1          reward: ", results_rewards['ucb'][0], " regret: ",
print("c = 5          reward: ", results_rewards['ucb'][1], " regret: ",
print("c = 10         reward: ", results_rewards['ucb'][2], " regret: ",
print("alpha = [1,1,1] beta = [1,1,1] reward: ", results_rewards['ts'][0]
print("alpha = [601,401,2] beta = [401,601,3] reward: ", results_rewards[

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
```

```
        ax.set_xticklabels([f"{epsilon:.2f}" for epsilon in param_values]

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(3, 3, figsize=(12, 12))

# epsilon_greedy
plot_expectation(axs[0, 0], results_rewards['epsilon_greedy'], experiment
plot_variance(axs[0, 1], results_variances['epsilon_greedy'], experiments
plot_combined(axs[0, 2], results_rewards['epsilon_greedy'], results_varia

# ucb
plot_expectation(axs[1, 0], results_rewards['ucb'], experiments['ucb'], "
plot_variance(axs[1, 1], results_variances['ucb'], experiments['ucb'], "c
plot_combined(axs[1, 2], results_rewards['ucb'], results_variances['ucb']

# ts
plot_expectation(axs[2, 0], results_rewards['ts'], experiments['ts'], "al
plot_variance(axs[2, 1], results_variances['ts'], experiments['ts'], "alp
plot_combined(axs[2, 2], results_rewards['ts'], results_variances['ts'],

plt.tight_layout()
plt.show()
```

```
epsilon = 0.1   reward:  3412.49  regret:  87.35449999999899  variances:
1358.7098999999998
epsilon = 0.5   reward:  3077.745  regret:  418.9664999999875  variances:
1039.8599749999996
epsilon = 0.9   reward:  2747.135  regret:  749.8360000000154  variances:
1244.376775
c = 1          reward:  3410.505  regret:  90.27000000004253  variances:
1167.5199750000002
c = 5          reward:  2980.715  regret:  519.4694999994001  variances:
1379.5837750000005
c = 10         reward:  2828.685  regret:  672.7809999995737  variances:
1159.6257750000002
alpha = [1,1,1] beta = [1,1,1] reward:  3481.865  regret:  18.158499999998
494  variances:  1138.876775
alpha = [601,401,2] beta = [401,601,3] reward:  3490.19  regret:  9.158999
999999068  variances:  993.6838999999999
```



Additionally, we can visualize the evolution of the estimated winning probability for each arm over time.

In [6]:
```
import numpy as np
import matplotlib.pyplot as plt

def plot_arm_reward_probability(algorithms_params, N=500, num_trials=20):
    fig, axs = plt.subplots(3, 3, figsize=(15, 12))
    axs = axs.flatten()

    plot_idx = 0

    for i, (alg_name, params) in enumerate(algorithms_params.items()):
        for j, param in enumerate(params):
            reward_probabilities = np.zeros((N, 3))

            for trial in range(num_trials):
                if alg_name == 'epsilon_greedy':
                    bandit = EpsilonGreedy(epsilon=param)
                elif alg_name == 'ucb':
                    bandit = UCB(c=param)
                elif alg_name == 'ts':
                    alpha = [1, 1, 1]
                    beta = [1, 1, 1]
```

```
                bandit = ThompsonSampling(alpha=alpha, beta=beta)

            arm_counts = np.zeros(3)
            arm_rewards = np.zeros(3)
            for t in range(N):
                arm = bandit.select_arm()
                reward = bandit.pull(arm)
                bandit.update(arm, reward)
                arm_rewards[arm] += reward
                arm_counts[arm] += 1

                reward_probabilities[t,arm] = arm_rewards[arm] / arm_
                if arm == 0:
                    if arm_counts[arm + 1] != 0 and arm_counts[arm +
                        reward_probabilities[t,arm + 1] = arm_rewards
                        reward_probabilities[t,arm + 2] = arm_rewards
                    elif arm_counts[arm + 1] == 0 and arm_counts[arm
                        reward_probabilities[t,arm + 1] = 0
                        reward_probabilities[t,arm + 2] = arm_rewards
                    elif arm_counts[arm + 1] != 0 and arm_counts[arm
                        reward_probabilities[t,arm + 1] = arm_rewards
                        reward_probabilities[t,arm + 2] = 0
                    elif arm_counts[arm + 1] == 0 and arm_counts[arm
                        reward_probabilities[t,arm + 1] = 0
                        reward_probabilities[t,arm + 2] = 0

                elif arm == 1:
                    if arm_counts[arm + 1] != 0 and arm_counts[arm -
                        reward_probabilities[t,arm + 1] = arm_rewards
                        reward_probabilities[t,arm - 1] = arm_rewards
                    elif arm_counts[arm + 1] == 0 and arm_counts[arm
                        reward_probabilities[t,arm + 1] = 0
                        reward_probabilities[t,arm - 1] = arm_rewards
                    elif arm_counts[arm + 1] != 0 and arm_counts[arm
                        reward_probabilities[t,arm + 1] = arm_rewards
                        reward_probabilities[t,arm - 1] = 0
                    elif arm_counts[arm + 1] == 0 and arm_counts[arm
                        reward_probabilities[t,arm + 1] = 0
                        reward_probabilities[t,arm - 1] = 0

                elif arm == 2:
                    if arm_counts[arm - 1] != 0 and arm_counts[arm -
                        reward_probabilities[t,arm - 1] = arm_rewards
                        reward_probabilities[t,arm - 2] = arm_rewards
                    elif arm_counts[arm - 1] == 0 and arm_counts[arm
                        reward_probabilities[t,arm - 1] = 0
                        reward_probabilities[t,arm - 2] = arm_rewards
                    elif arm_counts[arm - 1] != 0 and arm_counts[arm
                        reward_probabilities[t,arm - 1] = arm_rewards
                        reward_probabilities[t,arm - 2] = 0
                    elif arm_counts[arm - 1] == 0 and arm_counts[arm
                        reward_probabilities[t,arm - 1] = 0
                        reward_probabilities[t,arm - 2] = 0

        axs[plot_idx].plot(reward_probabilities)
        axs[plot_idx].set_title(f"{alg_name.capitalize()} ({param})")
        axs[plot_idx].set_xlabel("Steps")
        axs[plot_idx].set_ylabel("Reward Probability")
        axs[plot_idx].legend([f"Arm {k}" for k in range(3)])
        axs[plot_idx].grid(True)
```

```
            plot_idx += 1

    for i in range(plot_idx, len(axs)):
        fig.delaxes(axs[i])

    plt.tight_layout()
    plt.show()

algorithms_params = {
    'epsilon_greedy': [0.1, 0.5, 0.9],
    'ucb': [1, 5, 10],
    'ts': [
        [1, 1, 1],
        [601, 401, 2]
    ]
}

plot_arm_reward_probability(algorithms_params)
```

We find that as the number of pulling times increases, the estimated value we get is closer to the true value, which proves the validity of our experiment. At the same time, we also find that in Epsilon-greedy algorithm, when epsilon is small (equal to 0.1 in this case), the probability estimate obtained is not accurate, which is in line with our intuition, and we are more inclined to select the value we currently think is most likely to win, ignoring the importance of exploration. This also tells us that we should balance the relationship between exploration and application, and carry out rational updates in the course of continuous exploration, which will allow us to obtain accurate probability estimation and be more conducive to our application.

Problem 4

In [7]:
```
print("epsilon = 0.1   reward_gap: ", results_rewards['epsilon_greedy'][0
print("epsilon = 0.5   reward_gap: ", results_rewards['epsilon_greedy'][1
print("epsilon = 0.9   reward_gap: ", results_rewards['epsilon_greedy'][2
print("c = 1           reward_gap: ", results_rewards['ucb'][0]-3500)
print("c = 5           reward_gap: ", results_rewards['ucb'][1]-3500)
print("c = 10          reward_gap: ", results_rewards['ucb'][2]-3500)
print("alpha = [1,1,1] beta = [1,1,1] reward_gap: ", results_rewards['ts'
print("alpha = [601,401,2] beta = [401,601,3] reward_gap: ", results_rewa
```

```
epsilon = 0.1   reward_gap:  -87.51000000000022
epsilon = 0.5   reward_gap:  -422.2550000000001
epsilon = 0.9   reward_gap:  -752.8649999999998
c = 1           reward_gap:  -89.49499999999989
c = 5           reward_gap:  -519.2849999999999
c = 10          reward_gap:  -671.315
alpha = [1,1,1] beta = [1,1,1] reward_gap:  -18.13500000000022
alpha = [601,401,2] beta = [401,601,3] reward_gap:  -9.809999999999945
```

We calculated the gap between them and the oracle value using the expectations obtained above.

0 Comparison between different algorithms

Summary of the above results

ϵ-greedy algorithm

From the data obtained in our experiment, we can find that the score of Epsilon-greedy algorithm is very small when epsilon is large, which conforms to our cognition, but it still maintains exploration in the later stage and ignores the information already obtained.

UCB algorithm

The UCB algorithm has the highest score when c=1, but it is not stable from the variance observation.

TS algorithm

When we modify the two parameters of the TS algorithm, their scores and variances will change, which means that we get different prior information, which also inspires us to obtain a better scheme by modifying the prior information.

It is found from the above that when the prior of beta and alpha is set, the expected value obtained is the highest and closest to the ideal value. For Epsilon-greedy algorithm, the value of epsilon is small, and its expected value is also relatively high and approaches the ideal value, while the gap between UCB algorithm and the ideal value is the largest.But we found that none of them reached the ideal value (3,500).

1 Exploration of Algorithm

1.1 Further exploration of ϵ - greedy Algorithm

What we've done before is treat ϵ as a constant:[0.1, 0.5, 0.9], but we can think about it in two ways; The first point is: by changing the values of different epsilon, but requiring the value interval to be reduced, more epsilon values can be calculated, and the influence of different epsilon values on reward can be carefully calculated. Second, we can set epsilon as a function of t, that is, our exploration strategy is constantly changing and iterating and this case is real in life .Because we can look at it on a case-by-case basis and change our strategy based on current results, so as to explore and optimize the algorithm.

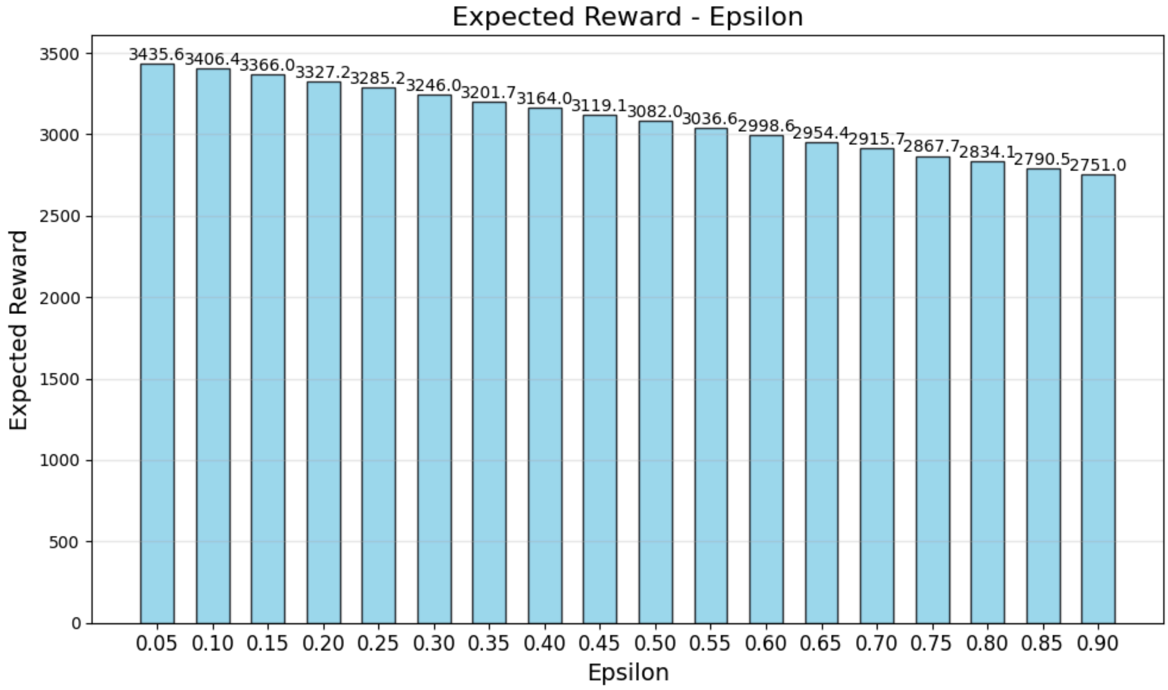
1.1(1) set different epsilon constant

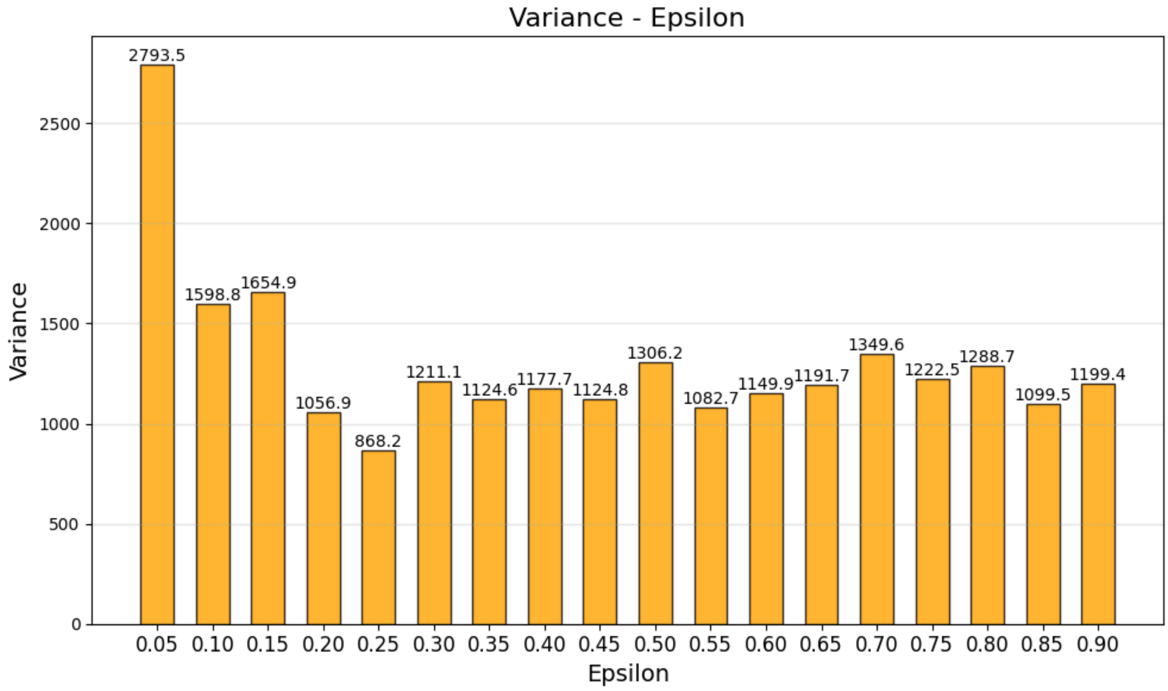
Before we set ϵ = [0.1,0.5,0.9] , this case we set 50 different values from 0.05 to 0.9(we set values from 0.05 to 0.9 because we can keep it effective)

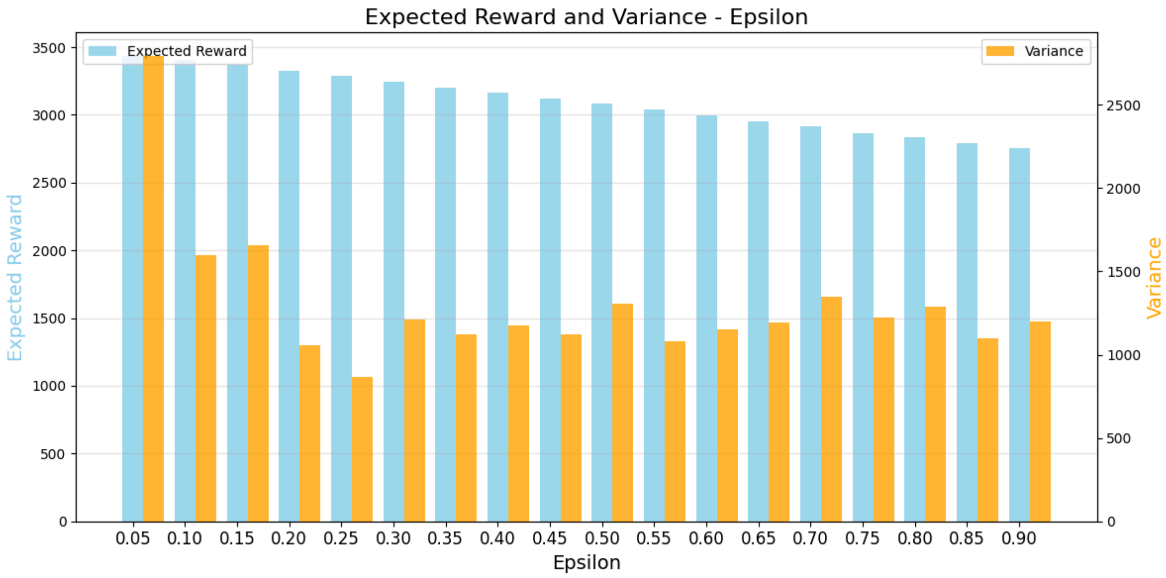
In [7]:
```
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
epsilons = np.arange(0.05, 0.95, 0.05)
results_rewards = {
    'epsilon_greedy': [],
}
results_variances = {
    'epsilon_greedy': [],
}
epsilon_rewards = {epsilon: [] for epsilon in epsilons}

for epsilon in epsilons:
    cumulative_rewards = 0
    trial_rewards_list = []

    for _ in range(num_trials):
        bandit = EpsilonGreedy(epsilon=epsilon)
        rewards = []
        for t in range(N):
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
```

```
            bandit.update(arm, reward)
            cumulative_rewards += reward
            rewards.append(reward)

        trial_rewards_list.append(np.sum(rewards))

    avg_reward = round(cumulative_rewards / num_trials, 1)
    variance = round(np.var(trial_rewards_list), 1)

    results_rewards['epsilon_greedy'].append(avg_reward)
    results_variances['epsilon_greedy'].append(variance)
    epsilon_rewards[epsilon] = rewards

def plot_expectation(means, param_values, param_name):
    plt.figure(figsize=(10, 6))
    bar_positions = np.arange(len(param_values))
    plt.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='

    for i, mean in enumerate(means):
        plt.text(bar_positions[i], mean + 0.01, f"{mean:.1f}", ha='center
    plt.xticks(bar_positions, [f"{val:.2f}" for val in param_values], fon
    plt.xlabel(f"{param_name.capitalize()}", fontsize=14)
    plt.ylabel("Expected Reward", fontsize=14)
    plt.title(f"Expected Reward - {param_name.capitalize()}", fontsize=16
    plt.grid(axis='y', alpha=0.3)
    plt.tight_layout()
    plt.show()

def plot_variance(variances, param_values, param_name):
    plt.figure(figsize=(10, 6))
    bar_positions = np.arange(len(param_values))
    plt.bar(bar_positions, variances, width=0.6, color='orange', edgecolo

    for i, variance in enumerate(variances):
        plt.text(bar_positions[i], variance + 0.01, f"{variance:.1f}", ha
    plt.xticks(bar_positions, [f"{val:.2f}" for val in param_values], fon
    plt.xlabel(f"{param_name.capitalize()}", fontsize=14)
    plt.ylabel("Variance", fontsize=14)
    plt.title(f"Variance - {param_name.capitalize()}", fontsize=16)
    plt.grid(axis='y', alpha=0.3)
    plt.tight_layout()
    plt.show()

def plot_combined(means, variances, param_values, param_name):
    fig, ax1 = plt.subplots(figsize=(12, 6))

    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax1.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue',

    ax1.set_xticks(bar_positions)
    ax1.set_xticklabels([f"{epsilon:.2f}" for epsilon in param_values], f
    ax1.set_xlabel(f"{param_name.capitalize()}", fontsize=14)
    ax1.set_ylabel("Expected Reward", fontsize=14, color='skyblue')
    ax1.set_title("Expected Reward and Variance - Epsilon", fontsize=16)
    ax1.grid(axis='y', alpha=0.3)

    ax2 = ax1.twinx()
```

```
    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=14, color='orange')

    fig.tight_layout()
    ax1.legend(loc='upper left')
    ax2.legend(loc='upper right')

    plt.show()

plot_expectation(results_rewards['epsilon_greedy'], epsilons, "epsilon")

plot_variance(results_variances['epsilon_greedy'], epsilons, "epsilon")

plot_combined(results_rewards['epsilon_greedy'], results_variances['epsil
```







We find that when epsilon tends to 0.1, it has the highest expected score, but the variance is large and the stability is insufficient, which is caused by ignoring exploration.

In addition, we found that as epsilon increases, the expected score decreases, which is due to over-exploration.

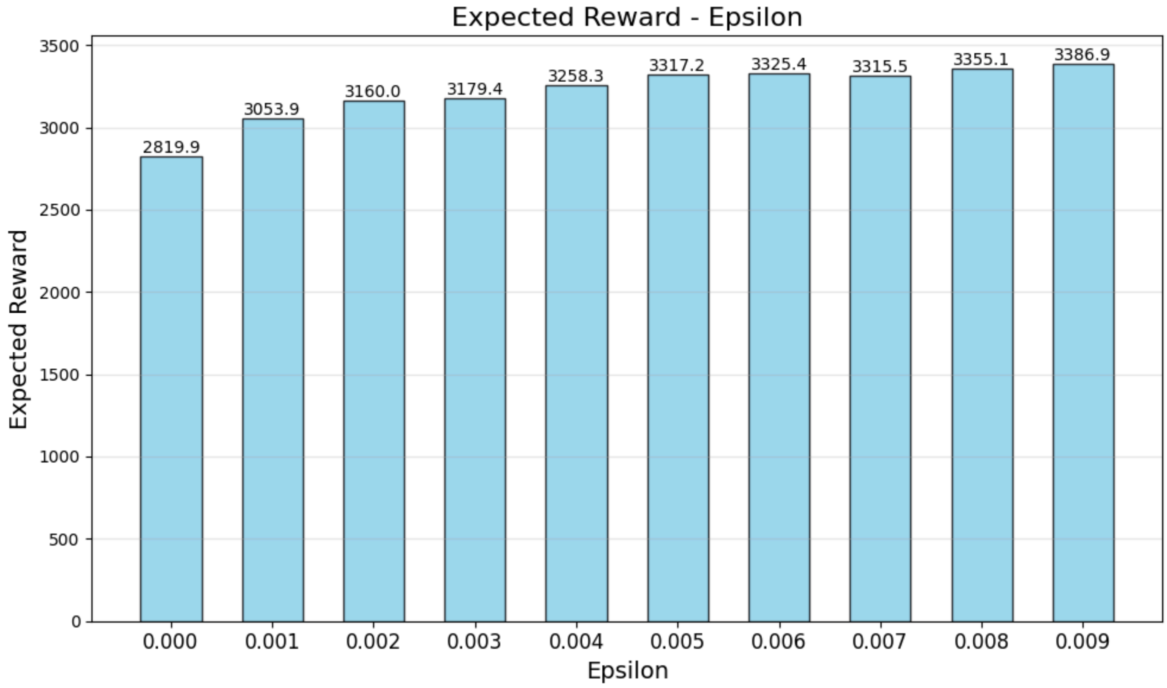
Expansion

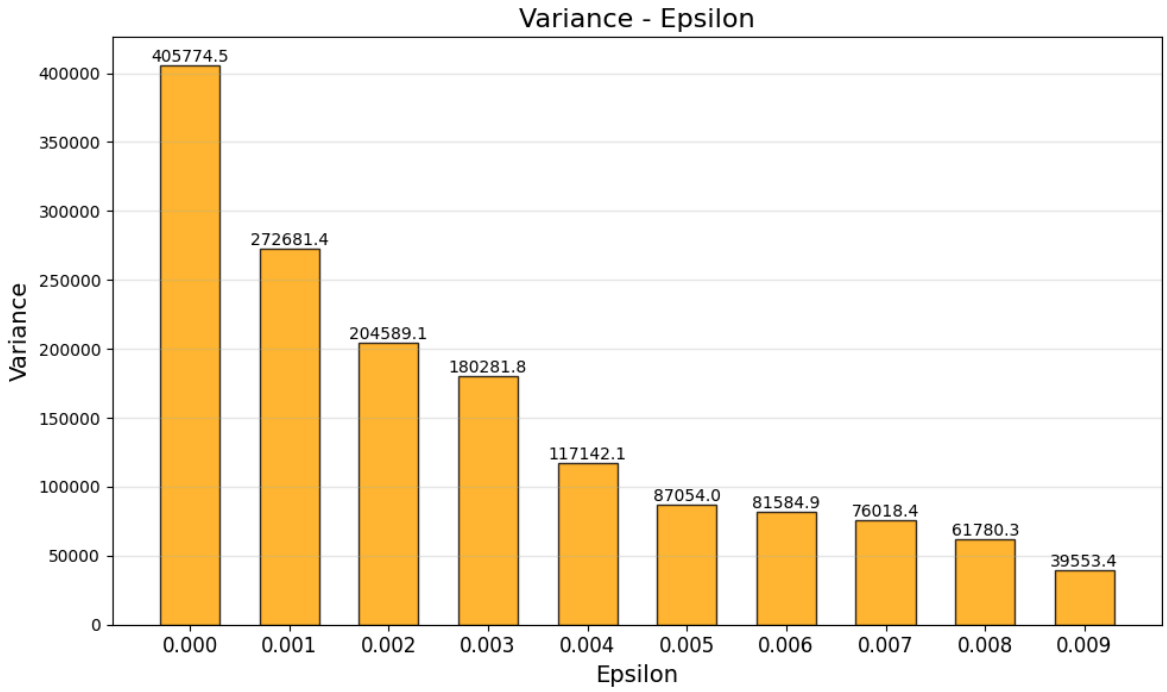
In order to study the situation when ϵ is very small, we set 0.001 to 0.01 with 0.001 as the gap for the experiment

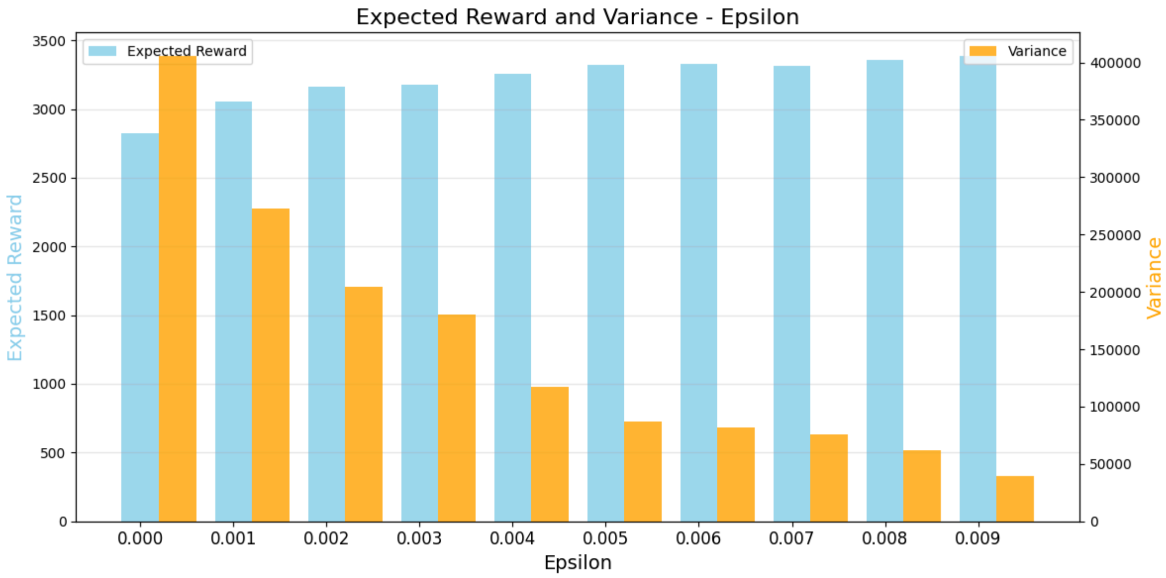
In [8]:
```python
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
epsilons = np.arange(0, 0.01, 0.001)
results_rewards = {
    'epsilon_greedy': [],
}
results_variances = {
    'epsilon_greedy': [],
}
epsilon_rewards = {epsilon: [] for epsilon in epsilons}

for epsilon in epsilons:
    cumulative_rewards = 0
    trial_rewards_list = []

    for _ in range(num_trials):
        bandit = EpsilonGreedy(epsilon=epsilon)
        rewards = []
        for t in range(N):
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
            bandit.update(arm, reward)
            cumulative_rewards += reward
            rewards.append(reward)
```

```
        trial_rewards_list.append(np.sum(rewards))

    avg_reward = round(cumulative_rewards / num_trials, 1)
    variance = round(np.var(trial_rewards_list), 1)

    results_rewards['epsilon_greedy'].append(avg_reward)
    results_variances['epsilon_greedy'].append(variance)
    epsilon_rewards[epsilon] = rewards

def plot_expectation(means, param_values, param_name):
    plt.figure(figsize=(10, 6))
    bar_positions = np.arange(len(param_values))
    plt.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='

    for i, mean in enumerate(means):
        plt.text(bar_positions[i], mean + 0.01, f"{mean:.1f}", ha='center
    plt.xticks(bar_positions, [f"{val:.3f}" for val in param_values], fon
    plt.xlabel(f"{param_name.capitalize()}", fontsize=14)
    plt.ylabel("Expected Reward", fontsize=14)
    plt.title(f"Expected Reward - {param_name.capitalize()}", fontsize=16
    plt.grid(axis='y', alpha=0.3)
    plt.tight_layout()
    plt.show()

def plot_variance(variances, param_values, param_name):
    plt.figure(figsize=(10, 6))
    bar_positions = np.arange(len(param_values))
    plt.bar(bar_positions, variances, width=0.6, color='orange', edgecolo

    for i, variance in enumerate(variances):
        plt.text(bar_positions[i], variance + 0.01, f"{variance:.1f}", ha
    plt.xticks(bar_positions, [f"{val:.3f}" for val in param_values], fon
    plt.xlabel(f"{param_name.capitalize()}", fontsize=14)
    plt.ylabel("Variance", fontsize=14)
    plt.title(f"Variance - {param_name.capitalize()}", fontsize=16)
    plt.grid(axis='y', alpha=0.3)
    plt.tight_layout()
    plt.show()

def plot_combined(means, variances, param_values, param_name):
    fig, ax1 = plt.subplots(figsize=(12, 6))

    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax1.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue',

    ax1.set_xticks(bar_positions)
    ax1.set_xticklabels([f"{epsilon:.3f}" for epsilon in param_values], f
    ax1.set_xlabel(f"{param_name.capitalize()}", fontsize=14)
    ax1.set_ylabel("Expected Reward", fontsize=14, color='skyblue')
    ax1.set_title("Expected Reward and Variance - Epsilon", fontsize=16)
    ax1.grid(axis='y', alpha=0.3)

    ax2 = ax1.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=14, color='orange')
```

```
    fig.tight_layout()
    ax1.legend(loc='upper left')
    ax2.legend(loc='upper right')

    plt.show()

plot_expectation(results_rewards['epsilon_greedy'], epsilons, "epsilon")

plot_variance(results_variances['epsilon_greedy'], epsilons, "epsilon")

plot_combined(results_rewards['epsilon_greedy'], results_variances['epsil
```







We observe that the expected score becomes unstable when ϵ is small. As ϵ approaches zero, the expected score decreases, and the variance increases significantly. Therefore, we cannot assume that minimizing exploration will always lead to better performance. In fact, exploration is crucial when there is insufficient information.

1.1(2) set epsilon as a function of t

In order to make our strategy more in line with the reality, we decided to set epsilon's function of t. Our initial idea was to set epsilon to decrease as t increased. The purpose of this is that as the experiment continues, in order to maximize the benefit, we always choose the arm with the highest probability to pull, which is intuitively in line with our cognition.

note: we must keep ϵ between 0 and 1

(1) linear: $\epsilon(t)$ = 0.9 - $0.00016 * t$ (in this case,we can keep ϵ between 0.1 and 0.9)

(2) exponential: $\epsilon(t)$ = $0.8 * e^{-0.5t} + 0.1$

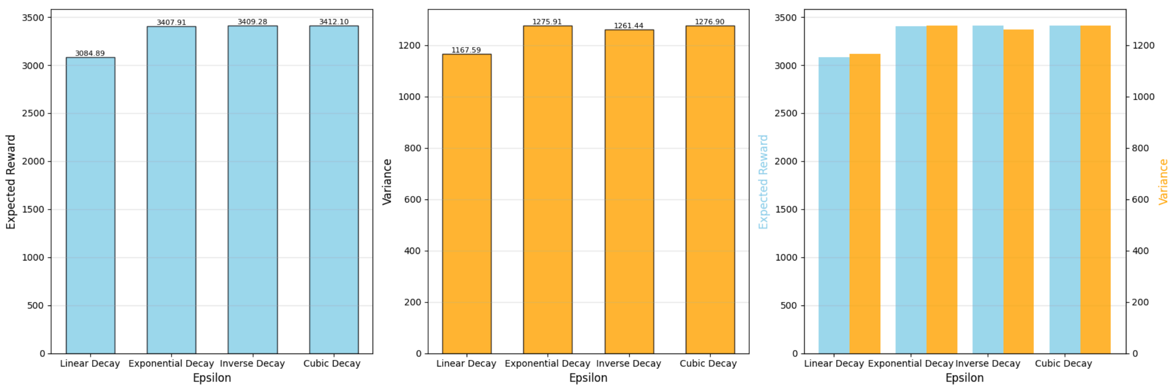
(3) inverse： $\epsilon(t)$ = 0.8* $\frac{1}{t} + 0.1$ (when t = 0, we let ϵ = 0.9)

(4) cubic : $\epsilon(t)$ = $\frac{0.8}{t^3+t^2+t+1}$ + 0.1

In [9]:
```
import numpy as np
import matplotlib.pyplot as plt

def epsilon_linear(t):
    return max(0.1, 0.9 - 0.00016 * t)

def epsilon_exponential(t):
    return 0.8 * np.exp(-0.5*t ) + 0.1

def epsilon_inverse(t):
    return 0.8 / max(1, t) + 0.1
```

```
def epsilon_cubic(t):
    return 0.8 / ((t+1)*(t+1)*(t+1)) + 0.1

N = 5000
num_trials = 200
epsilon_functions = {
    "Linear Decay": epsilon_linear,
    "Exponential Decay": epsilon_exponential,
    "Inverse Decay": epsilon_inverse,
    "Cubic Decay ": epsilon_cubic,
}

results_rewards = []
results_variances = []

for name, epsilon_func in epsilon_functions.items():
    cumulative_rewards = 0
    all_rewards = []

    for _ in range(num_trials):
        bandit = EpsilonGreedy(epsilon=epsilon_func(0))
        rewards = []
        for t in range(N):
            epsilon_t = epsilon_func(t)
            bandit.modify_parameter(epsilon_t)
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
            bandit.update(arm, reward)
            cumulative_rewards += reward
            rewards.append(reward)
        all_rewards.append(np.sum(rewards))

    avg_reward = cumulative_rewards / num_trials
    variance = np.var(all_rewards)
    results_rewards.append(avg_reward)
    results_variances.append(variance)

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)

    ax.set_xticklabels(param_values, fontsize=10)
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor
    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)

    ax.set_xticklabels(param_values, fontsize=10)
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
```

```
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)
    ax.set_xticklabels(param_values, fontsize=10)

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(1, 3, figsize=(18, 6))

plot_expectation(axs[0], results_rewards, list(epsilon_functions.keys()),
plot_variance(axs[1], results_variances, list(epsilon_functions.keys()),
plot_combined(axs[2], results_rewards, results_variances, list(epsilon_fu

plt.tight_layout()
plt.show()
```



Through the image, we find that the cubic function has high expectations, low variance, stability and efficiency, which is in line with our intuition: it combines exploration and application best. Like exponential function and inverse proportional function, they decline too fast and the exploration time is short. The effects of each experiment may be very different from one another, resulting in a large variance, indicating that they are not stable, but their scores also show that their effects are acceptable.Linear functions take too long to explore, so the expectation is minimal.Even if its variance is small, it is not worth adopting.

We keep ϵ at 0.1 to 0.9 by adding constant terms to the exponential and inverse functions

1.2 Further exploration of UCB Algorithm

We know that in the UCB algorithm, we balance the relationship between exploration and exploitation by changing the value of the incoming $I(t)$. However, the value of c may be different. We can measure its influence on our experiment by changing the value of c. Below, I set up an experiment to test the influence of c value on UCB algorithm

1.2(1) Suppose c is different positive constant.

Below, we will change the value of c (but always ensure that it is positive) to measure the effect of different values of c on the final expectation and variance.

(1) $c = 0.1$

(2) $c = 0.5$

(3) $c = 1$

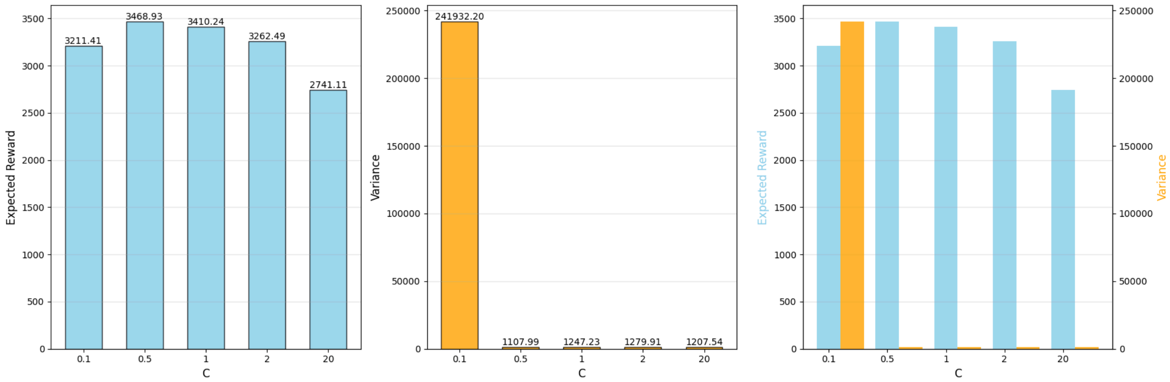
(4) $c = 2$

(5) $c = 20$

In [49]:
```
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
c_values = [0.1, 0.5, 1, 2, 20]
results_rewards = {'ucb': []}
results_variances = {'ucb': []}

for c in c_values:
    cumulative_rewards = 0
    trial_rewards_list = []

    for _ in range(num_trials):
        bandit = UCB(c=c)
        trial_rewards = []

        for t in range(N):
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
            bandit.update(arm, reward)
            trial_rewards.append(reward)
            cumulative_rewards += reward

        trial_rewards_list.append(np.sum(trial_rewards))

    mean_reward = cumulative_rewards / num_trials
    variance_reward = np.var(trial_rewards_list)

    results_rewards['ucb'].append(mean_reward)
    results_variances['ucb'].append(variance_reward)

for i, c in enumerate(c_values):
    print(f"c = {c}   reward: {results_rewards['ucb'][i]:.2f}   variances
```

```
def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"{val}" for val in param_values], fontsize=10)
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"{val}" for val in param_values], fontsize=10)
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    ax.set_xticklabels([f"{val}" for val in param_values], fontsize=10)

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(1, 3, figsize=(18, 6))

plot_expectation(axs[0], results_rewards['ucb'], c_values, "c")
plot_variance(axs[1], results_variances['ucb'], c_values, "c")
plot_combined(axs[2], results_rewards['ucb'], results_variances['ucb'], c

plt.tight_layout()
plt.show()
```

```
c = 0.1   reward: 3211.41   variances: 241932.20
c = 0.5   reward: 3468.93   variances: 1107.99
c = 1   reward: 3410.24   variances: 1247.23
c = 2   reward: 3262.49   variances: 1279.91
c = 20   reward: 2741.11   variances: 1207.54
```



We find that when c=0.1, its expectation is small and its variance is large; Guess the reason is: at this time we have the least interest in exploration, when we get the correct information at the beginning (which arm has the highest probability of scoring), then its expectation will be too large, and if not, it will go further and further down the wrong road, resulting in reduced expectations. This algorithm has no stability. According to our values, it is not difficult to find that the score expectation when c=0.5 is greater than the score expectation when c=1. We preliminarily infer that when c is between 0.1 and 1, there is a better score.

Exploration： when c between 0.1 and 1

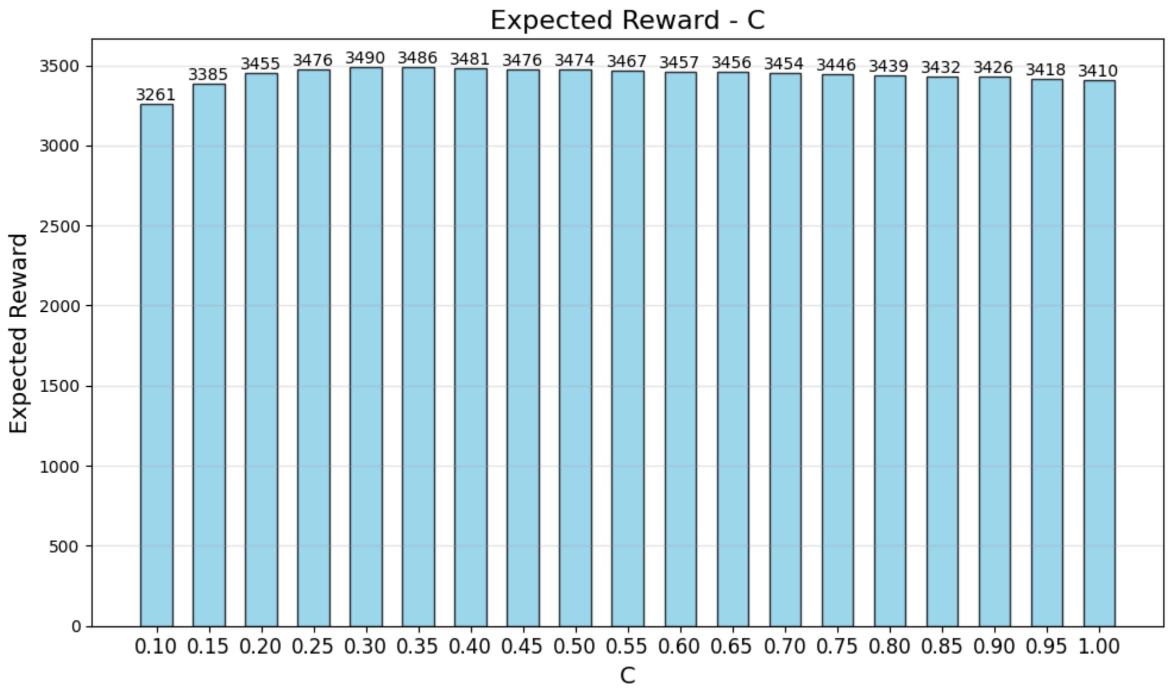
Below I will set the c value with a spacing of 0.05 for our observation.

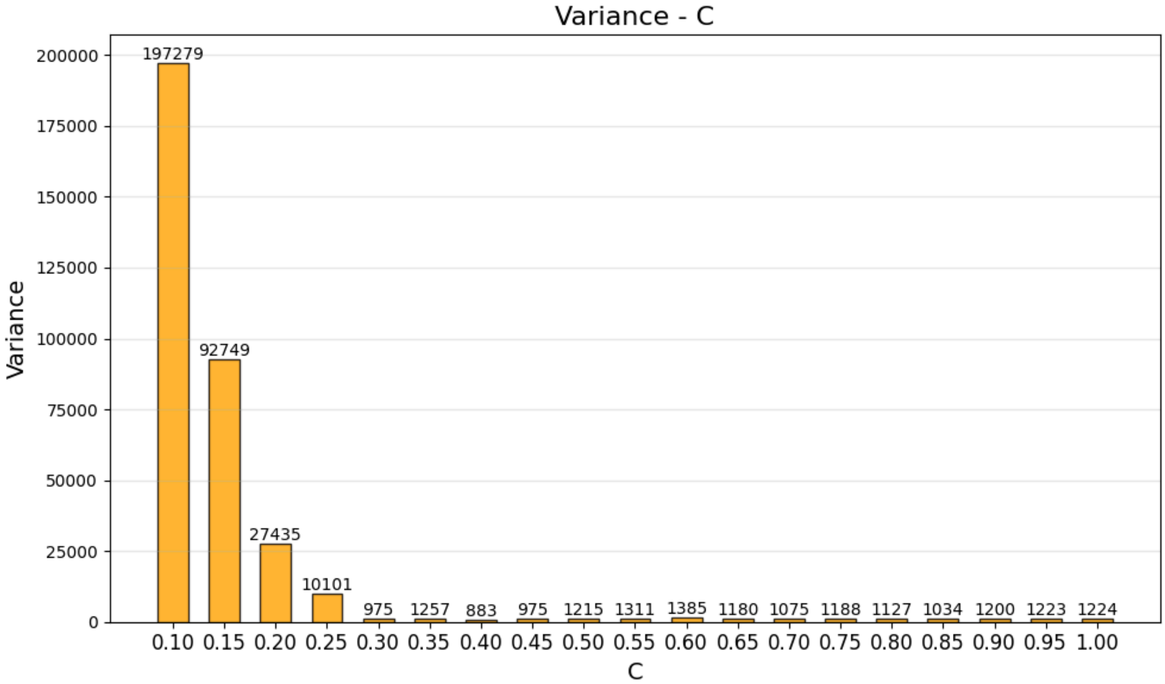
In [47]:
```
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
c_values = np.arange(0.1, 1.05, 0.05)
results_rewards = {'ucb': []}
results_variances = {'ucb': []}

for c in c_values:
    cumulative_rewards = 0
    trial_rewards_list = []

    for _ in range(num_trials):
        bandit = UCB(c=c)
        trial_rewards = []

        for t in range(N):
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
            bandit.update(arm, reward)
            trial_rewards.append(reward)
            cumulative_rewards += reward

        trial_rewards_list.append(np.sum(trial_rewards))

    mean_reward = cumulative_rewards / num_trials
    variance_reward = np.var(trial_rewards_list)

    results_rewards['ucb'].append(mean_reward)
    results_variances['ucb'].append(variance_reward)
```

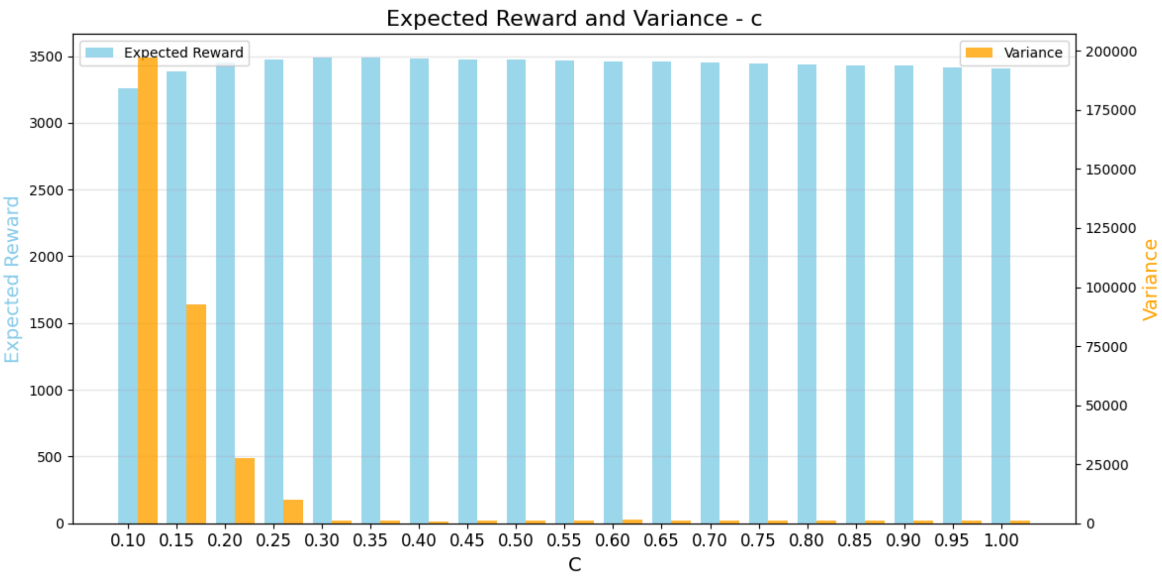
```
for i, c in enumerate(c_values):
    print(f"c = {c:.2f}   reward: {int(results_rewards['ucb'][i])}   vari

def plot_expectation(means, param_values, param_name):
    plt.figure(figsize=(10, 6))
    bar_positions = np.arange(len(param_values))
    plt.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='

    for i, mean in enumerate(means):
        plt.text(bar_positions[i], mean + 0.01, f"{int(mean)}", ha='cente
    plt.xticks(bar_positions, [f"{val:.2f}" for val in param_values], fon
    plt.xlabel(f"{param_name.capitalize()}", fontsize=14)
    plt.ylabel("Expected Reward", fontsize=14)
    plt.title(f"Expected Reward - {param_name.capitalize()}", fontsize=16
    plt.grid(axis='y', alpha=0.3)
    plt.tight_layout()
    plt.show()

def plot_variance(variances, param_values, param_name):
    plt.figure(figsize=(10, 6))
    bar_positions = np.arange(len(param_values))
    plt.bar(bar_positions, variances, width=0.6, color='orange', edgecolo

    for i, variance in enumerate(variances):
        plt.text(bar_positions[i], variance + 0.01, f"{int(variance)}", h
    plt.xticks(bar_positions, [f"{val:.2f}" for val in param_values], fon
    plt.xlabel(f"{param_name.capitalize()}", fontsize=14)
    plt.ylabel("Variance", fontsize=14)
    plt.title(f"Variance - {param_name.capitalize()}", fontsize=16)
    plt.grid(axis='y', alpha=0.3)
    plt.tight_layout()
    plt.show()

def plot_combined(means, variances, param_values, param_name):
    fig, ax1 = plt.subplots(figsize=(12, 6))

    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax1.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue',

    ax1.set_xticks(bar_positions)
    ax1.set_xticklabels([f"{val:.2f}" for val in param_values], fontsize=
    ax1.set_xlabel(f"{param_name.capitalize()}", fontsize=14)
    ax1.set_ylabel("Expected Reward", fontsize=14, color='skyblue')
    ax1.set_title("Expected Reward and Variance - c", fontsize=16)
    ax1.grid(axis='y', alpha=0.3)

    ax2 = ax1.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=14, color='orange')

    fig.tight_layout()
    ax1.legend(loc='upper left')
    ax2.legend(loc='upper right')

    plt.show()
```

```
plot_expectation(results_rewards['ucb'], c_values, "c")

plot_variance(results_variances['ucb'], c_values, "c")

plot_combined(results_rewards['ucb'], results_variances['ucb'], c_values,
```

```
c = 0.10   reward: 3261   variances: 197279
c = 0.15   reward: 3385   variances: 92749
c = 0.20   reward: 3455   variances: 27435
c = 0.25   reward: 3476   variances: 10101
c = 0.30   reward: 3490   variances: 975
c = 0.35   reward: 3486   variances: 1257
c = 0.40   reward: 3481   variances: 883
c = 0.45   reward: 3476   variances: 975
c = 0.50   reward: 3474   variances: 1215
c = 0.55   reward: 3467   variances: 1311
c = 0.60   reward: 3457   variances: 1385
c = 0.65   reward: 3456   variances: 1180
c = 0.70   reward: 3454   variances: 1075
c = 0.75   reward: 3446   variances: 1188
c = 0.80   reward: 3439   variances: 1127
c = 0.85   reward: 3432   variances: 1034
c = 0.90   reward: 3426   variances: 1200
c = 0.95   reward: 3418   variances: 1223
c = 1.00   reward: 3410   variances: 1224
```







We find that when c is 0.25, its expected score is the largest, but the variance is also larger than 0.3, and its algorithm is not stable enough. While when c is equal to 0.3, although the expected score is smaller than the expected score when c is equal to 0.25, the variance is small and the algorithm is stable.We can make a choice based on the actual situation.

1.2(2) Suppose c is a function of t

In the UCB algorithm, the term containing c is positively correlated with our acceptance of exploration (the degree of exploration), that is, the more we support exploration, the larger the value of c will be. In real life, our strategy will change dynamically. For example, we may support exploration in the early stage, but with the progress of the experiment, we acquire a lot of information and have a preliminary understanding and speculation about the experiment, that is, we have a speculation about which arm is more likely to win the prize. At this time, we can reduce our interest in exploration and turn to maximize our interests. Therefore, in the experiment, we should also set c as a function of t.

(1) $c(t)$ = -0.001*t + 5.1

(2) $c(t)$ = $ln(\frac{1}{t+1})$ + 9(because we keep c positive)

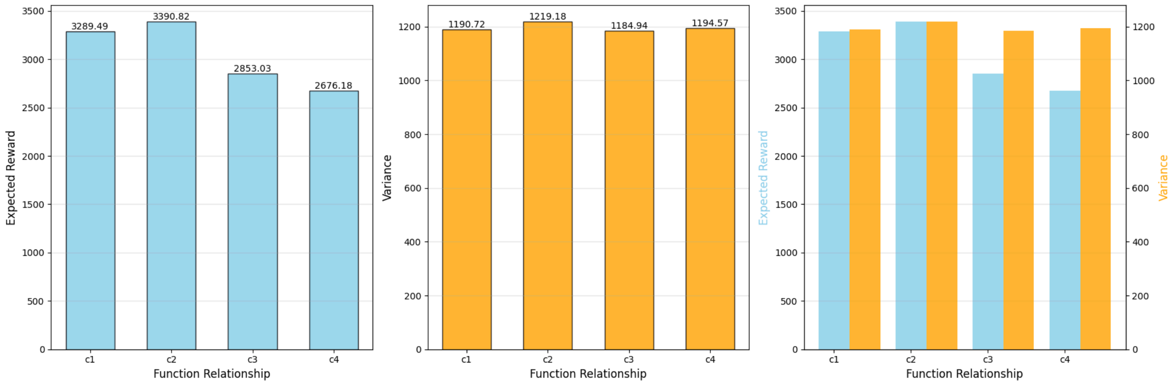
(3) $c(t)$ = $-ln(\frac{1}{t+1})$

(4) $c(t)$ = $e^{0.001*t}$

In [12]:
```
import numpy as np
import matplotlib.pyplot as plt
from math import log, exp

N = 5000
num_trials = 200
t_values = np.arange(1, N + 1)

def c1(t):
    return -0.001 * t + 5.1

def c2(t):
    return np.log(1 / (t + 1)) + 9

def c3(t):
    return -np.log(1 / (t + 1))

def c4(t):
    return np.exp(0.001 * t)

results_rewards = {'c1': [], 'c2': [], 'c3': [], 'c4': []}
results_variances = {'c1': [], 'c2': [], 'c3': [], 'c4': []}

for c_func, c_label in zip([c1, c2, c3, c4], ['c1', 'c2', 'c3', 'c4']):
    cumulative_rewards = 0
    trial_rewards_list = []

    for _ in range(num_trials):
        bandit = UCB(c=c_func(1))
        trial_rewards = []

        for t in t_values:
            c_t = c_func(t)
            bandit.c = c_t
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
            bandit.update(arm, reward)
            trial_rewards.append(reward)
            cumulative_rewards += reward

        trial_rewards_list.append(np.sum(trial_rewards))

    mean_reward = cumulative_rewards / num_trials
    variance_reward = np.var(trial_rewards_list)

    results_rewards[c_label].append(mean_reward)
    results_variances[c_label].append(variance_reward)

for c_label in ['c1', 'c2', 'c3', 'c4']:
```

```
    print(f"{c_label} reward: {results_rewards[c_label][0]:.2f}   varianc

def plot_expectation(ax, means, param_labels):
    bar_positions = np.arange(len(param_labels))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)
    ax.set_xticklabels(param_labels, fontsize=10)
    ax.set_xlabel("Function Relationship", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_labels):
    bar_positions = np.arange(len(param_labels))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)
    ax.set_xticklabels(param_labels, fontsize=10)
    ax.set_xlabel("Function Relationship", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_labels):
    bar_positions = np.arange(len(param_labels))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)
    ax.set_xticklabels(param_labels, fontsize=10)

    ax.set_xlabel("Function Relationship", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(1, 3, figsize=(18, 6))

plot_expectation(axs[0], [results_rewards[c][0] for c in ['c1', 'c2', 'c3
plot_variance(axs[1], [results_variances[c][0] for c in ['c1', 'c2', 'c3'
plot_combined(axs[2], [results_rewards[c][0] for c in ['c1', 'c2', 'c3',

plt.tight_layout()
plt.show()
```

```
c1 reward: 3289.49   variances: 1190.72
c2 reward: 3390.82   variances: 1219.18
c3 reward: 2853.03   variances: 1184.94
c4 reward: 2676.18   variances: 1194.57
```



From the expectation graph, we can see that there is a strong advantage in keeping the spirit of exploration at the beginning and choosing the arm with the higher score at the end. As you can see from the variance graph, they have similar stability

Introduce confidence levels and confidence intervals

With Hoeffding Boun,we can get:

$$reward \sim Bern(\hat{\theta})$$

u:the mean of the distribution of reward,
$\bar{u}$:$\frac{1}{n}\sum_n reward$,
n:count(I(t))

max{$(b-a)$} = 1

$$Pr(|u-\bar{u}| > \epsilon) = 2\delta <= 2e^{-\frac{2n\epsilon^2}{(b-a)^2}} <= 2e^{-\frac{n\epsilon^2}{2}}$$

let $\delta = e^{-\frac{n\epsilon^2}{2}}$,we can get:

$$\epsilon = \sqrt{\frac{2}{n} * log(\frac{1}{\delta})}$$

let $\frac{1}{\delta} = t$,we will get $\epsilon = \sqrt{\frac{2}{count} * log(t)}$

As we mentioned earlier, c can control how much we explore

And that explains, why is there an exploration and development part of this formula:

$$I(t) = argmax(\hat{\theta} + c * \sqrt{\frac{2log(t)}{count(j)}})$$

Confidence invariant about control

$$\epsilon = \sqrt{\frac{2}{n} * log(t)}$$

so the supremum of δ is $e^{-2nc^2\frac{2log(t)}{n}}$

If we want to control that δ is constant, we want to control that $c^2 * log(t)$ is constant, so we need to design a function of c with respect to t.

Assuming we make it equal to the constant w, then $w = c^2 * log(t)$,
$c = \sqrt{log(t) * w}$,when t < 4,we can make the parameters the same as the traditional UCB algorithm.
c should be positive.

1.3 Further exploration of TS Algorithm

The discussion of the TS algorithm is more complicated than the previous two algorithms, we will consider the absolute and relative sizes of the two parameters, and then assume that they have a functional relationship with respect to t. In our opinion, these two parameters represent our understanding of the prior, which provides us with the prior information. Although this prior information is a guess, it is conducive to achieving a larger score in some cases. It is this prior information that makes the TS algorithm score the highest, and the prior information plays a positive role at this time. Below I'll explore whether this prior information can lead us to make bad choices and score fewer points.

By observing the TS algorithm, we can find that the larger the initial value of α, means that we have obtained prior information: the more times we have successfully pulled this arm, and the larger the initial value of β, means that we have obtained prior information: the more times we have failed to pull this arm. Therefore, it is not difficult to see that a higher ratio of $\alpha$to βmeans that we have prior information: the higher the prior probability of success of pulling the arm. So it makes sense to study relative size.

We can change our prior information, for example, maybe the first arm knows more priors and the other arm knows less priors, detecting the impact of this situation. Secondly, we also need to examine the influence of the ratio of α and β on the experiment, and also consider the influence of their absolute and relative sizes.

In order to make the prior information more reasonable, we try to ensure that their ratios are close to the probability of the original question.

Study the effect of absolute size

(1) [(α_1 = 700, β_1 = 300), (α_2 = 500, β_2 = 500), (α_3 = 400, β_3 = 600)]

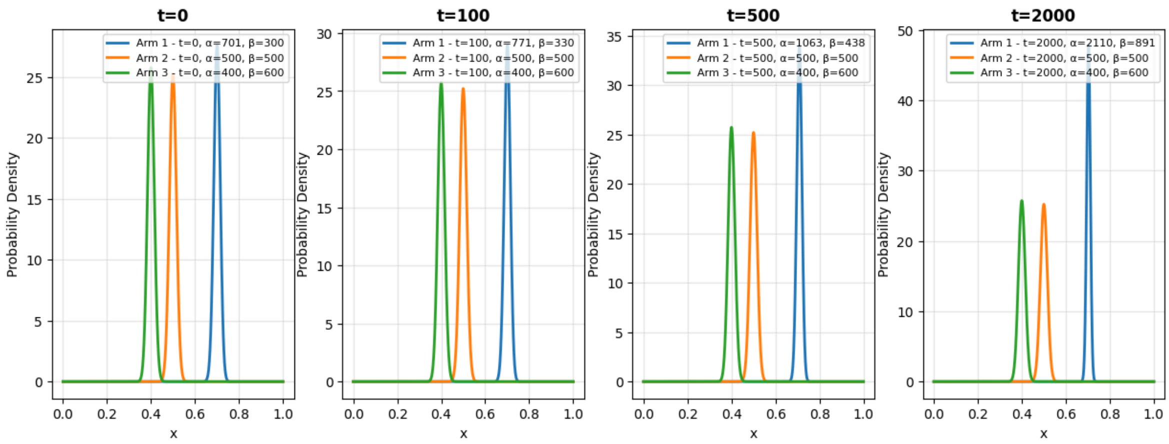
(2) [(α_1 = 7, β_1 = 3), (α_2 = 5, β_2 = 5), (α_3 = 4, β_3 = 6)]

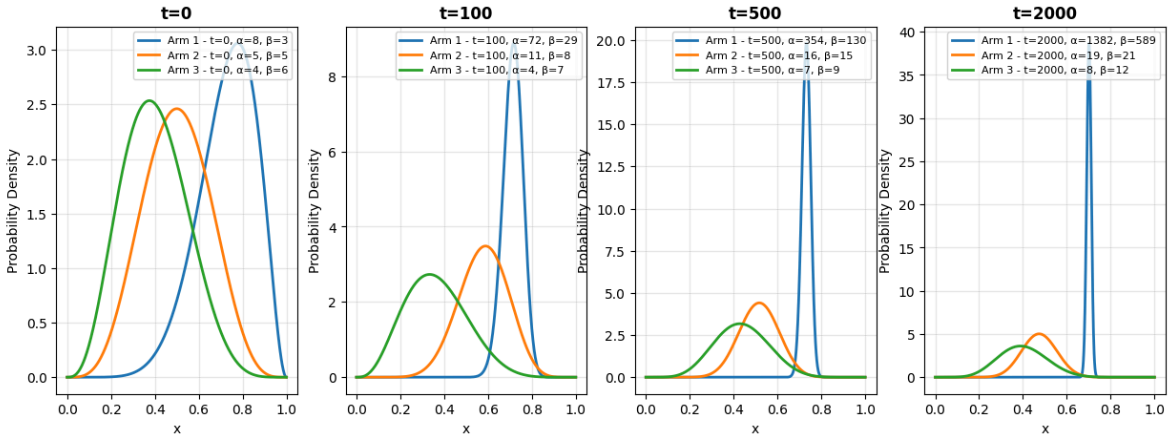
In [43]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

N = 5000
num_trials = 200
experiments = {
```

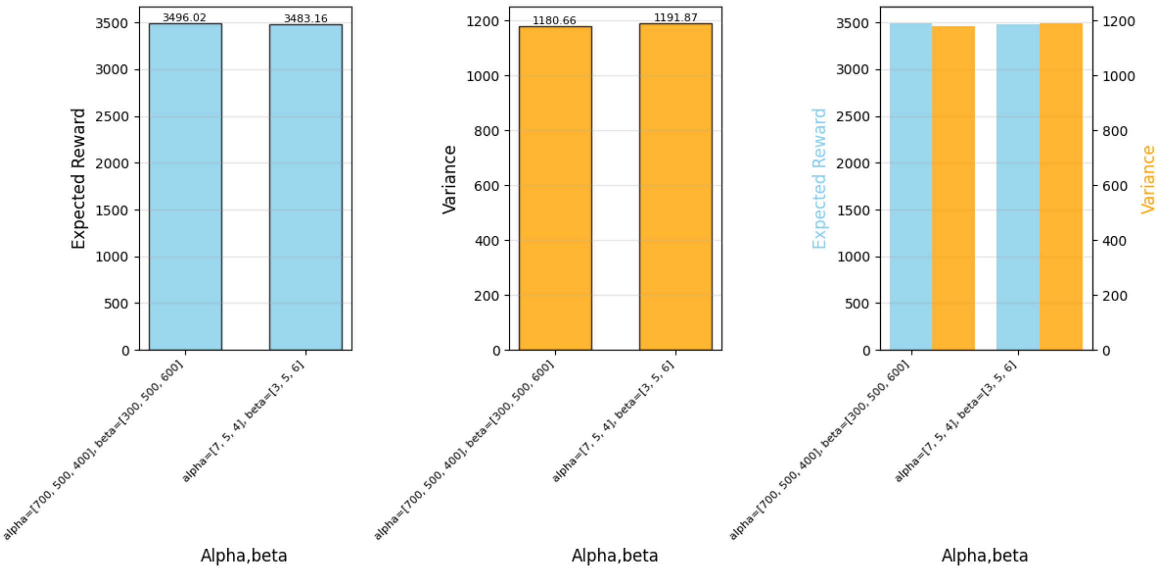
```
    'ts': [
        ([700, 500, 400], [300, 500, 600]),
        ([7, 5, 4], [3, 5, 6])
    ]
}
results_rewards = {
    'ts': []
}
results_variances = {
    'ts': []
}

for value in experiments['ts']:
    trial_rewards_list = []
    cumulative_rewards = 0

    for trial_idx in range(num_trials):
        bandit = ThompsonSampling(alpha=value[0], beta=value[1])

        trial_rewards = []
        for t in range(N):
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
            bandit.update(arm, reward)
            trial_rewards.append(reward)
            cumulative_rewards += reward

            if trial_idx == num_trials - 1 and t in [0, 100, 500, 2000]:

                if trial_idx == num_trials - 1 and t == 0:
                    num_subplots = 4
                    fig, axs = plt.subplots(1, num_subplots, figsize=(15,

                x = np.linspace(0, 1, 1000)

                t_values = [0, 100, 500, 2000]

                t_to_index = {0: 0, 100: 1, 500: 2, 2000: 3}

                for i, (alpha_param, beta_param) in enumerate(zip(bandit.
                    pdf = beta.pdf(x, alpha_param, beta_param)
                    axs_index = t_to_index[t]
                    axs[axs_index].plot(x, pdf, label=f"Arm {i+1} - t={t}
                            linestyle='-' if (t in [0, 100, 500, 2000]) e

                axs[axs_index].set_title(f"t={t}", fontsize=12, weight='b
                axs[axs_index].set_xlabel('x', fontsize=10)
                axs[axs_index].set_ylabel('Probability Density', fontsize
                axs[axs_index].grid(True, alpha=0.3)
                axs[axs_index].legend(loc='upper right', fontsize=8)

        trial_rewards_list.append(np.sum(trial_rewards))

    mean_reward = cumulative_rewards / num_trials
    variance_reward = np.var(trial_rewards_list)

    results_rewards['ts'].append(mean_reward)
    results_variances['ts'].append(variance_reward)

for idx, value in enumerate(experiments['ts']):
```

```
    print(f"alpha = {value[0]} beta = {value[1]} reward: {results_rewards

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(1, 3, figsize=(12, 6))

plot_expectation(axs[0], results_rewards['ts'], experiments['ts'], "alpha
plot_variance(axs[1], results_variances['ts'], experiments['ts'], "alpha,
plot_combined(axs[2], results_rewards['ts'], results_variances['ts'], exp

plt.tight_layout()
plt.show()
```

```
alpha = [700, 500, 400] beta = [300, 500, 600] reward: 3496.02 variances:
1180.66
alpha = [7, 5, 4] beta = [3, 5, 6] reward: 3483.16 variances: 1191.87
```







Compare the situation of (1) and (2) to observe the influence of absolute size on our expected score and stability.

Through experiments, we find that increasing the absolute size can make the algorithm more stable and help us get a better score while keeping the ratio unchanged and assuming that we know the prior information is correct.

What if there's something wrong with our prior information

(3) $[(\alpha_1 = 400, \beta_1 = 600), (\alpha_2 = 700, \beta_2 = 300), (\alpha_3 = 500, \beta_3 = 500)]$

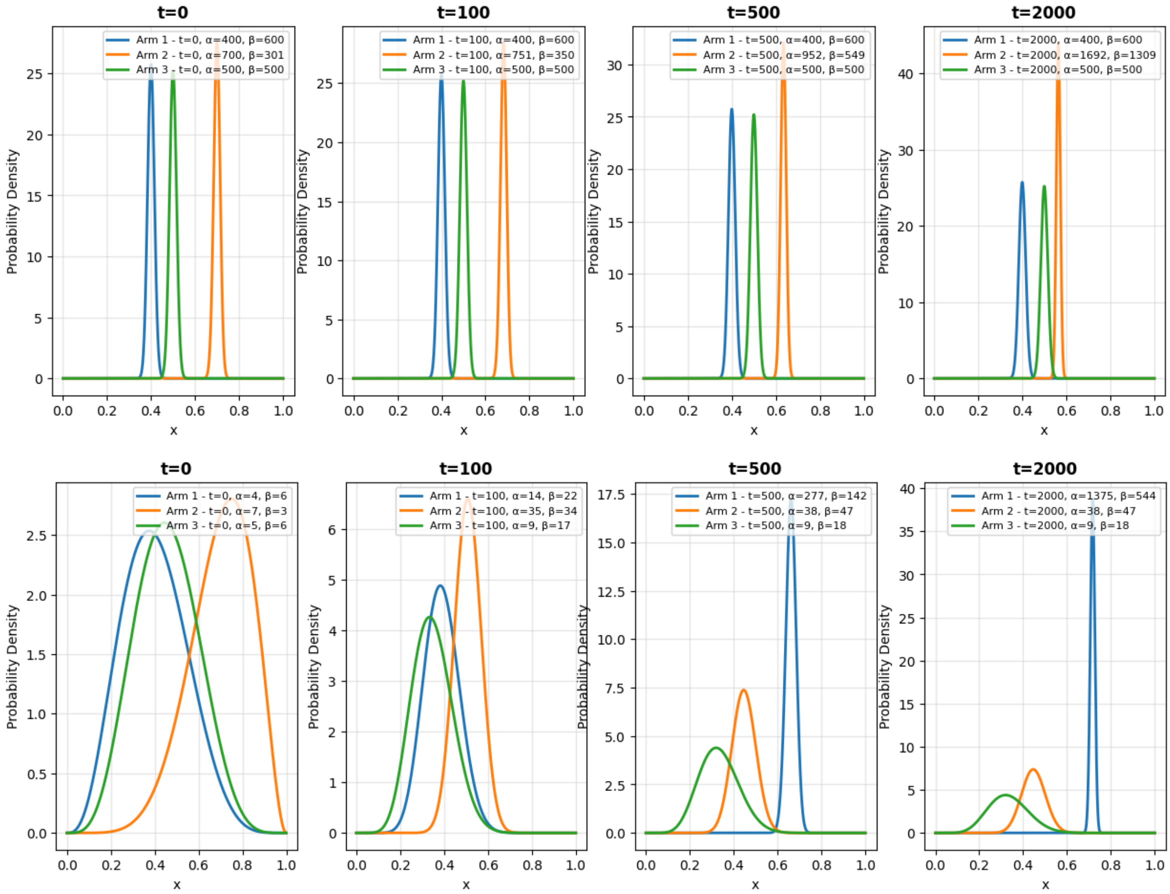
(4) $[(\alpha_1 = 4, \beta_1 = 6), (\alpha_2 = 7, \beta_2 = 3), (\alpha_3 = 5, \beta_3 = 5)]$

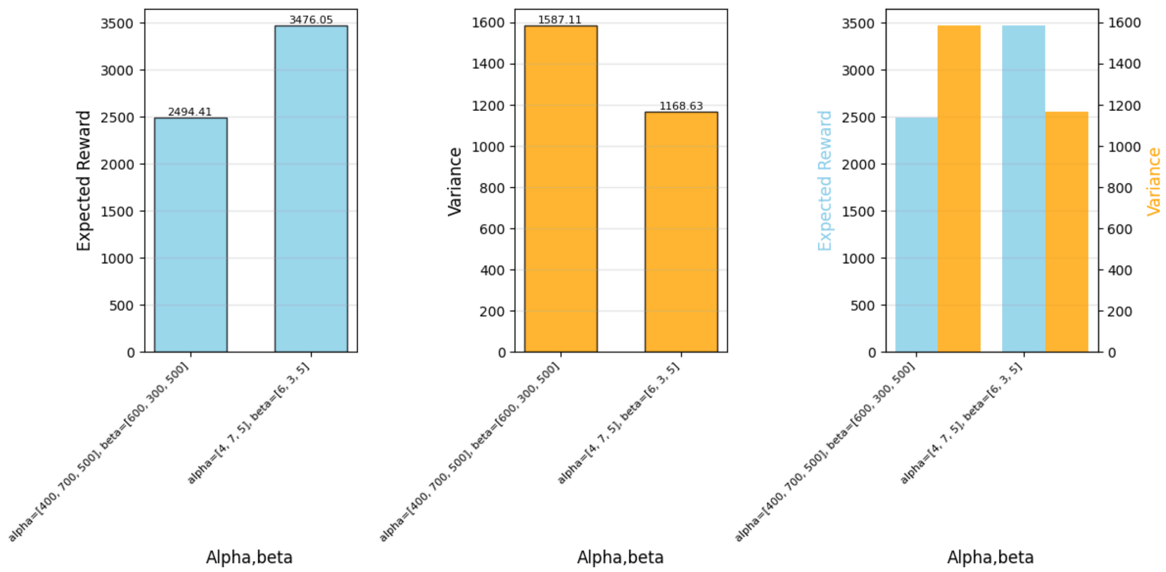
In [44]:
```python
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
experiments = {
    'ts': [
        ([400, 700, 500], [600, 300,500]),
        ([4, 7, 5], [6, 3, 5])
    ]
}
results_rewards = {
    'ts': []
}
results_variances = {
    'ts': []
}

for value in experiments['ts']:
    trial_rewards_list = []
    cumulative_rewards = 0

    for trial_idx in range(num_trials):
        bandit = ThompsonSampling(alpha=value[0], beta=value[1])

        trial_rewards = []
        for t in range(N):
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
            bandit.update(arm, reward)
            trial_rewards.append(reward)
            cumulative_rewards += reward

            if trial_idx == num_trials - 1 and t in [0, 100, 500, 2000]:

                if trial_idx == num_trials - 1 and t == 0:

                    num_subplots = 4
                    fig, axs = plt.subplots(1, num_subplots, figsize=(15,

                x = np.linspace(0, 1, 1000)

                t_values = [0, 100, 500, 2000]

                t_to_index = {0: 0, 100: 1, 500: 2, 2000: 3}

                for i, (alpha_param, beta_param) in enumerate(zip(bandit.
                    pdf = beta.pdf(x, alpha_param, beta_param)
                    axs_index = t_to_index[t]
                    axs[axs_index].plot(x, pdf, label=f"Arm {i+1} - t={t}
                            linestyle='-' if (t in [0, 100, 500, 2000]) e

                axs[axs_index].set_title(f"t={t}", fontsize=12, weight='b
                axs[axs_index].set_xlabel('x', fontsize=10)
                axs[axs_index].set_ylabel('Probability Density', fontsize
                axs[axs_index].grid(True, alpha=0.3)
                axs[axs_index].legend(loc='upper right', fontsize=8)

        trial_rewards_list.append(np.sum(trial_rewards))
```

```
    mean_reward = cumulative_rewards / num_trials
    variance_reward = np.var(trial_rewards_list)

    results_rewards['ts'].append(mean_reward)
    results_variances['ts'].append(variance_reward)

for idx, value in enumerate(experiments['ts']):
    print(f"alpha = {value[0]} beta = {value[1]} reward: {results_rewards

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(1, 3, figsize=(12, 6))

plot_expectation(axs[0], results_rewards['ts'], experiments['ts'], "alpha
plot_variance(axs[1], results_variances['ts'], experiments['ts'], "alpha,
plot_combined(axs[2], results_rewards['ts'], results_variances['ts'], exp
```

```
plt.tight_layout()
plt.show()
```

```
alpha = [400, 700, 500] beta = [600, 300, 500] reward: 2494.41 variances:
1587.11
alpha = [4, 7, 5] beta = [6, 3, 5] reward: 3476.05 variances: 1168.63
```





We find that when we increase the absolute size of the error prior information, then we greatly reduce the score expectation and decrease the stability. When the absolute size of the error information is small, it has less impact on the score expectation of the experiment. The reason is speculated: we have conducted multiple pull arms, and the error information accounts for a relatively small proportion in the total information.

Suppose we know only some prior information of the arm, or the other prior information of the arm is small relative to the relative size of the prior information of the arm

(5) $[(\alpha_1 = 700, \beta_1 = 300), (\alpha_2 = 5, \beta_2 = 5), (\alpha_3 = 4, \beta_3 = 6)]$

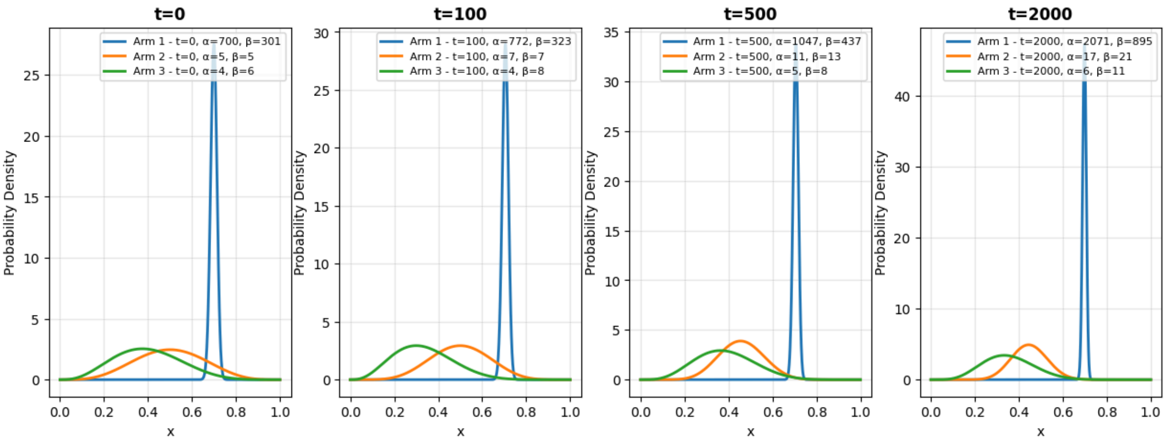
(6) $[(\alpha_1 = 7, \beta_1 = 3), (\alpha_2 = 500, \beta_2 = 500), (\alpha_3 = 400, \beta_3 = 600)]$

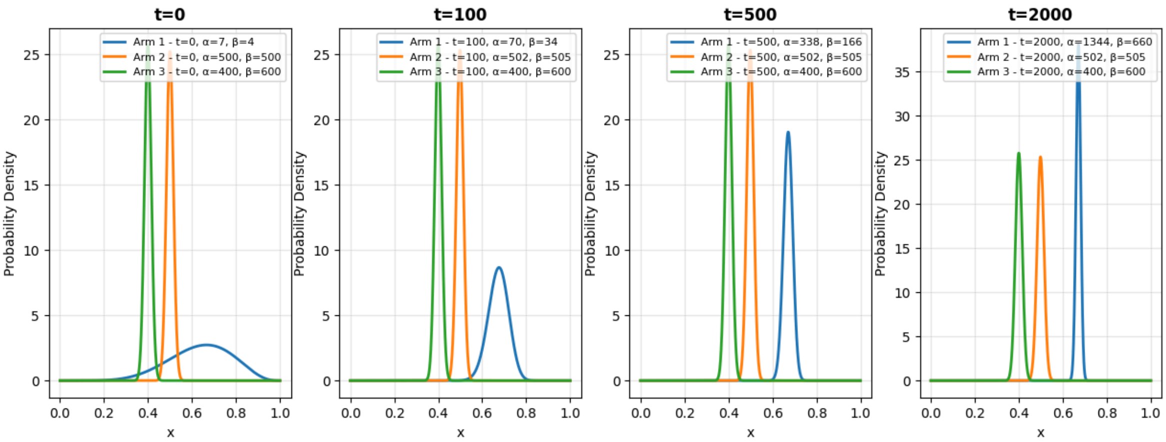
In [45]:
```
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
experiments = {
    'ts': [
        ([700, 5, 4], [300, 5,6]),
        ([7, 500, 400], [3, 500, 600])
    ]
}
results_rewards = {
    'ts': []
}
results_variances = {
    'ts': []
}

for value in experiments['ts']:
    trial_rewards_list = []
    cumulative_rewards = 0

    for trial_idx in range(num_trials):
        bandit = ThompsonSampling(alpha=value[0], beta=value[1])

        trial_rewards = []
        for t in range(N):
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
            bandit.update(arm, reward)
            trial_rewards.append(reward)
            cumulative_rewards += reward

            if trial_idx == num_trials - 1 and t in [0, 100, 500, 2000]:

                if trial_idx == num_trials - 1 and t == 0:

                    num_subplots = 4
                    fig, axs = plt.subplots(1, num_subplots, figsize=(15,

                x = np.linspace(0, 1, 1000)

                t_values = [0, 100, 500, 2000]

                t_to_index = {0: 0, 100: 1, 500: 2, 2000: 3}

                for i, (alpha_param, beta_param) in enumerate(zip(bandit.
                    pdf = beta.pdf(x, alpha_param, beta_param)
                    axs_index = t_to_index[t]
                    axs[axs_index].plot(x, pdf, label=f"Arm {i+1} - t={t}
                            linestyle='-' if (t in [0, 100, 500, 2000]) e
```

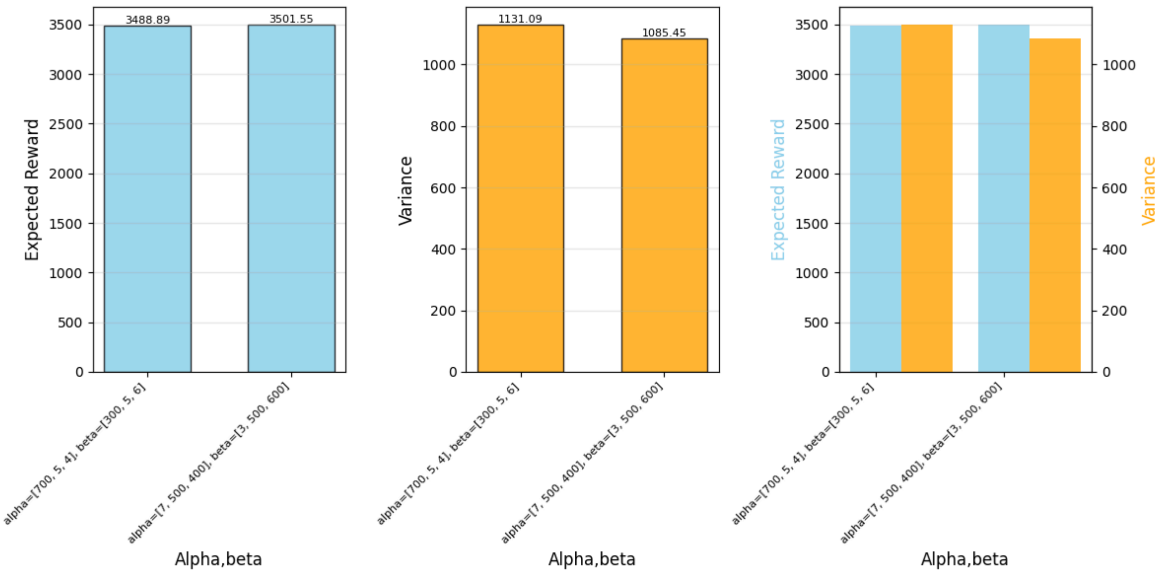
```
                axs[axs_index].set_title(f"t={t}", fontsize=12, weight='b
                axs[axs_index].set_xlabel('x', fontsize=10)
                axs[axs_index].set_ylabel('Probability Density', fontsize
                axs[axs_index].grid(True, alpha=0.3)
                axs[axs_index].legend(loc='upper right', fontsize=8)

        trial_rewards_list.append(np.sum(trial_rewards))

    mean_reward = cumulative_rewards / num_trials
    variance_reward = np.var(trial_rewards_list)

    results_rewards['ts'].append(mean_reward)
    results_variances['ts'].append(variance_reward)

for idx, value in enumerate(experiments['ts']):
    print(f"alpha = {value[0]} beta = {value[1]} reward: {results_rewards

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')
```

```python
fig, axs = plt.subplots(1, 3, figsize=(12, 6))

plot_expectation(axs[0], results_rewards['ts'], experiments['ts'], "alpha
plot_variance(axs[1], results_variances['ts'], experiments['ts'], "alpha,
plot_combined(axs[2], results_rewards['ts'], results_variances['ts'], exp

plt.tight_layout()
plt.show()
```

```
alpha = [700, 5, 4] beta = [300, 5, 6] reward: 3488.89 variances: 1131.09
alpha = [7, 500, 400] beta = [3, 500, 600] reward: 3501.55 variances: 108
5.45
```







We find that when we know the correct a priori more, we are more likely to make the correct choice.

What if there's something wrong with our prior information

(7) [(α_1 = 300, β_1 = 700), (α_2 = 5, β_2 = 5), (α_3 = 6, β_3 = 4)]

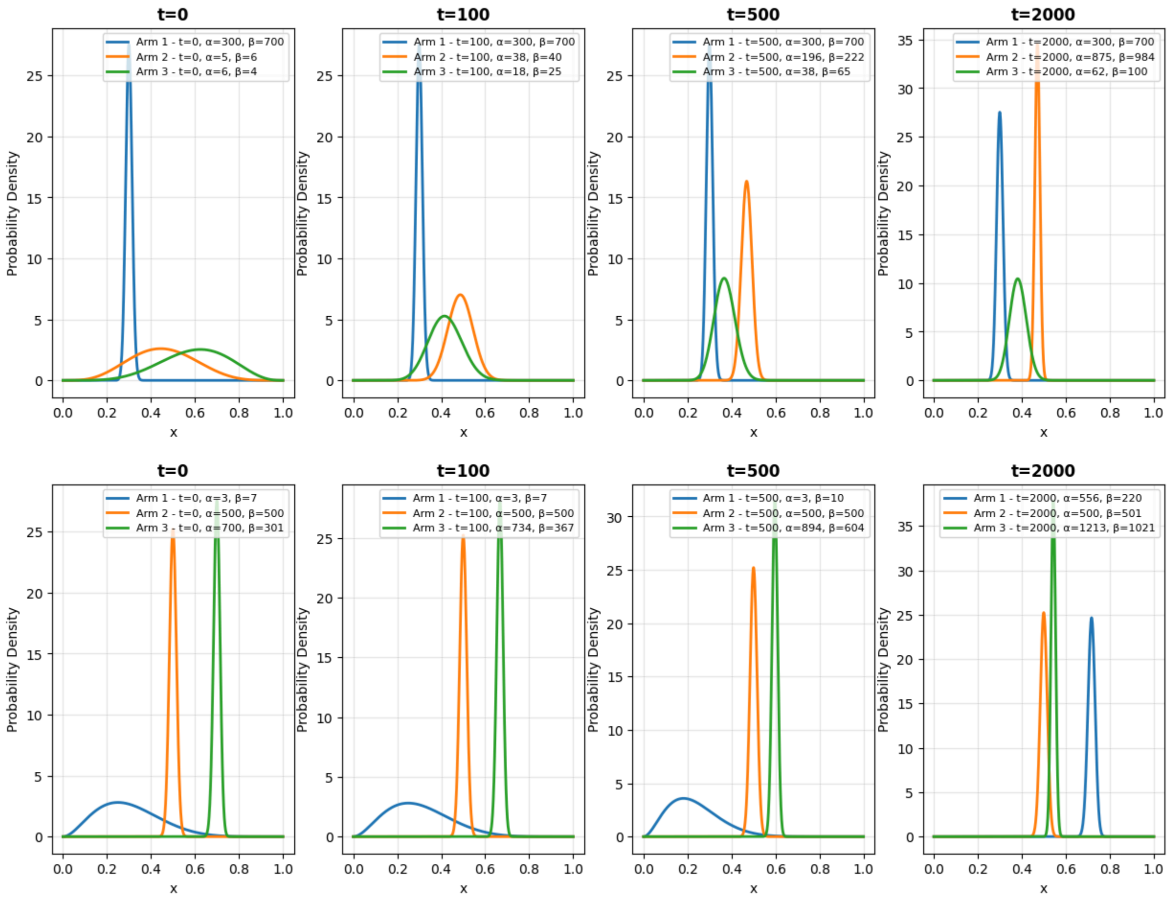
(8) [(α_1 = 3, β_1 = 7), (α_2 = 500, β_2 = 500), (α_3 = 600, β_3 = 300)]

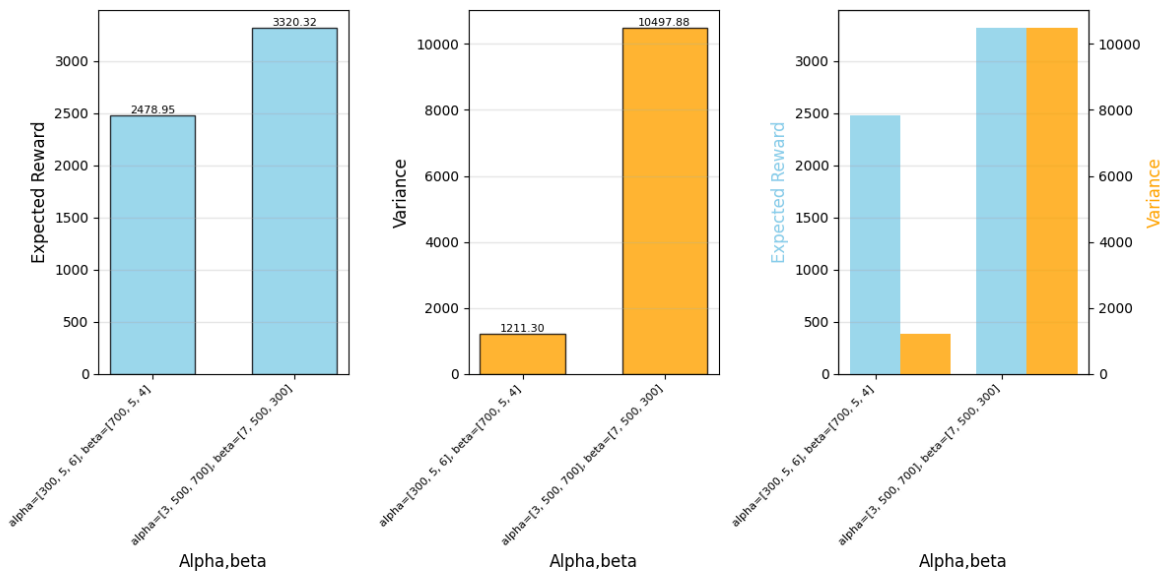
In [46]:
```
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
experiments = {
    'ts': [
        ([300, 5, 6], [700, 5,4]),
        ([3, 500, 700], [7, 500, 300])
    ]
}
results_rewards = {
    'ts': []
}
results_variances = {
    'ts': []
}

for value in experiments['ts']:
    trial_rewards_list = []
    cumulative_rewards = 0

    for trial_idx in range(num_trials):
        bandit = ThompsonSampling(alpha=value[0], beta=value[1])

        trial_rewards = []
        for t in range(N):
            arm = bandit.select_arm()
            reward = bandit.pull(arm)
            bandit.update(arm, reward)
            trial_rewards.append(reward)
            cumulative_rewards += reward

            if trial_idx == num_trials - 1 and t in [0, 100, 500, 2000]:

                if trial_idx == num_trials - 1 and t == 0:

                    num_subplots = 4
                    fig, axs = plt.subplots(1, num_subplots, figsize=(15,

                x = np.linspace(0, 1, 1000)

                t_values = [0, 100, 500, 2000]

                t_to_index = {0: 0, 100: 1, 500: 2, 2000: 3}

                for i, (alpha_param, beta_param) in enumerate(zip(bandit.
                    pdf = beta.pdf(x, alpha_param, beta_param)
                    axs_index = t_to_index[t]
```

```
                    axs[axs_index].plot(x, pdf, label=f"Arm {i+1} - t={t}
                            linestyle='-' if (t in [0, 100, 500, 2000]) e

                axs[axs_index].set_title(f"t={t}", fontsize=12, weight='b
                axs[axs_index].set_xlabel('x', fontsize=10)
                axs[axs_index].set_ylabel('Probability Density', fontsize
                axs[axs_index].grid(True, alpha=0.3)
                axs[axs_index].legend(loc='upper right', fontsize=8)

        trial_rewards_list.append(np.sum(trial_rewards))

    mean_reward = cumulative_rewards / num_trials
    variance_reward = np.var(trial_rewards_list)

    results_rewards['ts'].append(mean_reward)
    results_variances['ts'].append(variance_reward)

for idx, value in enumerate(experiments['ts']):
    print(f"alpha = {value[0]} beta = {value[1]} reward: {results_rewards

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)
    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in param
    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c
```

```
    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(1, 3, figsize=(12, 6))

plot_expectation(axs[0], results_rewards['ts'], experiments['ts'], "alpha
plot_variance(axs[1], results_variances['ts'], experiments['ts'], "alpha,
plot_combined(axs[2], results_rewards['ts'], results_variances['ts'], exp

plt.tight_layout()
plt.show()
```

```
alpha = [300, 5, 6] beta = [700, 5, 4] reward: 2478.95 variances: 1211.30
alpha = [3, 500, 700] beta = [7, 500, 300] reward: 3320.32 variances: 1049
7.88
```





We found that when we know that the error information of the arm with the highest probability of winning is relatively large, it will cause us to misestimate its probability of winning, which will lead to a significant reduction in our final score, and this phenomenon is not accidental. However, when we know the wrong priori of other arms, the expectation change is relatively small. The reason is that we can recover the guess of the probability of the arm with the highest probability of winning the prize through several experiments, and then pull it more times in the development and utilization stage, and the expected score is larger.

In general, we find that when applying the TS algorithm, we should make as good a guess as possible, otherwise it will reduce our score expectations. In addition, if we can make a guess about the relative size of the arm with a high probability of winning, we will have a higher score.

Problem 5

The exploration-exploitation trade-off is a central concept in multi-armed bandit (MAB) algorithms, and it arises from the need to balance two competing objectives:

1. **Exploration**: Trying out different options (or "arms") to gather more information about their potential. The goal of exploration is to learn more about which arms provide the highest rewards, even if it means taking actions that may not yield the best reward in the short term.

2. **Exploitation**: Leveraging the current knowledge to maximize the reward based on past experiences. In exploitation, the algorithm favors the arm with the highest observed reward so far, assuming that this arm will continue to perform well.

The trade-off occurs because both exploration and exploitation are necessary but contradictory:

- **Exploration** can lead to suboptimal immediate rewards because you're trying arms that may not perform well, but it helps in gathering data that can guide better decisions in the long run.
- **Exploitation** maximizes immediate rewards by sticking with the arm that seems to be the best based on current knowledge, but it can miss out on potentially better options in the future by not exploring less frequently tried arms.

In the context of bandit algorithms:

- **Epsilon-Greedy**: This algorithm controls exploration and exploitation using a parameter (\epsilon). With probability (\epsilon), the algorithm explores by choosing a random arm (exploration), and with probability (1 - \epsilon), it exploits by selecting the arm with the highest estimated reward (exploitation). As (\epsilon) decreases over time, the algorithm shifts towards more exploitation, relying on the accumulated knowledge.

- **Upper Confidence Bound (UCB)**: UCB dynamically adjusts the exploration-exploitation balance by selecting arms based on both the estimated reward and the uncertainty (or variance) associated with each arm. Arms with high uncertainty are explored more frequently, while arms with high expected rewards (and low uncertainty) are exploited more. The exploration is explicitly controlled by the confidence bound term, which depends on the number of times an arm has been pulled.

- **Thompson Sampling**: This algorithm uses a probabilistic model to balance exploration and exploitation. It samples from the posterior distribution of each arm's expected reward and selects the arm with the highest sampled value. The exploration comes from the inherent randomness in the posterior distributions, which encourages trying arms that are less certain about their expected reward, while exploitation naturally happens when one arm consistently has a high sampled reward.

Each algorithm addresses the exploration-exploitation dilemma differently, but they all strive to find the optimal balance to maximize cumulative rewards over time. The key challenge is to explore enough to uncover the best options, but also to exploit them sufficiently to capitalize on the knowledge gathered.

In addition, we found that we should make a concrete analysis based on specific situations. Although some algorithms (which can change the variance and expected score by adjusting parameters) are not stable enough, they are likely to make us obtain the greatest benefits, that is, after selecting the arm with the highest probability of winning the prize, pull it all the time, which is suitable for situations with very little cost. If the cost is more, we should pursue the maximum profit and higher stability, at this time we should maintain the spirit of exploration, get enough information, in the development and utilization.

Problem 6

This time, the reward distributions of these three arms are dependent, and we have prior information about their dependencies. To leverage this information for improved performance, we need to identify the key differences compared to the independent case.

When the three arms are independent, exploration is inefficient, as pulling arm i only updates its estimated value, offering no information about the other arms. However, when the arms are dependent, pulling arm i can provide information about all arms related to it.

(1) Consider a special case where the oracle values $\theta = [0.6, 0.6, 0.5]$ for the three arms. The theoretically maximized expectation is 3000. Let X_{ij} denote the indicator random variable representing the success of the i-th arm on the j-th pull. Assume

that the correlation between the success of arms 1 and 2 on the same pull is given by:

$$\text{Corr}(X_{1j}, X_{2j}) = 1.$$

We propose three algorithms based on ϵ-greedy, UCB, and Thompson Sampling, respectively. For simplicity, we treat arms 1 and 2 as a single arm, denoted as arm 1-2. Each time we explore either arm 1 or arm 2, we update the estimated success probability for both arms simultaneously.

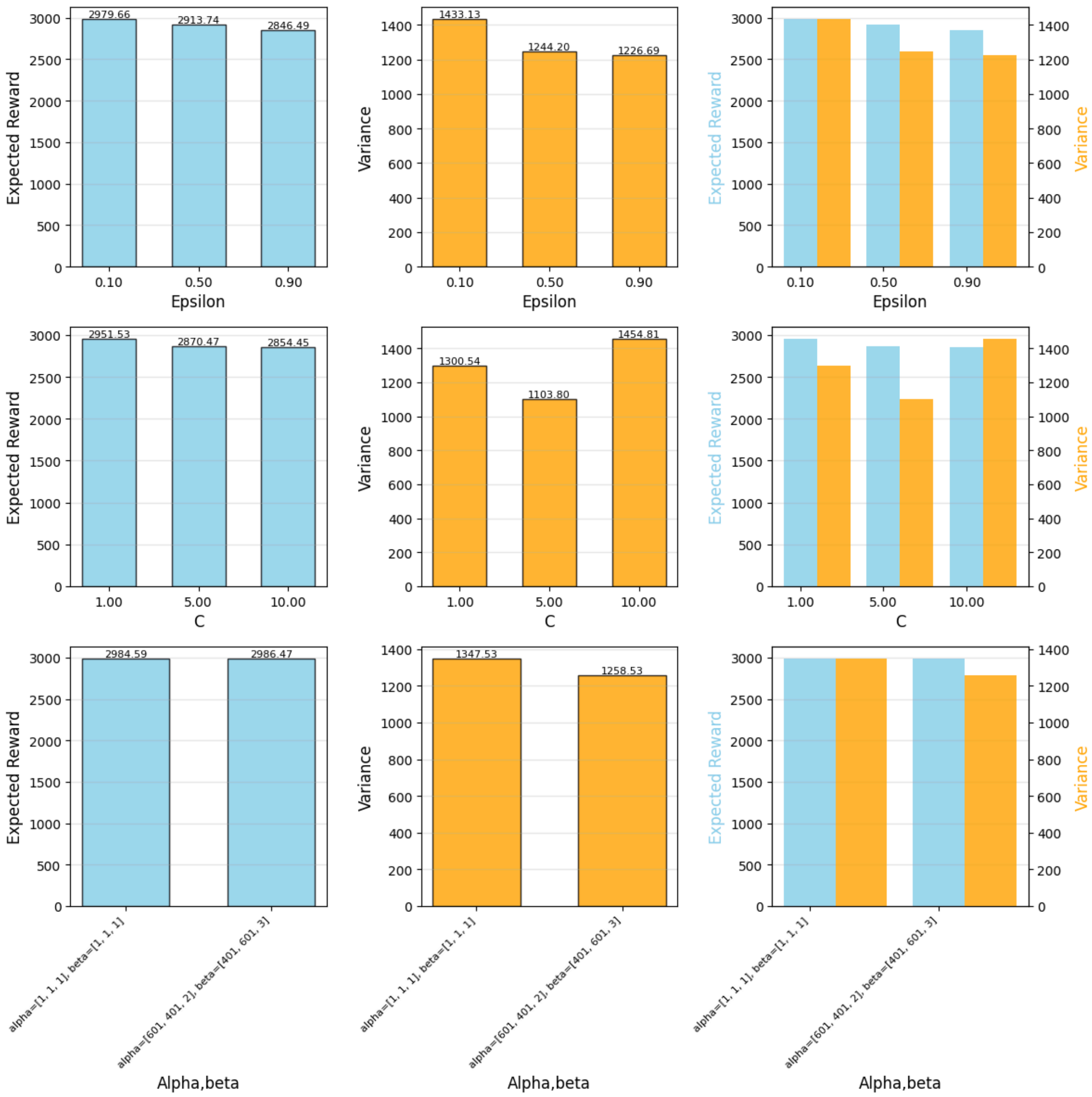
Here are the results of the original algorithms.

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
experiments = {
    'epsilon_greedy': [0.1, 0.5, 0.9],
    'ucb': [1, 5, 10],
    'ts': [
        ([1, 1, 1], [1, 1, 1]),
        ([601, 401, 2], [401, 601, 3])
    ]
}
results_rewards = {
    'epsilon_greedy': [],
    'ucb': [],
    'ts': []
}
results_variances = {
    'epsilon_greedy': [],
    'ucb': [],
    'ts': []
}
results_regrets = {
    'epsilon_greedy': [],
    'ucb': [],
    'ts': []
}

for key in ['epsilon_greedy', 'ucb', 'ts']:
    for value in experiments[key]:
        trial_rewards_list = []
        cumulative_rewards = 0
        cumulative_regrets = 0

        for _ in range(num_trials):
            if key == 'epsilon_greedy':
                bandit = EpsilonGreedy(theta = [0.6, 0.6, 0.5], epsilon=v
            elif key == 'ucb':
                bandit = UCB(theta = [0.6, 0.6, 0.5], c=value)
            elif key == 'ts':
                bandit = ThompsonSampling(theta = [0.6, 0.6, 0.5], alpha=

            trial_rewards = []
            for t in range(N):
```

```
                arm = bandit.select_arm()
                reward = bandit.pull(arm)
                bandit.update(arm, reward)
                trial_rewards.append(reward)
                cumulative_rewards += reward
                regret = max(bandit.theta) - bandit.theta[arm]
                cumulative_regrets += regret

            trial_rewards_list.append(np.sum(trial_rewards))

        mean_reward = cumulative_rewards / num_trials
        variance_reward = np.var(trial_rewards_list)

        results_rewards[key].append(mean_reward)
        results_variances[key].append(variance_reward)
        results_regrets[key].append(cumulative_regrets / num_trials)

print("epsilon = 0.1   reward: ", results_rewards['epsilon_greedy'][0], "
print("epsilon = 0.5   reward: ", results_rewards['epsilon_greedy'][1], "
print("epsilon = 0.9   reward: ", results_rewards['epsilon_greedy'][2], "
print("c = 1          reward: ", results_rewards['ucb'][0], " regret: ",
print("c = 5          reward: ", results_rewards['ucb'][1], " regret: ",
print("c = 10         reward: ", results_rewards['ucb'][2], " regret: ",
print("alpha = [1,1,1] beta = [1,1,1] reward: ", results_rewards['ts'][0]
print("alpha = [601,401,2] beta = [401,601,3] reward: ", results_rewards[

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)
```

```
def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{epsilon:.2f}" for epsilon in param_values]

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(3, 3, figsize=(12, 12))

# epsilon_greedy
plot_expectation(axs[0, 0], results_rewards['epsilon_greedy'], experiment
plot_variance(axs[0, 1], results_variances['epsilon_greedy'], experiments
plot_combined(axs[0, 2], results_rewards['epsilon_greedy'], results_varia

# ucb
plot_expectation(axs[1, 0], results_rewards['ucb'], experiments['ucb'], "
plot_variance(axs[1, 1], results_variances['ucb'], experiments['ucb'], "c
plot_combined(axs[1, 2], results_rewards['ucb'], results_variances['ucb']

# ts
plot_expectation(axs[2, 0], results_rewards['ts'], experiments['ts'], "al
plot_variance(axs[2, 1], results_variances['ts'], experiments['ts'], "alp
plot_combined(axs[2, 2], results_rewards['ts'], results_variances['ts'],

plt.tight_layout()
plt.show()
```

```
epsilon = 0.1   reward:  2979.665  regret:  21.851499999991837  variances:
1433.1327750000003
epsilon = 0.5   reward:  2913.74  regret:  84.44750000017359  variances:
1244.2024
epsilon = 0.9   reward:  2846.485  regret:  150.44349999921323  variances:
1226.6897749999998
c = 1          reward:  2951.525  regret:  48.40550000008844  variances:
1300.5393749999998
c = 5          reward:  2870.475  regret:  127.62349999954529  variances:
1103.799375
c = 10         reward:  2854.45  regret:  145.85499999927998  variances:
1454.8075
alpha = [1,1,1] beta = [1,1,1] reward:  2984.585  regret:  13.711499999993
007  variances:  1347.532775
alpha = [601,401,2] beta = [401,601,3] reward:  2986.47  regret:  12.81499
9999993823  variances:  1258.5291
```



To improve the performance, we modify the ϵ-greedy algorithm.

Each time we pull arm 1 (with the result denoted as r_i), we update the counts for both arm 1 and arm 2 by incrementing them by 1. Additionally, we update the values of θ for both arms using the following update rule:

$$\theta(1) \leftarrow \theta(1) + \frac{1}{\text{count}(1)}(r_i - \theta(1))$$

$$\theta(2) \leftarrow \theta(2) + \frac{1}{\text{count}(2)}(r_i - \theta(2))$$

Similarly, each time we pull arm 2, we apply the same updates. However, after pulling arm 3, we revert to the standard ϵ-greedy update mechanism.

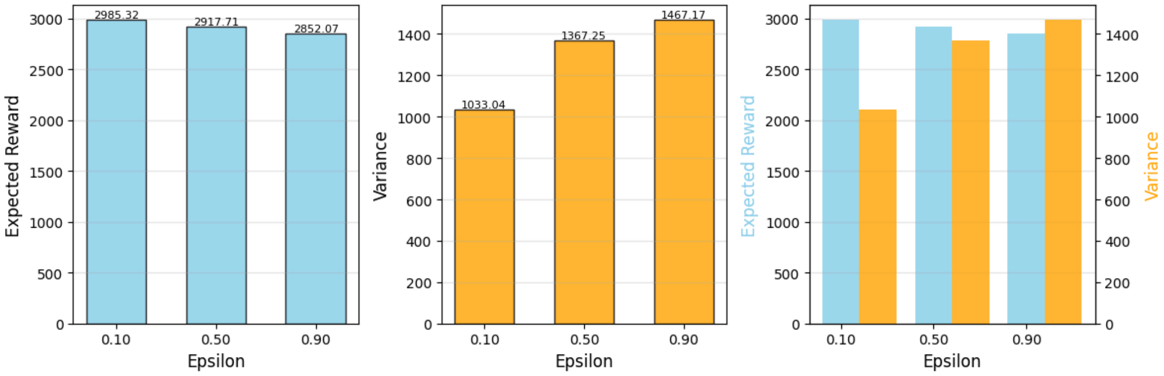
Here are the results of the modified-ϵ-greedy algorithm.

In [13]:
```
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
```

```
experiments = {
    'epsilon_greedy': [0.1, 0.5, 0.9],
}
results_rewards = {
    'epsilon_greedy': [],
}
results_variances = {
    'epsilon_greedy': [],
}
results_regrets = {
    'epsilon_greedy': [],
}

for key in ['epsilon_greedy']:
    for value in experiments[key]:
        trial_rewards_list = []
        cumulative_rewards = 0
        cumulative_regrets = 0

        for _ in range(num_trials):
            bandit = EpsilonGreedy(theta = [0.6, 0.6, 0.5], epsilon=value

            trial_rewards = []
            for t in range(N):
                arm = bandit.select_arm()
                reward = bandit.pull(arm)
                if arm == 1 or arm == 2:
                    bandit.update(1, reward)
                    bandit.update(2, reward)
                else:
                    bandit.update(arm, reward)

                trial_rewards.append(reward)
                cumulative_rewards += reward

                regret = max(bandit.theta) - bandit.theta[arm]
                cumulative_regrets += regret

            trial_rewards_list.append(np.sum(trial_rewards))

        mean_reward = cumulative_rewards / num_trials
        variance_reward = np.var(trial_rewards_list)

        results_rewards[key].append(mean_reward)
        results_variances[key].append(variance_reward)
        results_regrets[key].append(cumulative_regrets / num_trials)

print("epsilon = 0.1   reward: ", results_rewards['epsilon_greedy'][0], "
print("epsilon = 0.5   reward: ", results_rewards['epsilon_greedy'][1], "
print("epsilon = 0.9   reward: ", results_rewards['epsilon_greedy'][2], "

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
```

```
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{epsilon:.2f}" for epsilon in param_values]

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(1, 3, figsize=(12, 4))

# epsilon_greedy
plot_expectation(axs[0], results_rewards['epsilon_greedy'], experiments['
plot_variance(axs[1], results_variances['epsilon_greedy'], experiments['e
plot_combined(axs[2], results_rewards['epsilon_greedy'], results_variance

plt.tight_layout()
plt.show()
```

```
epsilon = 0.1   reward:  2985.315  regret:  16.66599999999032  variances:
1033.035775
epsilon = 0.5   reward:  2917.71  regret:  83.56750000018639  variances:
1367.2459000000003
epsilon = 0.9   reward:  2852.065  regret:  149.88299999922137  variances:
1467.170775
```



We also modify the UCB algorithm.

Similarly, each time we pull arm 1 (with the result denoted as r_i), we update the counts for both arm 1 and arm 2 by incrementing them by 1. Additionally, we update the values of θ for both arms using the following update rule:

$$\theta(1) \leftarrow \theta(1) + \frac{1}{\text{count}(1)}(r_i - \theta(1))$$

$$\theta(2) \leftarrow \theta(2) + \frac{1}{\text{count}(2)}(r_i - \theta(2))$$

Each time we pull arm 2, we apply the same updates. However, after pulling arm 3, we revert to the standard UCB update mechanism.

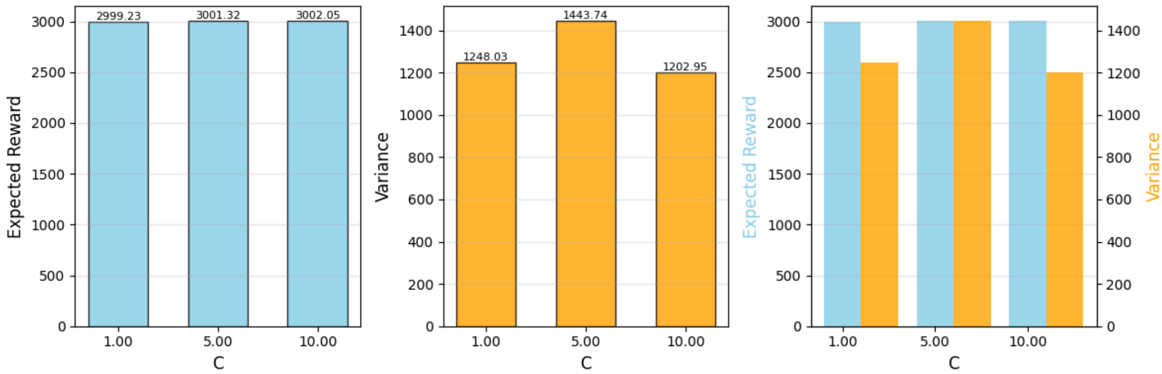
Here are the results of the modified-UCB algorithm.

In [20]:
```python
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
experiments = {
    'ucb': [1, 5, 10],
}
results_rewards = {
    'ucb': [],
}
results_variances = {
    'ucb': [],
}
results_regrets = {
    'ucb': [],
}

for key in ['ucb']:
    for value in experiments[key]:
        trial_rewards_list = []
        cumulative_rewards = 0
```

```
        cumulative_regrets = 0

        for _ in range(num_trials):
            bandit = UCB(theta = [0.6, 0.6, 0.5], c=value)

            trial_rewards = []
            for t in range(N):
                arm = bandit.select_arm()
                reward = bandit.pull(arm)
                if arm == 1 or arm == 2:
                    bandit.update(1, reward)
                    bandit.update(2, reward)
                else:
                    bandit.update(arm, reward)

                trial_rewards.append(reward)
                cumulative_rewards += reward

                regret = max(bandit.theta) - bandit.theta[arm]
                cumulative_regrets += regret

            trial_rewards_list.append(np.sum(trial_rewards))

        mean_reward = cumulative_rewards / num_trials
        variance_reward = np.var(trial_rewards_list)

        results_rewards[key].append(mean_reward)
        results_variances[key].append(variance_reward)
        results_regrets[key].append(cumulative_regrets / num_trials)

print("c = 1          reward: ", results_rewards['ucb'][0], " regret: ",
print("c = 5          reward: ", results_rewards['ucb'][1], " regret: ",
print("c = 10         reward: ", results_rewards['ucb'][2], " regret: ",

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
```

```
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{epsilon:.2f}" for epsilon in param_values]

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)

    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(1, 3, figsize=(12, 4))

# ucb
plot_expectation(axs[0], results_rewards['ucb'], experiments['ucb'], "c")
plot_variance(axs[1], results_variances['ucb'], experiments['ucb'], "c")
plot_combined(axs[2], results_rewards['ucb'], results_variances['ucb'], e

plt.tight_layout()
plt.show()
```

```
c = 1          reward:  2999.23  regret:  0.0  variances:  1248.0271
c = 5          reward:  3001.325  regret:  0.0  variances:  1443.739375
c = 10         reward:  3002.05  regret:  0.0  variances:  1202.9475
```



Now let us introduce our modified-Thompson Sampling algorithm.

Each time we pull arm 1 (with the result denoted as r_i), we update the α value for both arm 1 and arm 2 by incrementing them by r_i, and update the β value for both arm 1 and arm 2 by incrementing them by $1 - r_i$.

Each time we pull arm 2, we apply the same updates. However, after pulling arm 3, we revert to the standard TS update mechanism.

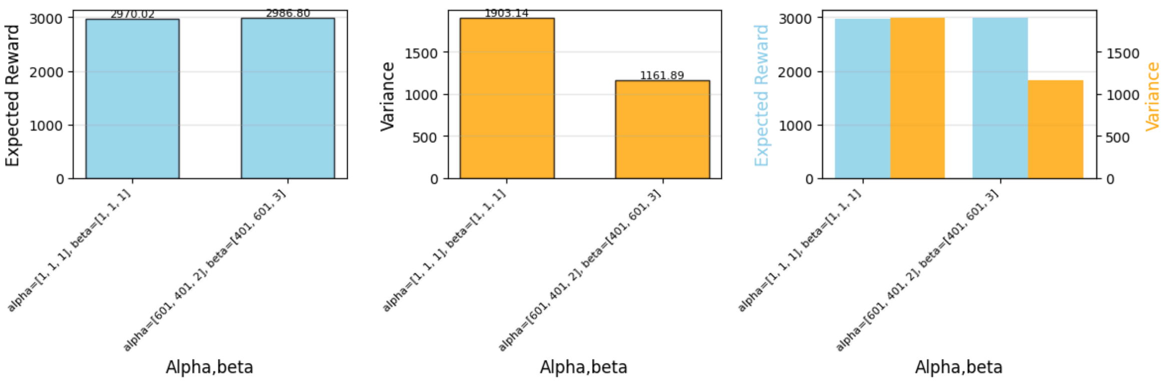
Here are the results of the modified-TS algorithm.

In [17]:
```
import numpy as np
import matplotlib.pyplot as plt

N = 5000
num_trials = 200
experiments = {
    'ts': [
        ([1, 1, 1], [1, 1, 1]),
        ([601, 401, 2], [401, 601, 3])
    ]
}
results_rewards = {
    'ts': []
}
results_variances = {
    'ts': []
}
results_regrets = {
    'ts': []
}

for key in ['ts']:
    for value in experiments[key]:
        trial_rewards_list = []
        cumulative_rewards = 0
        cumulative_regrets = 0

        for _ in range(num_trials):
            bandit = ThompsonSampling(theta = [0.6, 0.6, 0.5], alpha=valu

            trial_rewards = []
            for t in range(N):
                arm = bandit.select_arm()
                reward = bandit.pull(arm)
                if arm == 1 or arm == 2:
                    bandit.update(1, reward)
                    bandit.update(2, reward)
                else:
                    bandit.update(arm, reward)

                trial_rewards.append(reward)
                cumulative_rewards += reward

                regret = max(bandit.theta) - bandit.theta[arm]
                cumulative_regrets += regret

            trial_rewards_list.append(np.sum(trial_rewards))

        mean_reward = cumulative_rewards / num_trials
```

```
        variance_reward = np.var(trial_rewards_list)

        results_rewards[key].append(mean_reward)
        results_variances[key].append(variance_reward)
        results_regrets[key].append(cumulative_regrets / num_trials)

print("alpha = [1,1,1] beta = [1,1,1] reward: ", results_rewards['ts'][0]
print("alpha = [601,401,2] beta = [401,601,3] reward: ", results_rewards[

def plot_expectation(ax, means, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, means, width=0.6, color='skyblue', edgecolor='b

    for i, mean in enumerate(means):
        ax.text(bar_positions[i], mean + 0.01, f"{mean:.2f}", ha='center'
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_variance(ax, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    ax.bar(bar_positions, variances, width=0.6, color='orange', edgecolor

    for i, variance in enumerate(variances):
        ax.text(bar_positions[i], variance + 0.01, f"{variance:.2f}", ha=
    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{val:.2f}" for val in param_values], fontsi

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Variance", fontsize=12)
    ax.grid(axis='y', alpha=0.3)

def plot_combined(ax, means, variances, param_values, param_name):
    bar_positions = np.arange(len(param_values))
    bar_width = 0.4

    ax.bar(bar_positions, means, bar_width, alpha=0.8, color='skyblue', l

    ax.set_xticks(bar_positions)

    if param_name == "alpha,beta":
        ax.set_xticklabels([f"alpha={val[0]}, beta={val[1]}" for val in p
    else:
        ax.set_xticklabels([f"{epsilon:.2f}" for epsilon in param_values]

    ax.set_xlabel(f"{param_name.capitalize()}", fontsize=12)
    ax.set_ylabel("Expected Reward", fontsize=12, color='skyblue')
    ax.grid(axis='y', alpha=0.3)
```

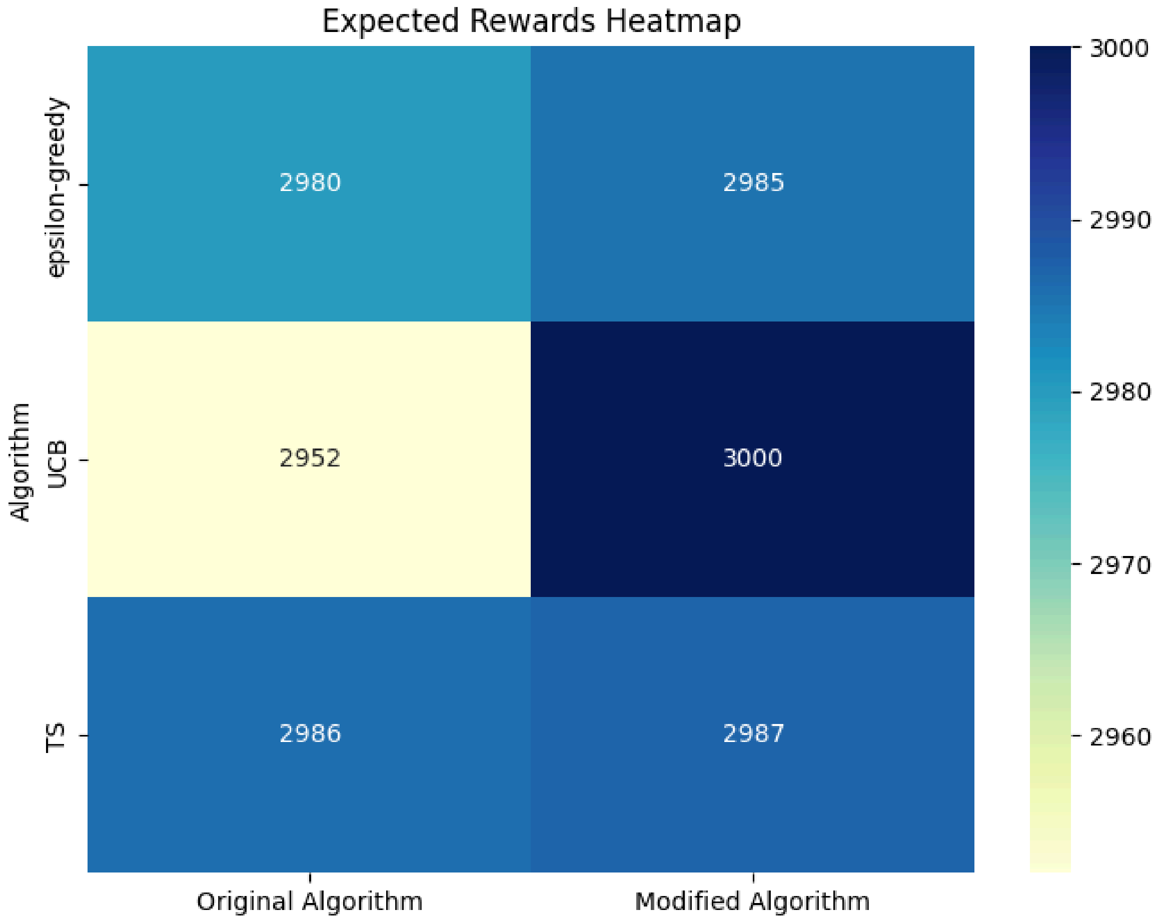
```
    ax2 = ax.twinx()

    ax2.bar(bar_positions + bar_width, variances, bar_width, alpha=0.8, c

    ax2.set_ylabel("Variance", fontsize=12, color='orange')

fig, axs = plt.subplots(1, 3, figsize=(12, 4))

# ts
plot_expectation(axs[0], results_rewards['ts'], experiments['ts'], "alpha
plot_variance(axs[1], results_variances['ts'], experiments['ts'], "alpha,
plot_combined(axs[2], results_rewards['ts'], results_variances['ts'], exp

plt.tight_layout()
plt.show()
```

```
alpha = [1,1,1] beta = [1,1,1] reward:  2970.02  regret:  33.2035000000331
4  variances:  1903.1396000000002
alpha = [601,401,2] beta = [401,601,3] reward:  2986.8  regret:  13.382499
999993307  variances:  1161.89
```



Now compare the results of the original algorithms and the modified algorithms.

In [19]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

data = {
    "Algorithm": ["epsilon-greedy", "UCB", "TS"],
    "Original Algorithm": [2980, 2952, 2986],
    "Modified Algorithm": [2985, 3000, 2987]
}

df = pd.DataFrame(data)

df.set_index('Algorithm', inplace=True)

plt.figure(figsize=(8, 6))
sns.heatmap(df, annot=True, cmap="YlGnBu", fmt="d", cbar=True)

plt.title("Expected Rewards Heatmap")
plt.show()
```



Our modified algorithms leverage the dependencies between different arms to make exploration more efficient. As a result, the performance of the modified algorithms surpasses that of the original ones. Notably, the modified UCB algorithm is very close to the upper bound of the expected rewards, highlighting its effectiveness.

(2) More general cases:

We have examined a specific case to test our modified algorithm. However, in more general scenarios, the problem becomes more complex. Nevertheless, the optimization approach remains the same. We should continue to leverage the dependencies between arms to accelerate the exploration process.

Part II: Bayesian Bandit Algorithms

There are two arms which may be pulled repeatedly in any order. Each pull may result in either a success or a failure. The sequence of successes and failures which results from pulling arm i ($i \in \{1, 2\}$) forms a Bernoulli process with unknown success probability θ_i. A success at the t^{th} pull yields a reward γ^{t-1} ($0 < \gamma < 1$), while an unsuccessful pull yields a zero reward. At time zero, each θ_i has a Beta prior distribution with two parameters α_i, β_i and these distributions are independent for different arms. These prior distributions are updated to posterior distributions as arms are pulled. Since the class of Beta distributions is closed under Bernoulli sampling, posterior distributions are all Beta distributions. How should the arm to pull next in each time slot be chosen to maximize the total expected reward from an infinite sequence of pulls?

1. One intuitive policy suggests that in each time slot we should pull the arm for which the current expected value of θ_i is the largest. This policy behaves very good in most cases. Please design simulations to check the behavior of this policy.

In [8]:
```
import numpy as np
from tqdm import tqdm

def intuitive_policy(N, gamma, true_theta, alpha, beta):
    """
    Implements the intuitive policy for a two-armed bandit problem with d

    Args:
        N (int): Number of time steps
        gamma (float): Discount factor
        true_theta (np.ndarray): True probabilities for each arm
        alpha (list): Initial alpha parameters for Beta distribution
        beta (list): Initial beta parameters for Beta distribution

    Returns:
        float: Sum of discounted rewards
    """
    rewards = np.zeros(N)
    alpha = np.array(alpha)
    beta = np.array(beta)

    for t in range(N):
        theta_estimate = alpha / (alpha + beta)
        chosen_arm = np.argmax(theta_estimate)

        reward = np.random.rand() < true_theta[chosen_arm]
        rewards[t] = reward * (gamma ** t)

        alpha[chosen_arm] += reward
        beta[chosen_arm] += 1 - reward

    return np.sum(rewards)

num_trials = 200
```

```python
N = 5000
gamma_list = [0.1, 0.3, 0.5, 0.7, 0.9, 0.99]
alpha_list = [[1, 1], [2, 1], [20, 1]]
beta_list = [[1, 1], [1, 1], [10, 1]]

for gamma in gamma_list:
    for alpha, beta in zip(alpha_list, beta_list):
        rewards = np.zeros(num_trials)
        regret_rate = np.zeros(num_trials)

        for i in tqdm(range(num_trials)):
            true_theta = np.random.rand(2)

            rewards[i] = intuitive_policy(N, gamma, true_theta, alpha, be
            max_value = np.max(true_theta) / (1 - gamma)
            regret_rate[i] = 1 - rewards[i] / max_value

        print(f"gamma: {gamma}, alpha: {alpha}, beta: {beta}, "
              f"avg_reward: {np.mean(rewards):.4f}, "
              f"avg_regret_rate: {np.mean(regret_rate):.4f}")
```

```
100%|██████████| 200/200 [00:03<00:00, 51.54it/s]
gamma: 0.1, alpha: [1, 1], beta: [1, 1], avg_reward: 0.5534, avg_regret_ra
te: 0.2794
100%|██████████| 200/200 [00:03<00:00, 51.16it/s]
gamma: 0.1, alpha: [2, 1], beta: [1, 1], avg_reward: 0.5405, avg_regret_ra
te: 0.2721
100%|██████████| 200/200 [00:03<00:00, 51.99it/s]
gamma: 0.1, alpha: [20, 1], beta: [10, 1], avg_reward: 0.5649, avg_regret_
rate: 0.2339
100%|██████████| 200/200 [00:03<00:00, 53.22it/s]
gamma: 0.3, alpha: [1, 1], beta: [1, 1], avg_reward: 0.7975, avg_regret_ra
te: 0.1243
100%|██████████| 200/200 [00:04<00:00, 48.74it/s]
gamma: 0.3, alpha: [2, 1], beta: [1, 1], avg_reward: 0.7813, avg_regret_ra
te: 0.2369
100%|██████████| 200/200 [00:03<00:00, 50.42it/s]
gamma: 0.3, alpha: [20, 1], beta: [10, 1], avg_reward: 0.6766, avg_regret_
rate: 0.2897
100%|██████████| 200/200 [00:03<00:00, 51.15it/s]
gamma: 0.5, alpha: [1, 1], beta: [1, 1], avg_reward: 1.0809, avg_regret_ra
te: 0.1999
100%|██████████| 200/200 [00:03<00:00, 50.66it/s]
gamma: 0.5, alpha: [2, 1], beta: [1, 1], avg_reward: 1.0306, avg_regret_ra
te: 0.2144
100%|██████████| 200/200 [00:03<00:00, 52.67it/s]
gamma: 0.5, alpha: [20, 1], beta: [10, 1], avg_reward: 0.9873, avg_regret_
rate: 0.2047
100%|██████████| 200/200 [00:03<00:00, 52.06it/s]
gamma: 0.7, alpha: [1, 1], beta: [1, 1], avg_reward: 2.0485, avg_regret_ra
te: 0.1281
100%|██████████| 200/200 [00:03<00:00, 52.50it/s]
gamma: 0.7, alpha: [2, 1], beta: [1, 1], avg_reward: 1.8086, avg_regret_ra
te: 0.2016
100%|██████████| 200/200 [00:03<00:00, 50.82it/s]
gamma: 0.7, alpha: [20, 1], beta: [10, 1], avg_reward: 1.6074, avg_regret_
rate: 0.2460
```

```
100%|██████████| 200/200 [00:03<00:00, 53.62it/s]
gamma: 0.9, alpha: [1, 1], beta: [1, 1], avg_reward: 6.2194, avg_regret_ra
te: 0.0916
100%|██████████| 200/200 [00:03<00:00, 52.34it/s]
gamma: 0.9, alpha: [2, 1], beta: [1, 1], avg_reward: 6.0296, avg_regret_ra
te: 0.0888
100%|██████████| 200/200 [00:03<00:00, 52.35it/s]
gamma: 0.9, alpha: [20, 1], beta: [10, 1], avg_reward: 5.4094, avg_regret_
rate: 0.2213
100%|██████████| 200/200 [00:03<00:00, 51.78it/s]
gamma: 0.99, alpha: [1, 1], beta: [1, 1], avg_reward: 61.6180, avg_regret_
rate: 0.0568
100%|██████████| 200/200 [00:03<00:00, 53.32it/s]
gamma: 0.99, alpha: [2, 1], beta: [1, 1], avg_reward: 64.7995, avg_regret_
rate: 0.0434
100%|██████████| 200/200 [00:03<00:00, 53.15it/s]
gamma: 0.99, alpha: [20, 1], beta: [10, 1], avg_reward: 61.9752, avg_regre
t_rate: 0.1035
```

To evaluate the performance of the algorithm, we need to find a suitable metric. Regret seems to be a good choice, but it is not normalized, leading to different scales for different settings. (For example, larger γ leads to larger regret.) Thus, we use the regret rate, which shows the portion of regret to the maximum expected reward. The regret rate is defined as follow:

$$\text{regret rate} = 1 - \frac{Reward}{\max_i \theta_i/(1-\gamma)}$$

where the maximum possible reward is achieved by always pulling the arm with the highest true probability. For the discounted setting, this equals $\frac{\max_i \theta_i}{1-\gamma}$.

The simulation results show that the intuitive policy performs well in most cases, achieving low regret rates.

γ	Prior (α, β)	Average Reward	Average Regret Rate
0.1	[1,1], [1,1]	0.5534	0.2794
0.1	[2,1], [1,1]	0.5405	0.2721
0.1	[20,1], [10,1]	0.5649	0.2339
0.3	[1,1], [1,1]	0.7975	0.1243
0.3	[2,1], [1,1]	0.7813	0.2369
0.3	[20,1], [10,1]	0.6766	0.2897
0.5	[1,1], [1,1]	1.0809	0.1999
0.5	[2,1], [1,1]	1.0306	0.2144
0.5	[20,1], [10,1]	0.9873	0.2047
0.7	[1,1], [1,1]	2.0485	0.1281
0.7	[2,1], [1,1]	1.8086	0.2016
0.7	[20,1], [10,1]	1.6074	0.2460

γ	Prior (α, β)	Average Reward	Average Regret Rate
0.9	[1,1], [1,1]	6.2194	0.0916
0.9	[2,1], [1,1]	6.0296	0.0888
0.9	[20,1], [10,1]	5.4094	0.2213
0.99	[1,1], [1,1]	61.6180	0.0568
0.99	[2,1], [1,1]	64.7995	0.0434
0.99	[20,1], [10,1]	61.9752	0.1035

This can be attributed to several factors:

1. **Efficient Exploration**: The policy naturally balances exploration and exploitation through Bayesian updating of the Beta distributions.

2. **Prior Knowledge Integration**: The Beta distribution parameters (α, β) allow incorporating prior knowledge about the arms, which helps guide initial exploration.

3. **Quick Convergence**: As more rewards are observed, the posterior distributions quickly concentrate around the true probabilities, leading to optimal arm selection.

Looking at the simulation results across different discount factors (γ) and prior parameters (α, β), we see consistently low regret rates, indicating the policy's robustness to different parameter settings. However, as we'll see in the counter-example, there are specific scenarios where this policy can be suboptimal.

2. However, such intuitive policy is unfortunately not optimal. Please provide an example to show why such policy is not optimal.

In [9]:
```
# Here's a counter-example with a strongly biased prior, but the second a
# We'll run fewer steps (N_short=100), so the policy doesn't have time to

test_runs = 1000
test_rewards = np.zeros(test_runs)
N_short = 100
gamma_close_to_1 = 0.99


counter_alpha = [200, 1]
counter_beta = [1, 1]
counter_true_theta = np.array([0.6, 0.8])  # second arm has higher probab

for i in range(test_runs):
    local_alpha = counter_alpha.copy()
    local_beta = counter_beta.copy()
    test_rewards[i] = intuitive_policy(N_short, gamma_close_to_1, counter

avg_r = np.mean(test_rewards)
optimal_r = np.max(counter_true_theta) / (1 - gamma_close_to_1)
```

```
print(f"Average reward: {avg_r:.2f}")
print(f"Maximum possible reward: {optimal_r:.2f}")
print(f"Regret rate: {1 - avg_r/optimal_r:.2f}")
```

```
Average reward: 38.13
Maximum possible reward: 80.00
Regret rate: 0.52
```

Given two arms with prior distributions:

Arm1 Beta(200,1), suggesting an expected value close to 1.0

Arm2 Beta(1,1), suggesting an expected value close to 0.5

A greedy strategy that consistently favors the arm with the higher expected value may lead to repeatedly selecting Arm 1. However, this approach has critical flaws. The prior for Arm 2 suggests significant uncertainty, as the Beta(1, 1) distribution is essentially non-informative, assigning equal probability to all values between 0 and 1.

By selecting Arm 2 more frequently, we can reduce this uncertainty and potentially uncover a true value for Arm 2 that exceeds that of Arm 1.

Focusing exclusively on Arm 1 due to its higher initial expected value neglects the possibility that Arm 2 could ultimately provide greater rewards once more data is collected. Failing to explore Arm 2 adequately risks missing out on higher returns that could arise if its true value is found to be higher than initially estimated.

When priors differ significantly in terms of uncertainty, a strategy that relies solely on expected values can lead to consistently selecting a suboptimal arm. It is crucial to strike a balance between exploiting known information and exploring uncertain but potentially more rewarding alternatives.

3. For the expected total reward under an optimal policy, show that the following recurrence equation holds:

$$\begin{aligned}
R_1(\alpha_1,\beta_1) =& \frac{\alpha_1}{\alpha_1+\beta_1}[1+\gamma R(\alpha_1+1,\beta_1,\alpha_2,\beta_2)] \\
&+ \frac{\beta_1}{\alpha_1+\beta_1}[\gamma R(\alpha_1,\beta_1+1,\alpha_2,\beta_2)]; \\
R_2(\alpha_2,\beta_2) =& \frac{\alpha_2}{\alpha_2+\beta_2}[1+\gamma R(\alpha_1,\beta_1,\alpha_2+1,\beta_2)] \\
&+ \frac{\beta_2}{\alpha_2+\beta_2}[\gamma R(\alpha_1,\beta_1,\alpha_2,\beta_2+1)]; \\
R(\alpha_1,\beta_1,\alpha_2,\beta_2) =& \max\left\{R_1(\alpha_1,\beta_1), R_2(\alpha_2,\beta_2)\right\}.
\end{aligned}$$

We first consider the case of pulling arm 1

When pulling arm 1:

- Success occurs with probability $\frac{\alpha_1}{\alpha_1+\beta_1}$ (mean of Beta distribution)
  - Immediate reward: 1

Since its Bayesian Inferece, with Beta-Binoimal conjugate, so the posterior distribution of θ_1 is still a Beta distribution, The future expected reward, considering a success, updates the parameters to $Beta(\alpha_1+1, \beta_1)$ and $Beta(\alpha_1, \beta_1+1)$ considering failure.

So the next steps' rewards is $R(\alpha_1+1, \beta_1, \alpha_2, \beta_2)$ when success at this time, and $R(\alpha_1, \beta_1+1, \alpha_2, \beta_2)$ when failure at this time.

Since at the t^{th} pull yields a reward γ^{t-1} $(0<\gamma<1)$, which means that the future's reward is will recieve a discount γ for each time.

Considering a sucess at this time

So for this time, if it sucess, we can recieve the reward 1. And the parameters become $(\alpha_1+1, \beta_1, \alpha_2, \beta_2)$ due to the Beta-Binoimal conjugate. After the discount, the future's reward is $\gamma R(\alpha_1+1, \beta_1, \alpha_2, \beta_2)$.

Also, since success happens with probability θ_1. So the total rewards when success at this time is

$$\theta_1[1+\gamma R(\alpha_1+1, \beta_1, \alpha_2, \beta_2)]$$

Considering a failure at this time

For this time, if it fail, we can recieve the reward 0. And the parameters become $(\alpha_1, \beta_1+1, \alpha_2, \beta_2)$ due to the Beta-Binoimal conjugate. After the discount, the future's reward is $0+\gamma R(\alpha_1, \beta_1+1, \alpha_2, \beta_2)$.

Also, since failure happens with probability $1-\theta_1$. So the total rewards when success at this time is

$$(1-\theta_1)[0+\gamma R(\alpha_1, \beta_1+1, \alpha_2, \beta_2)] = (1-\theta_1)[\gamma R(\alpha_1, \beta_1+1, \alpha_2, \beta_2)]$$

So combine the two parts, we can get that the total rewards when pull the first arm is that

$$R_1(\alpha_1, \beta_1) = \theta_1[1+\gamma R(\alpha_1+1, \beta_1, \alpha_2, \beta_2)] + (1-\theta_1)[\gamma R(\alpha_1, \beta_1+1, \alpha_2, \beta_2)]$$

$$R_1(\alpha_1, \beta_1) = \frac{\alpha_1}{\alpha_1+\beta_1}[1+\gamma R(\alpha_1+1, \beta_1, \alpha_2, \beta_2)] + \frac{\beta_1}{\alpha_1+\beta_1}[\gamma R(\alpha_1, \beta_1+1, \alpha_2, \beta$$

Similar Reasoning for Arm 2:

The expected reward for pulling arm 2 follows the same logic, adjusting for the parameters of arm 2:

Similarly, since

$$\theta_2 \sim Beta(\alpha_2, \beta_2)$$

So with the same method above, we can get that:

$$R_2(\alpha_2, \beta_2) = \frac{\alpha_2}{\alpha_2 + \beta_2}[1 + \gamma R(\alpha_1, \beta_1, \alpha_2 + 1, \beta_2)] + \frac{\beta_2}{\alpha_2 + \beta_2}[\gamma R(\alpha_2, \beta_1, \alpha_2, \beta_2 +$$

And since we want to maximize the total reward, so we can get that:

$$R(\alpha_1, \beta_1, \alpha_2, \beta_2) = \max\{R_1(\alpha_1, \beta_1), R_2(\alpha_2, \beta_2)\}$$

So above all, the following recurrence equation holds have been proven.

$$R_1(\alpha_1, \beta_1) = \frac{\alpha_1}{\alpha_1 + \beta_1}[1 + \gamma R(\alpha_1 + 1, \beta_1, \alpha_2, \beta_2)] + \frac{\beta_1}{\alpha_1 + \beta_1}[\gamma R(\alpha_1, \beta_1 + 1, \alpha_2, \beta$$

$$R_2(\alpha_2, \beta_2) = \frac{\alpha_2}{\alpha_2 + \beta_2}[1 + \gamma R(\alpha_1, \beta_1, \alpha_2 + 1, \beta_2)] + \frac{\beta_2}{\alpha_2 + \beta_2}[\gamma R(\alpha_2, \beta_1, \alpha_2, \beta_2 +$$

$$R(\alpha_1, \beta_1, \alpha_2, \beta_2) = \max\{R_1(\alpha_1, \beta_1), R_2(\alpha_2, \beta_2)\}$$

4 For the above equations, our solution:

To solve the recursive equations, we use an approximate method since solving them exactly is impractical due to the infinite number of states and the absence of clear boundaries. In our approach, we introduce a counter to track the number of times each arm has been pulled. Once the counter exceeds 100 pulls, we assume that the exploration phase has provided sufficient information about the arms. At this point, we transition to the exploitation phase, where we choose the arm with the higher mean value. The mean for each arm is computed as $\frac{\alpha}{\alpha+\beta}$, based on its Beta distribution parameters.

To enhance efficiency, we implement a small optimization by using a dictionary to store the results of states that have already been calculated. This prevents redundant computations and accelerates the process significantly, as many states are encountered repeatedly. This optimization is conceptually similar to memoization in dynamic programming, where previously computed results are reused to avoid recalculating them. By adopting this approach, we strike a balance between exploration and exploitation, while also improving the computational efficiency of our algorithm.

In [10]:
```
results_cache = {}
policy = {}

def calculate_R(alpha1, beta1, alpha2, beta2, discount_factor, exploratio
    if (alpha1, beta1, alpha2, beta2) in results_cache:
        return results_cache[(alpha1, beta1, alpha2, beta2)]

    if exploration_count > max_exploration:
        mean_arm1 = alpha1 / (alpha1 + beta1)
        mean_arm2 = alpha2 / (alpha2 + beta2)
        results_cache[(alpha1, beta1, alpha2, beta2)] = max(mean_arm1, me
        policy[(alpha1, beta1, alpha2, beta2)] = mean_arm1 > mean_arm2
        return max(mean_arm1, mean_arm2)

    expected_reward_arm1 = (
```

```
        (alpha1 / (alpha1 + beta1)) * (1 + discount_factor * calculate_R(
        + (beta1 / (alpha1 + beta1)) * (discount_factor * calculate_R(alp
    )

    expected_reward_arm2 = (
        (alpha2 / (alpha2 + beta2)) * (1 + discount_factor * calculate_R(
        + (beta2 / (alpha2 + beta2)) * (discount_factor * calculate_R(alp
    )

    max_reward = max(expected_reward_arm1, expected_reward_arm2)

    results_cache[(alpha1, beta1, alpha2, beta2)] = max_reward
    policy[(alpha1, beta1, alpha2, beta2)] = expected_reward_arm1 > expec

    return max_reward
```

The code above contains our implementation.

Another possible solution is by using Q-Learning. As we can regard R_1 and R_2 as the Q-value of the state, and R as the value of the state, the problem is actually a Markov Decision Process (MDP) problem. We can use Q-learning or other reinforcement learning algorithms to solve it.

In [11]:
```
import numpy as np

class BayesianBanditQLearning:
    def __init__(self, alpha1, beta1, alpha2, beta2, gamma, learning_rate
        self.alpha1 = alpha1
        self.beta1 = beta1
        self.alpha2 = alpha2
        self.beta2 = beta2
        self.gamma = gamma
        self.learning_rate = learning_rate
        self.epsilon = epsilon
        self.q_values = {}

    def get_state_key(self, alpha1, beta1, alpha2, beta2):
        return (alpha1, beta1, alpha2, beta2)

    def get_q_value(self, state, action):
        if state not in self.q_values:
            self.q_values[state] = np.zeros(2)
        return self.q_values[state][action]

    def choose_action(self, state):
        if np.random.random() < self.epsilon:
            return np.random.randint(2)
        else:
            return np.argmax(self.q_values.get(state, np.zeros(2)))

    def update(self, state, action, reward, next_state):
        current_q = self.get_q_value(state, action)
        next_max_q = np.max(self.q_values.get(next_state, np.zeros(2)))

        # Q-learning update rule
        new_q = current_q + self.learning_rate * (reward + self.gamma * n

        if state not in self.q_values:
```

```
            self.q_values[state] = np.zeros(2)
        self.q_values[state][action] = new_q

    def train(self, episodes=1000, max_steps=100):
        for _ in range(episodes):
            # Reset state for new episode
            alpha1, beta1 = self.alpha1, self.beta1
            alpha2, beta2 = self.alpha2, self.beta2

            for step in range(max_steps):
                state = self.get_state_key(alpha1, beta1, alpha2, beta2)
                action = self.choose_action(state)

                # Generate reward based on Beta distribution
                if action == 0:
                    success_prob = alpha1 / (alpha1 + beta1)
                    reward = 1 if np.random.random() < success_prob else
                    if reward:
                        alpha1 += 1
                    else:
                        beta1 += 1
                else:
                    success_prob = alpha2 / (alpha2 + beta2)
                    reward = 1 if np.random.random() < success_prob else
                    if reward:
                        alpha2 += 1
                    else:
                        beta2 += 1

                reward = reward * (self.gamma ** step)
                next_state = self.get_state_key(alpha1, beta1, alpha2, be

                self.update(state, action, reward, next_state)

# Test the Q-learning implementation
ql = BayesianBanditQLearning(alpha1=1, beta1=1, alpha2=1, beta2=1, gamma=
ql.train()

# Get optimal policy for initial state
initial_state = ql.get_state_key(1, 1, 1, 1)
optimal_action = np.argmax(ql.q_values.get(initial_state, np.zeros(2)))
print(f"Optimal action for initial state: Arm {optimal_action + 1}")
print(f"Q-values for initial state: {ql.q_values.get(initial_state, np.ze
```

```
Optimal action for initial state: Arm 1
Q-values for initial state: [2.29027634 1.20295999]
```

5 The optimal policy:

In [12]:
```
def optimal_policy(N, gamma, true_theta, alpha, beta, max_exploration = 1
    rewards = np.zeros(N)
    alpha = np.array(alpha)
    beta = np.array(beta)
    calculate_R(alpha[0], beta[0], alpha[1], beta[1], gamma, 0, max_explo

    for t in range(N):
        chosen_arm = 0 if policy[(alpha[0], beta[0], alpha[1], beta[1])]
        reward = np.random.rand() < true_theta[chosen_arm]
        rewards[t] = reward * (gamma ** t)
        alpha[chosen_arm] += reward
```

```
        beta[chosen_arm] += 1 - reward

    return np.sum(rewards)
```

Let's test the optimal policy and compare its performance with the intuitive policy.

In [137…

```
test_runs = 1000
test_rewards_intuitive = np.zeros(test_runs)
test_rewards_optimal = np.zeros(test_runs)
N_short = 100
gamma_close_to_1 = 0.99

counter_alpha = [1, 1]
counter_beta = [1, 1]
# Different scenarios of true probabilities
counter_true_thetas = [
    np.array([0.85, 0.9]),   # Case 1: Second arm slightly better
    np.array([0.6, 0.8]),    # Case 2: Second arm significantly better
    np.array([0.95, 0.85]),  # Case 3: First arm better
    np.array([0.5, 0.55])    # Case 4: Close probabilities
]

for scenario_idx, counter_true_theta in enumerate(counter_true_thetas, 1)
    print(f"\nScenario {scenario_idx}: True probabilities = {counter_true

    for i in range(test_runs):
        local_alpha = counter_alpha.copy()
        local_beta = counter_beta.copy()
        test_rewards_intuitive[i] = intuitive_policy(N_short, gamma_close
                                                     counter_true_theta, lo

    for i in range(test_runs):
        local_alpha = counter_alpha.copy()
        local_beta = counter_beta.copy()
        test_rewards_optimal[i] = optimal_policy(N_short, gamma_close_to_
                                                 counter_true_theta, local_

    avg_r_intuitive = np.mean(test_rewards_intuitive)
    avg_r_optimal = np.mean(test_rewards_optimal)
    optimal_r = np.max(counter_true_theta) / (1 - gamma_close_to_1)
    regret_rate_intuitive = 1 - avg_r_intuitive/optimal_r
    regret_rate_optimal = 1 - avg_r_optimal/optimal_r

    print(f"Intuitive Average reward: {avg_r_intuitive:.2f}")
    print(f"Optimal Average reward: {avg_r_optimal:.2f}")
    print(f"Regret Rate (Intuitive): {regret_rate_intuitive:.4f}")
    print(f"Regret Rate (Optimal): {regret_rate_optimal:.4f}")
    print(f"Improvement: {((avg_r_optimal - avg_r_intuitive)/avg_r_intuit
```

```
Scenario 1: True probabilities = [0.85 0.9 ]
Intuitive Average reward: 54.39
Optimal Average reward: 56.20
Regret Rate (Intuitive): 0.3957
Regret Rate (Optimal): 0.3755
Improvement: 3.33%

Scenario 2: True probabilities = [0.6 0.8]
Intuitive Average reward: 44.74
Optimal Average reward: 48.76
Regret Rate (Intuitive): 0.4407
Regret Rate (Optimal): 0.3906
Improvement: 8.96%

Scenario 3: True probabilities = [0.95 0.85]
Intuitive Average reward: 60.04
Optimal Average reward: 57.71
Regret Rate (Intuitive): 0.3680
Regret Rate (Optimal): 0.3926
Improvement: -3.88%

Scenario 4: True probabilities = [0.5  0.55]
Intuitive Average reward: 33.42
Optimal Average reward: 33.68
Regret Rate (Intuitive): 0.3924
Regret Rate (Optimal): 0.3877
Improvement: 0.78%
```

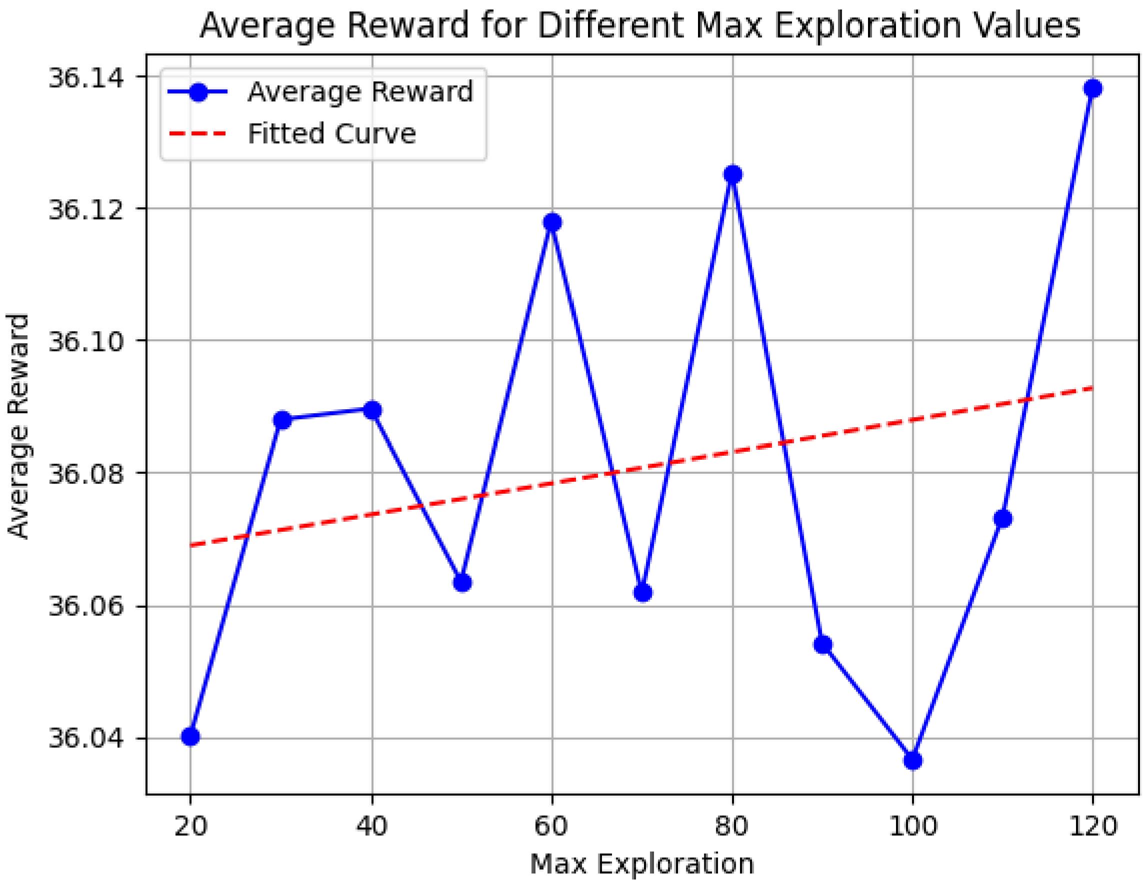
Based on the results presented above, we can conclude that the optimal policy significantly enhances performance

Further investigation: Let's adjust the hyperparameter 'max_exploration' to control the number of times we explore rather than exploit.

In [141…

```python
import numpy as np
import matplotlib.pyplot as plt

test_runs = 10000
test_rewards_optimal = np.zeros(test_runs)
N_short = 100
gamma_close_to_1 = 0.99

counter_alpha = [1, 1]
counter_beta = [1, 1]
counter_true_theta = np.array([0.5, 0.6])

avg_rewards_per_exploration = []

explore = [i for i in range(20, 121, 10)]

for max_exploration in explore:
    total_rewards = []

    for i in range(test_runs):
        local_alpha = counter_alpha.copy()
        local_beta = counter_beta.copy()
        test_rewards_optimal[i] = optimal_policy(N_short, gamma_close_to_

    avg_r_optimal = np.mean(test_rewards_optimal)
```

```
    avg_rewards_per_exploration.append(avg_r_optimal)

plt.plot(explore, avg_rewards_per_exploration, marker='o', linestyle='-',
plt.xlabel('Max Exploration')
plt.ylabel('Average Reward')
plt.title('Average Reward for Different Max Exploration Values')
plt.grid(True)

# 数据拟合成一条曲线
z = np.polyfit(explore, avg_rewards_per_exploration, 2)
p = np.poly1d(z)

plt.plot(explore, p(explore), 'r--')
plt.legend(['Average Reward', 'Fitted Curve'])


plt.show()
```



It appears that increasing exploration leads to better results. Here's our reasoning: We can interpret the equations in Problem 3 as Bellman equations in dynamic programming, except that they lack a base case. The base case actually exists when $\alpha, \beta \to \infty$. To approximate this base case, we use a sufficiently large number. Consequently, the larger the number, the closer it is to infinity, and the solution becomes more accurate.