In [4]:
```
import matplotlib.pyplot as plt
import numpy as np
import random, math, copy
```

In [5]:
```
# Set random seed for reproducibility
np.random.seed(42)

# Parameters
num_arms = 3

# Oracle theta of each arm
theta = np.array([0.7, 0.5, 0.4])
```

Problem 2: Implement classical bandit algorithms

1. The epsilon-greedy Algorithm

In [80]:
```
def epsilon_greedy(epsilon, N, theta):
    """
    Implement the epsilon-greedy algorithm for a Bernoulli bandit problem.

    Parameters
    ----------
    epsilon : float
        The probability of exploration.
    N : int
        Number of time steps.
    theta : array-like
        True success probabilities of each arm.

    Returns
    -------
    total_reward : float
        Total reward accumulated over N time steps.
    """
    Q = np.zeros(num_arms)  # Estimated values for each arm
    counts = np.zeros(num_arms)  # Count of how many times each arm is pulled
    total_reward = 0  # Total reward tracker

    # Initialization: Pull each arm once
    for arm in range(num_arms):
        reward = 1 if np.random.rand() < theta[arm] else 0
        counts[arm] = 1
        Q[arm] = reward
        total_reward += reward

    # Main loop: Epsilon-greedy exploration and exploitation
    for t in range(num_arms, N):
        if np.random.rand() < epsilon:
```

```
            # Exploration: choose a random arm
            arm = np.random.randint(num_arms)
        else:
            # Exploitation: choose the arm with the highest estimated value
            arm = np.argmax(Q)

        # Simulate pulling the chosen arm
        reward = 1 if np.random.rand() < theta[arm] else 0

        counts[arm] += 1
        Q[arm] += (1 / counts[arm]) * (reward - Q[arm])

        total_reward += reward

    return total_reward
```

2. The UCB (Upper Confidence Bound) Algorithm

In [81]:
```
def ucb(c, N, theta):
    """
    Implement the UCB (Upper Confidence Bound) algorithm for a Bernoulli bandit pro

    Parameters
    ----------
    c : float
        Confidence level parameter for the UCB algorithm.
    N : int
        Number of time steps.
    theta : array-like
        True success probabilities of each arm.

    Returns
    -------
    rewards_history : array
        The rewards obtained at each time step.
    """

    Q = np.zeros(num_arms)
    counts = np.zeros(num_arms)
    total_reward = 0

    # Initialize by pulling each arm once
    for arm in range(num_arms):
        reward = 1 if np.random.rand() < theta[arm] else 0
        Q[arm] = reward
        counts[arm] = 1
        total_reward += reward

    for t in range(num_arms+1, N+1):
        # Avoid division by zero because each arm was pulled once
        ucb_values = Q + c * np.sqrt((2*np.log(t))/counts)
        arm = np.argmax(ucb_values)
        reward = 1 if np.random.rand() < theta[arm] else 0
        counts[arm] += 1
```

```
        Q[arm] += (1/counts[arm])*(reward - Q[arm])
        total_reward += reward
    return total_reward
```

3. TS (Thompson Sampling) Algorithm

In [82]:

```
from scipy.stats import beta

def thompson_sampling(N, theta, alpha_init, beta_init):
    """
    Implement the Thompson Sampling (TS) algorithm for a Bernoulli bandit problem.

    Parameters
    ----------
    N : int
        Number of time steps.
    theta : array-like
        True success probabilities of each arm.
    alpha_init : array-like
        Initial alpha parameters for the Beta distributions of each arm.
    beta_init : array-like
        Initial beta parameters for the Beta distributions of each arm.

    Returns
    -------
    rewards_history : array
        The rewards obtained at each time step.
    """
    alpha = alpha_init.copy()
    beta_ = beta_init.copy()
    total_reward = 0
    for t in range(N):
        sampled_thetas = [np.random.beta(alpha[j], beta_[j]) for j in range(num_arm
        arm = np.argmax(sampled_thetas)
        reward = 1 if np.random.rand() < theta[arm] else 0
        total_reward += reward
        alpha[arm] += reward
        beta_[arm] += 1 - reward
    return total_reward
```

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 [83]:

```
# Parameters
N = 5000
num_trials = 200
epsilons = [0.1, 0.5, 0.9]
cs = [1, 5, 10]
```

```
# Two sets of prior parameters for TS
# Set 1: (1,1), (1,1), (1,1)
alpha_set_1 = np.array([1, 1, 1])
beta_set_1 = np.array([1, 1, 1])

# Set 2: (601,401), (401,601), (2,3)
alpha_set_2 = np.array([601, 401, 2])
beta_set_2 = np.array([401, 601, 3])

# True parameters of the arms (as per the oracle, but not known to the algorithm)
theta = np.array([0.7, 0.5, 0.4])
```

In [84]:

```
# Epsilon-greedy
print("Epsilon-greedy results:")
for eps in epsilons:
    rewards = []
    for _ in range(num_trials):
        rewards.append(epsilon_greedy(eps, N, theta))
    mean_reward = np.mean(rewards)
    print(f"Epsilon = {eps}, Average total reward over {num_trials} trials: {mean_r
```

```
Epsilon-greedy results:
Epsilon = 0.1, Average total reward over 200 trials: 3408.44
Epsilon = 0.5, Average total reward over 200 trials: 3085.66
Epsilon = 0.9, Average total reward over 200 trials: 2748.215
```

In [85]:

```
# UCB
print("\nUCB results:")
for c_val in cs:
    rewards = []
    for _ in range(num_trials):
        rewards.append(ucb(c_val, N, theta))
    mean_reward = np.mean(rewards)
    print(f"c = {c_val}, Average total reward over {num_trials} trials: {mean_rewar
```

```
UCB results:
c = 1, Average total reward over 200 trials: 3408.315
c = 5, Average total reward over 200 trials: 2979.74
c = 10, Average total reward over 200 trials: 2829.24
```

In [86]:

```
# Thompson Sampling
print("\nThompson Sampling results:")
rewards_set_1 = []
for _ in range(num_trials):
    rewards_set_1.append(thompson_sampling(N, theta, alpha_set_1, beta_set_1))
mean_set_1 = np.mean(rewards_set_1)
print(f"Set 1 Priors (1,1),(1,1),(1,1), Average total reward: {mean_set_1}")

rewards_set_2 = []
for _ in range(num_trials):
    rewards_set_2.append(thompson_sampling(N, theta, alpha_set_2, beta_set_2))
mean_set_2 = np.mean(rewards_set_2)
print(f"Set 2 Priors (601,401),(401,601),(2,3), Average total reward: {mean_set_2}"
```

```
Thompson Sampling results:
Set 1 Priors (1,1),(1,1),(1,1), Average total reward: 3480.75
Set 2 Priors (601,401),(401,601),(2,3), Average total reward: 3492.41
```

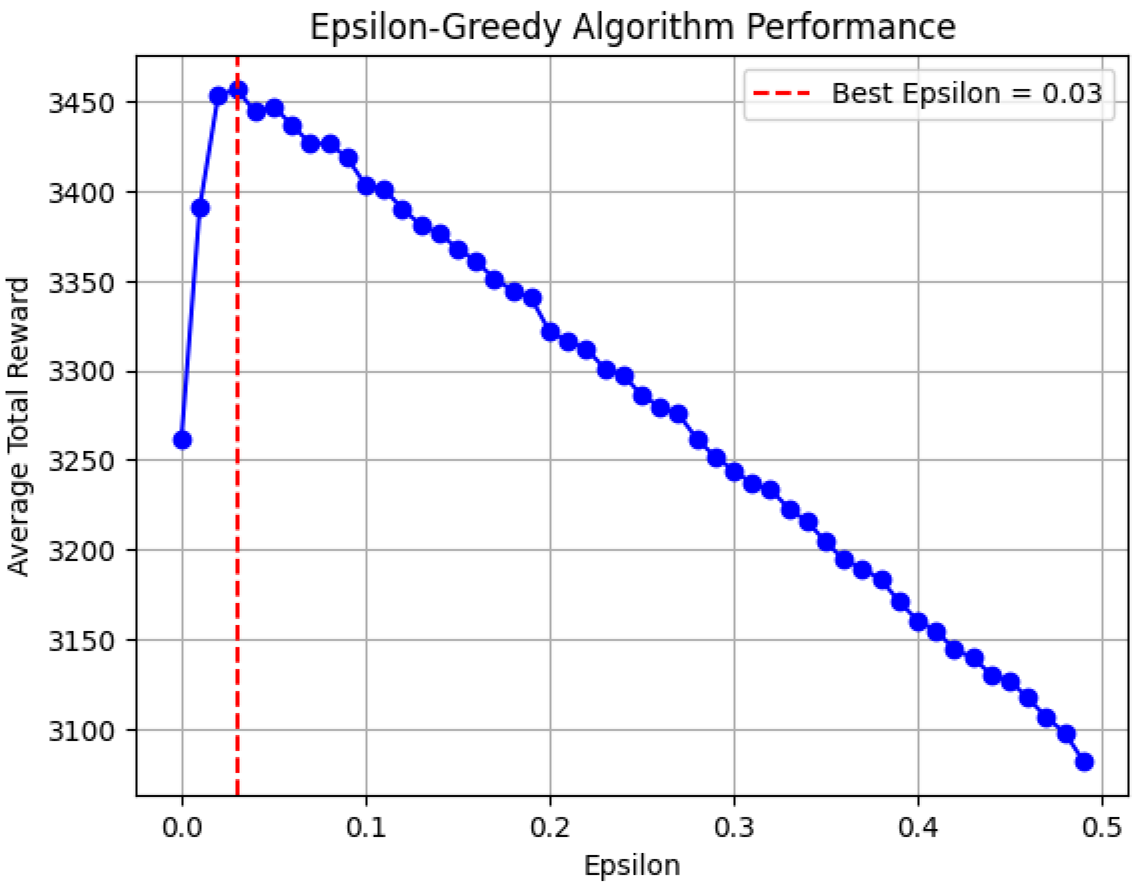
Problem 4

4.1 Find the optimal results for each algorithm

In [94]:

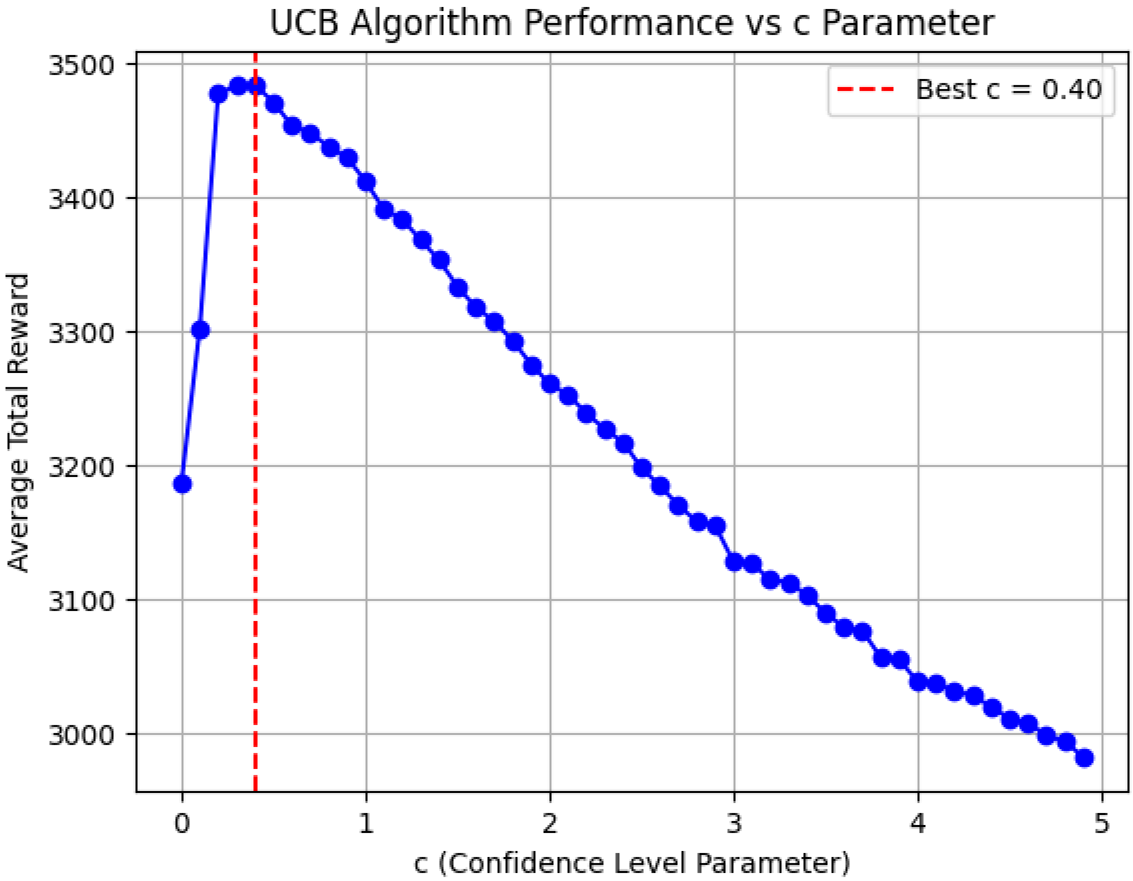
```python
num_trials = 100

epsilon_values = np.arange(0, 0.5, 0.01)
average_rewards = []

for eps in epsilon_values:
    rewards = []
    for _ in range(num_trials):
        rewards.append(epsilon_greedy(eps, N, theta))
    average_rewards.append(np.mean(rewards))

# Find the best epsilon
best_epsilon = epsilon_values[np.argmax(average_rewards)]
print(f"Best epsilon: {best_epsilon:.2f}")
print(f"Maximum average total reward: {np.max(average_rewards):.2f}")

# Plot the results
import matplotlib.pyplot as plt

plt.plot(epsilon_values, average_rewards, marker='o', linestyle='-', color = 'b')
plt.axvline(x=best_epsilon, color='r', linestyle='--', label=f'Best Epsilon = {best
plt.xlabel('Epsilon')
plt.ylabel('Average Total Reward')
plt.title('Epsilon-Greedy Algorithm Performance')
plt.legend()
plt.grid(True)
plt.show()
```

```
Best epsilon: 0.03
Maximum average total reward: 3457.02
```



In [93]:
```python
c_values = np.arange(0, 5, 0.1)
average_rewards = []

# Run UCB for each value of c and compute the average reward over multiple trials
for c in c_values:
    rewards = []
    for _ in range(num_trials):
        total_reward = ucb(c, N, theta)
        rewards.append(total_reward)
    average_rewards.append(np.mean(rewards))

# Identify the best c
best_c_index = np.argmax(average_rewards)
best_c = c_values[best_c_index]
best_average_reward = average_rewards[best_c_index]

print(f"Best c value: {best_c:.2f}")
print(f"Maximum average total reward: {best_average_reward:.2f}")

# Plot the results
plt.plot(c_values, average_rewards, marker='o', linestyle='-', color='b')
plt.axvline(x=best_c, color='r', linestyle='--', label=f'Best c = {best_c:.2f}')
plt.xlabel('c (Confidence Level Parameter)')
plt.ylabel('Average Total Reward')
plt.title('UCB Algorithm Performance vs c Parameter')
plt.legend()
plt.grid(True)
plt.show()
```

```
Best c value: 0.40
Maximum average total reward: 3483.90
```



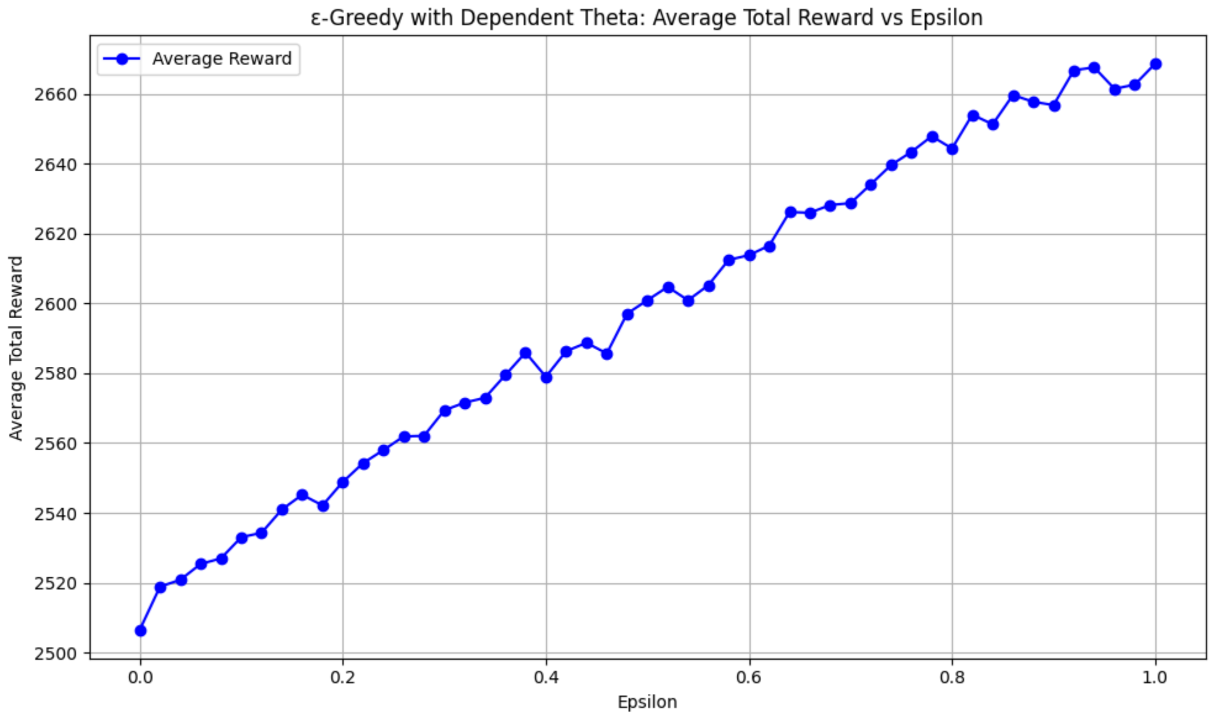
Problem 6

In [2]:
```python
import numpy as np
import random
import matplotlib.pyplot as plt
from itertools import product
import seaborn as sns
import pandas as pd

num_arms = 3
```

In [2]:
```python
# Initialize global variables for counts and estimated thetas
count = [0, 0, 0]  # Corresponds to Arm 1, Arm 2, Arm 3
theta = [0.0, 0.0, 0.0]  # Estimated thetas for Arm 1, Arm 2, Arm 3

def init_greedy():
    """
    Initializes the counts and estimated thetas for the greedy algorithm.
    """
    global count, theta
    count = [0, 0, 0]  # Reset counts for Arms 1, 2, 3
    theta = [0.0, 0.0, 0.0]  # Reset estimated thetas

def greedy_dependence(n, epsilon, initial_theta_oracled, p):
```

```
    """
    Greedy algorithm with dependency in theta_oracled.

    Parameters:
    - n: Number of time steps
    - epsilon: Exploration rate
    - initial_theta_oracled: Initial probabilities for each arm [θ1, θ2, θ3]
    - p: Probability adjustment parameter
    """
    global count, theta
    init_greedy()  # Initialize counts and estimates
    total_reward = 0  # Total actual rewards obtained

    # Deep copy to avoid modifying the original initial_theta_oracled
    current_theta = initial_theta_oracled.copy()

    for t in range(n):
        prob = random.random()  # Generate a random number in [0,1)

        if prob < epsilon:
            # Explore: choose a random arm from {0,1,2} corresponding to Arm 1, 2,
            arm = random.randint(0, 2)
        else:
            # Exploit: choose the arm with the highest estimated theta
            arm = np.argmax(theta)
            if theta[arm] == 0:
                # If all estimated thetas are 0, choose a random arm
                arm = random.randint(0, 2)

        # Simulate pulling the chosen arm: reward is 1 with probability current_the
        r_i = np.random.binomial(1, current_theta[arm])

        # Accumulate the actual reward
        total_reward += r_i

        # Update counts and estimated thetas using incremental averaging
        count[arm] += 1
        theta[arm] += (r_i - theta[arm]) / count[arm]

        # Update theta_oracled based on the outcome
        if r_i == 1:
            # If reward obtained, decrease theta of pulled arm and increase others
            current_theta[arm] = max(current_theta[arm] - p, 0.0)
            for other_arm in range(3):
                if other_arm != arm:
                    current_theta[other_arm] = min(current_theta[other_arm] + p / 2
        else:
            # If no reward, increase theta of pulled arm and decrease others
            current_theta[arm] = min(current_theta[arm] + p, 1.0)
            for other_arm in range(3):
                if other_arm != arm:
                    current_theta[other_arm] = max(current_theta[other_arm] - p / 2

    return total_reward

# Define the initial true reward probabilities (unknown to the algorithm)
```

```
initial_theta_oracled = [0.7, 0.5, 0.4]  # ϑ1=0.7, ϑ2=0.5, ϑ3=0.4

# Experiment Parameters
epsilon_values = np.arange(0, 1.02, 0.02)  # Epsilon from 0 to 1 in steps of 0.02
repeat_time = 100  # Number of trials for each epsilon
N = 5000  # Number of time steps per trial
p = 0.005  # Probability adjustment parameter

rewards = np.zeros(len(epsilon_values))  # Average rewards for each epsilon

# Run experiments for each epsilon
for i, eps in enumerate(epsilon_values):
    for trial in range(repeat_time):
        # For each trial, reset the initial theta_oracled
        theta_oracled = initial_theta_oracled.copy()
        reward = greedy_dependence(N, eps, theta_oracled, p)
        rewards[i] += reward / repeat_time

# Plot the results
plt.figure(figsize=(10, 6))

# Plot Average Total Reward vs. Epsilon
plt.plot(epsilon_values, rewards, marker='o', linestyle='-', color='blue', label='A
plt.scatter(epsilon_values, rewards, color='red', s=10)
plt.xlabel('Epsilon')
plt.ylabel('Average Total Reward')
plt.title('ε-Greedy with Dependent Theta: Average Total Reward vs Epsilon')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()

# Identify and print the best epsilon based on rewards
best_index = np.argmax(rewards)
best_epsilon = epsilon_values[best_index]
best_reward = rewards[best_index]
print(f"Best epsilon: {best_epsilon:.2f}")
print(f"Maximum average total reward: {best_reward:.2f}")
```

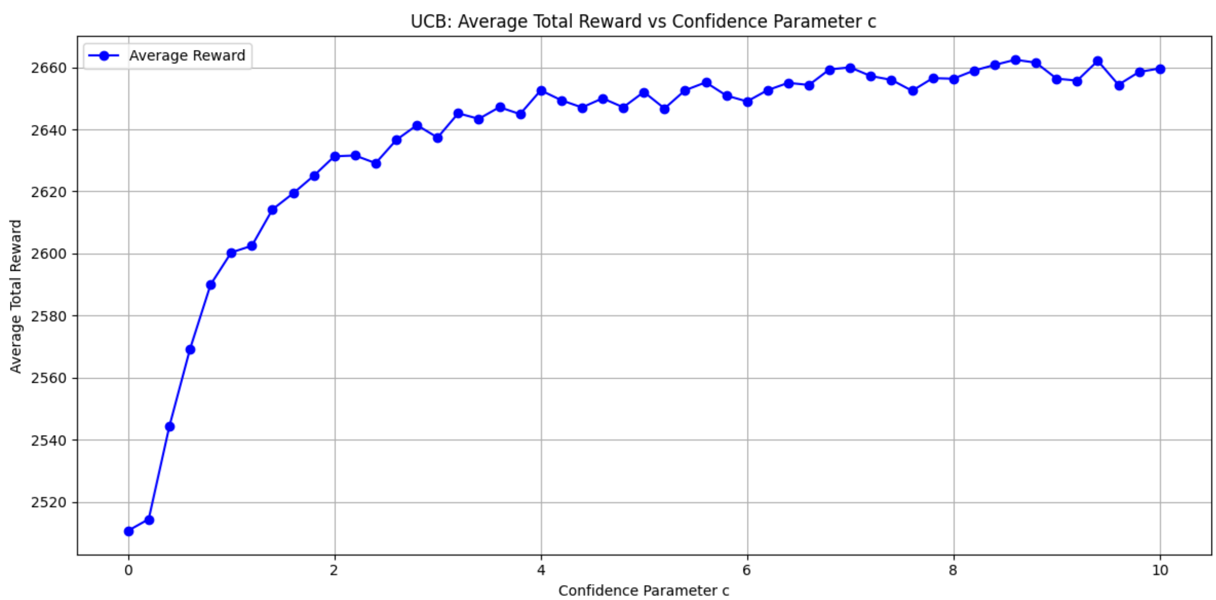



```
Best epsilon: 1.00
Maximum average total reward: 2668.57
```

In [3]:
```
def ucb_dependence(c, N, initial_theta_oracled, p=0.005):
    """
    UCB algorithm with independent arms.

    Parameters:
    - c: Confidence parameter for UCB
    - N: Number of time steps
    - initial_theta_oracled: List of initial true reward probabilities for each arm
    - p: Probability adjustment parameter
    """
    num_arms = 3
    Q = np.zeros(num_arms)          # Estimated rewards for each arm
    counts = np.zeros(num_arms)     # Number of times each arm has been pulled
    total_reward = 0                 # Total accumulated reward

    # Deep copy to avoid modifying the original initial_theta_oracled
    theta_oracled = initial_theta_oracled.copy()

    # Initialize by pulling each arm once
    for arm in range(num_arms):
        reward = 1 if random.random() < theta_oracled[arm] else 0
        Q[arm] = reward
        counts[arm] = 1
        total_reward += reward

        # Update theta_oracled based on the outcome
        if reward == 1:
            # If reward obtained, decrease the probability of the pulled arm and in
            theta_oracled[arm] = max(theta_oracled[arm] - p, 0.0)
            for other_arm in range(num_arms):
                if other_arm != arm:
```

```
                theta_oracled[other_arm] = min(theta_oracled[other_arm] + p / 2
        else:
            # If no reward, increase the probability of the pulled arm and decrease
            theta_oracled[arm] = min(theta_oracled[arm] + p, 1.0)
            for other_arm in range(num_arms):
                if other_arm != arm:
                    theta_oracled[other_arm] = max(theta_oracled[other_arm] - p / 2

    # Run UCB algorithm for remaining time steps
    for t in range(num_arms, N):
        # Compute UCB values for each arm
        ucb_values = Q + c * np.sqrt((2 * np.log(t + 1)) / counts)
        arm = np.argmax(ucb_values)

        # Pull the selected arm and observe the reward
        reward = 1 if random.random() < theta_oracled[arm] else 0
        total_reward += reward

        # Update counts and estimated rewards
        counts[arm] += 1
        Q[arm] += (reward - Q[arm]) / counts[arm]

        # Update theta_oracled based on the outcome
        if reward == 1:
            # If reward obtained, decrease the probability of the pulled arm and in
            theta_oracled[arm] = max(theta_oracled[arm] - p, 0.0)
            for other_arm in range(num_arms):
                if other_arm != arm:
                    theta_oracled[other_arm] = min(theta_oracled[other_arm] + p / 2
        else:
            # If no reward, increase the probability of the pulled arm and decrease
            theta_oracled[arm] = min(theta_oracled[arm] + p, 1.0)
            for other_arm in range(num_arms):
                if other_arm != arm:
                    theta_oracled[other_arm] = max(theta_oracled[other_arm] - p / 2

    return total_reward

# Define the initial true reward probabilities (unknown to the algorithm)
initial_theta_oracled = [0.7, 0.5, 0.4]  # [ϑ1, ϑ2, ϑ3]

# Experiment Parameters
c_values = np.arange(0.0, 10.2, 0.2)  # Confidence parameter c from 0 to 10 in step
repeat_time = 100  # Number of trials for each c
N = 5000  # Number of time steps per trial
p = 0.005  # Probability adjustment parameter

average_rewards = np.zeros(len(c_values))  # Average rewards for each c

# Run experiments for each c
for i, c in enumerate(c_values):
    for trial in range(repeat_time):
        # For each trial, reset the initial_theta_oracled
        theta_oracled = initial_theta_oracled.copy()
        reward = ucb_dependence(c, N, theta_oracled, p)
        average_rewards[i] += reward / repeat_time
```

```
# Plot the results
plt.figure(figsize=(12, 6))

# Plot Average Total Reward vs. Confidence Parameter c
plt.plot(c_values, average_rewards, marker='o', linestyle='-', color='blue', label=
plt.scatter(c_values, average_rewards, color='red', s=10)
plt.xlabel('Confidence Parameter c')
plt.ylabel('Average Total Reward')
plt.title('UCB: Average Total Reward vs Confidence Parameter c')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()

# Identify and print the best c based on rewards
best_index = np.argmax(average_rewards)
best_c = c_values[best_index]
best_reward = average_rewards[best_index]
print(f"Best c: {best_c:.2f}")
print(f"Maximum average total reward: {best_reward:.2f}")
```




```
Best c: 8.60
Maximum average total reward: 2662.44
```

In [3]:
```
def thompson_sampling_dependence(N, theta_oracled, alpha_init, beta_init, p=0.005):
    num_arms = 3
    alpha = alpha_init.copy()
    beta = beta_init.copy()
    total_reward = 0.0

    # Deep copy to avoid modifying the original theta_oracled
    theta_current = theta_oracled.copy()

    for t in range(N):
        # Sample theta from Beta distributions for each arm
        sampled_thetas = [np.random.beta(alpha[j], beta[j]) for j in range(num_arms
```

```
        # Select the arm with the highest sampled theta
        arm = np.argmax(sampled_thetas)

        # Simulate pulling the selected arm: reward is 1 with probability theta_cur
        reward = 1 if np.random.random() < theta_current[arm] else 0
        total_reward += reward

        # Update the Beta distribution parameters for the selected arm
        alpha[arm] += reward
        beta[arm] += (1 - reward)

        # Update theta_oracled based on the outcome
        if reward == 1:
            # If reward obtained, decrease theta of pulled arm and increase others
            theta_current[arm] = max(theta_current[arm] - p, 0.0)
            for other_arm in range(num_arms):
                if other_arm != arm:
                    theta_current[other_arm] = min(theta_current[other_arm] + p / 2
        else:
            # If no reward, increase theta of pulled arm and decrease others
            theta_current[arm] = min(theta_current[arm] + p, 1.0)
            for other_arm in range(num_arms):
                if other_arm != arm:
                    theta_current[other_arm] = max(theta_current[other_arm] - p / 2

    return total_reward
```

In [4]:

```
def dependency_aware_thompson_sampling(N, theta, alpha_init, beta_init, p=0.005, ep
    alpha = alpha_init.copy()
    beta  = beta_init.copy()
    K = len(alpha)
    theta_current = theta.copy()
    total_reward = 0

    for t in range(N):
        # Epsilon-greedy: with prob epsilon, pick a random arm
        if np.random.rand() < epsilon:
            chosen_arm = np.random.choice(K)
        else:
            # Otherwise, Thompson sample from each arm's Beta posterior
            samples = [np.random.beta(alpha[i], beta[i]) for i in range(K)]
            chosen_arm = np.argmax(samples)

        # Observe reward from environment
        reward = (np.random.rand() < theta_current[chosen_arm])
        total_reward += reward

        # --- Update Beta posterior for chosen arm and other arms ---
        if reward:
            # Chosen arm gets a standard Beta update
            alpha[chosen_arm] += 1

            # If gamma > 0, nudge alpha of the other arms
            for other_arm in range(K):
                if other_arm != chosen_arm:
```

```
                alpha[other_arm] += gamma
        else:
            # Chosen arm gets a standard Beta update
            beta[chosen_arm] += 1

            # If gamma > 0, nudge beta of the other arms
            for other_arm in range(K):
                if other_arm != chosen_arm:
                    beta[other_arm] += gamma

        # environment update
        if reward:
            theta_current[chosen_arm] = max(theta_current[chosen_arm] - p, 0.0)
            for other_arm in range(K):
                if other_arm != chosen_arm:
                    theta_current[other_arm] = min(theta_current[other_arm] + p/2,
        else:
            theta_current[chosen_arm] = min(theta_current[chosen_arm] + p, 1.0)
            for other_arm in range(K):
                if other_arm != chosen_arm:
                    theta_current[other_arm] = max(theta_current[other_arm] - p/2,

    return total_reward
```

In [6]:

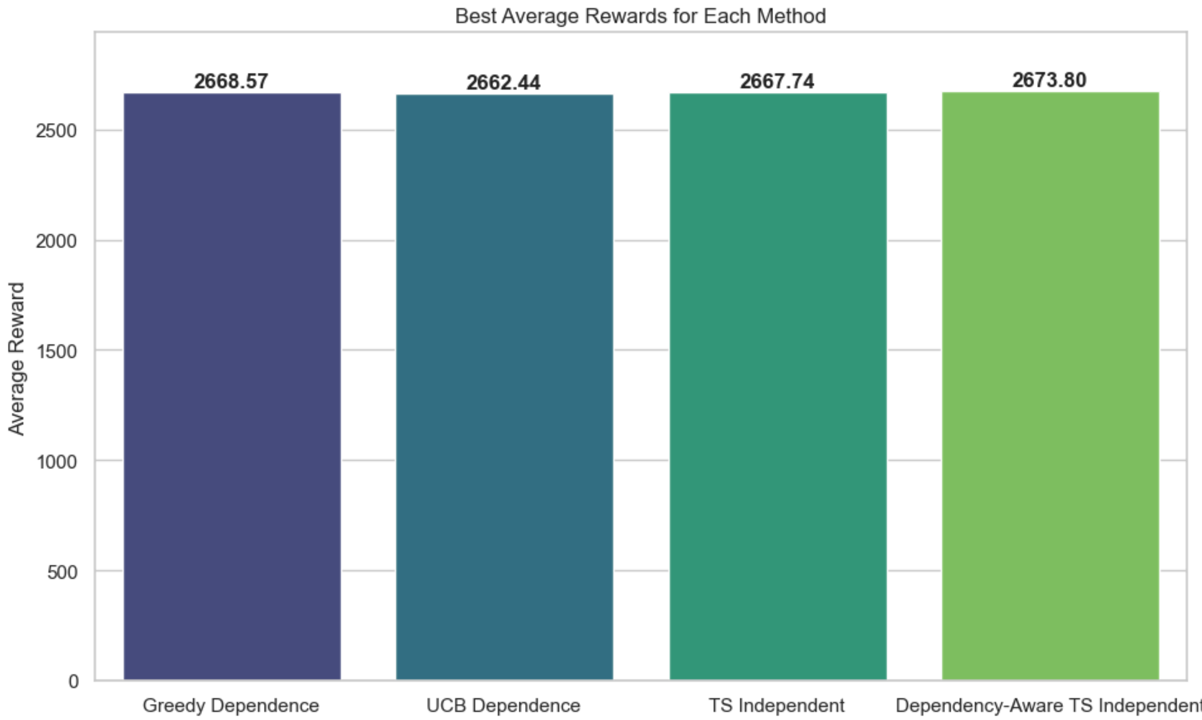
```
# Define the true reward probabilities (independent arms)
theta1_true = 0.5
theta2_true = 0.4
theta3_true = 0.7
theta = [theta3_true, theta1_true, theta2_true]  # [ϑ1, ϑ2, ϑ3] = [0.7, 0.5, 0.4]

# ---------- Parameter Ranges for Three Arms ----------
alpha1_values = [450, 600]
beta1_values  = [1, 16]
alpha2_values = [300, 450]
beta2_values  = [1, 16]
alpha3_values = [150, 300, 450]
beta3_values  = [16, 31]

N = 5000
repeat_time = 50

# ---------- Generate All Parameter Combinations for Three Arms ----------
parameter_combinations = list(product(
    alpha1_values, alpha2_values, alpha3_values,
    beta1_values, beta2_values, beta3_values
))

# ---------- Initialize Result Lists for Both Algorithms ----------
results_ts_independent = []  # For thompson_sampling_independent
results_da_ts_independent = []  # For dependency_aware_thompson_sampling (now indep

# ---------- Running Both Algorithms Across All Parameter Combinations ----------
for idx, (alpha1_val, alpha2_val, alpha3_val, beta1_val, beta2_val, beta3_val) in e
    alpha_init_ts = [alpha1_val, alpha2_val, alpha3_val]
    beta_init_ts = [beta1_val, beta2_val, beta3_val]
```

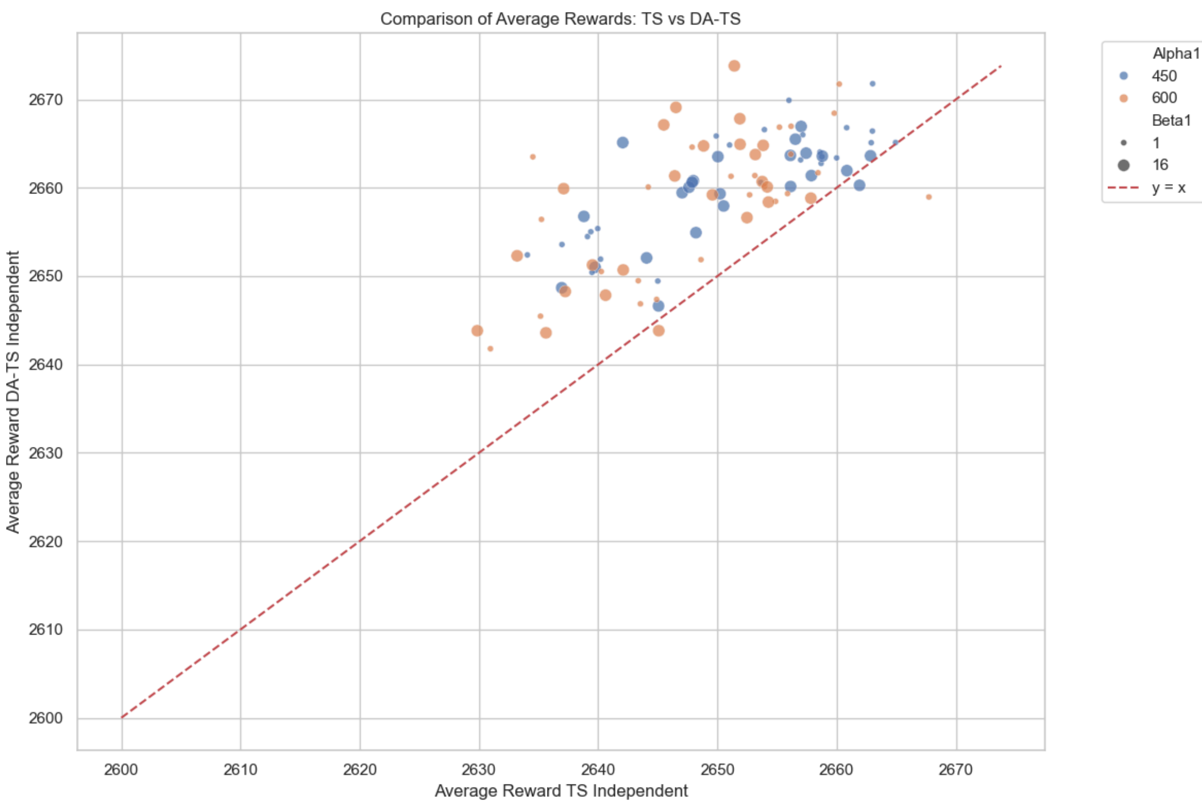
```
    alpha_init_da = [alpha1_val, alpha2_val, alpha3_val]
    beta_init_da = [beta1_val, beta2_val, beta3_val]

    # ---------- Run Trials for Thompson Sampling Independent ----------
    total_reward_ts = 0.0
    for _ in range(repeat_time):
        theta_oracled = [0.7, 0.5, 0.4]  # [ϑ1, ϑ2, ϑ3]
        reward = thompson_sampling_dependence(N, theta_oracled, alpha_init_ts, beta
        total_reward_ts += reward / repeat_time

    # ---------- Run Trials for Dependency-Aware Thompson Sampling ----------
    epsilon_values = [1e-2,7e-3, 5e-3, 3e-3, 1e-3]
    gamma_values = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2]

    best_reward_da = -np.inf
    best_epsilon = None
    best_gamma = None

    for epsilon in epsilon_values:
        for gamma in gamma_values:
            total_reward_da = 0.0
            for _ in range(repeat_time):
                # Reset theta_oracled for each trial
                theta_oracled = [theta3_true, theta1_true, theta2_true]  # [ϑ1, ϑ2,
                # Run DA-TS with current epsilon and gamma
                reward = dependency_aware_thompson_sampling(
                    N, theta_oracled, alpha_init_da, beta_init_da,
                    p=0.005, epsilon=epsilon, gamma=gamma
                )
                total_reward_da += reward / repeat_time

            # Check if this (epsilon, gamma) pair yields a better reward
            if total_reward_da > best_reward_da:
                best_reward_da = total_reward_da
                best_epsilon = epsilon
                best_gamma = gamma

    # ---------- Store Results ----------
    results_ts_independent.append({
        'Alpha1': alpha1_val,
        'Alpha2': alpha2_val,
        'Alpha3': alpha3_val,
        'Beta1': beta1_val,
        'Beta2': beta2_val,
        'Beta3': beta3_val,
        'Avg Reward TS': total_reward_ts
    })
    results_da_ts_independent.append({
        'Alpha1': alpha1_val,
        'Alpha2': alpha2_val,
        'Alpha3': alpha3_val,
        'Beta1': beta1_val,
        'Beta2': beta2_val,
        'Beta3': beta3_val,
        'Best Epsilon DA-TS': best_epsilon,
        'Best Gamma DA-TS': best_gamma,
```

```
        'Avg Reward DA-TS': best_reward_da
    })

# ---------- Convert Results to DataFrames ----------
df_ts = pd.DataFrame(results_ts_independent)
df_da = pd.DataFrame(results_da_ts_independent)

# ---------- Merge DataFrames for Easier Comparison ----------
df_combined = pd.merge(df_ts, df_da, on=['Alpha1', 'Alpha2', 'Alpha3', 'Beta1', 'Be

# ---------- Find Best Outcomes for Each Method ----------

best_avg_ts = df_ts['Avg Reward TS'].max()
best_avg_da = df_da['Avg Reward DA-TS'].max()

# ---------- Enhanced Printing ----------

print("===== Best Average Rewards =====")
print(f"Thompson Sampling Independent: {best_avg_ts:.2f}")
print(f"Dependency-Aware Thompson Sampling Independent: {best_avg_da:.2f}")

# ---------- List Top 5 Parameter Combinations for Each Method ----------

print("\n===== Top 5 Parameter Combinations for Thompson Sampling Independent =====
top5_ts = df_ts.sort_values(by='Avg Reward TS', ascending=False).head(5)
print(top5_ts.to_string(index=False))

print("\n===== Top 5 Parameter Combinations for Dependency-Aware Thompson Sampling 
top5_da = df_da.sort_values(by='Avg Reward DA-TS', ascending=False).head(5)
print(top5_da.to_string(index=False))

# ---------- Plotting ----------

# Set the style for seaborn
sns.set(style="whitegrid")

# 1. Bar Plot of Best Average Rewards
greedy_dependence_avg = 2668.57
ucb_dependence_avg = 2662.44
plt.figure(figsize=(10, 6))
methods = ['Greedy Dependence', 'UCB Dependence', 'TS Independent', 'Dependency-Awa
avg_rewards = [greedy_dependence_avg, ucb_dependence_avg, best_avg_ts, best_avg_da]
# Fix deprecated palette usage in barplot
sns.barplot(x=methods,
            y=avg_rewards,
            hue=methods,  # Assign x to hue
            legend=False, # Hide redundant legend
            palette="viridis")
plt.ylabel('Average Reward')
plt.title('Best Average Rewards for Each Method')
plt.ylim(0, max(avg_rewards)*1.1)
for i, v in enumerate(avg_rewards):
    plt.text(i, v + max(avg_rewards)*0.01, f"{v:.2f}", ha='center', fontweight='bol
plt.tight_layout()
plt.show()
```

```
# 2. Scatter Plot Comparing Both Methods with Reference Line
plt.figure(figsize=(12, 8))
scatter = sns.scatterplot(
    data=df_combined,
    x='Avg Reward TS',
    y='Avg Reward DA-TS',
    hue='Alpha1',
    size='Beta1',
    palette='deep',
    alpha=0.7
)
# Add reference line y = x
max_val = max(df_combined['Avg Reward TS'].max(), df_combined['Avg Reward DA-TS'].m
plt.plot([2600, max_val], [2600, max_val], 'r--', label='y = x')
plt.xlabel('Average Reward TS Independent')
plt.ylabel('Average Reward DA-TS Independent')
plt.title('Comparison of Average Rewards: TS vs DA-TS')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()

# 3. Optional: Save Plots
plt.savefig('pics/best_average_rewards.png')
plt.savefig('pics/comparison_scatter.png')
```

```
===== Best Average Rewards =====
Thompson Sampling Independent: 2667.74
Dependency-Aware Thompson Sampling Independent: 2673.80

===== Top 5 Parameter Combinations for Thompson Sampling Independent =====
 Alpha1  Alpha2  Alpha3  Beta1  Beta2  Beta3  Avg Reward TS
    600     300     450      1      1     31        2667.74
    450     450     300      1     16     16        2664.94
    450     450     450      1      1     31        2663.02
    450     450     450      1     16     31        2663.00
    450     300     450      1      1     31        2662.90

===== Top 5 Parameter Combinations for Dependency-Aware Thompson Sampling Independen
t =====
 Alpha1  Alpha2  Alpha3  Beta1  Beta2  Beta3  Best Epsilon DA-TS  Best Gamma DA-TS 
Avg Reward DA-TS
    600     450     450     16     16     31               0.001          0.000001 
2673.80
    450     450     450      1      1     31               0.007          0.000001 
2671.78
    600     450     450      1      1     31               0.007          0.000001 
2671.74
    450     300     300      1      1     31               0.010          0.010000 
2669.90
    600     450     300     16      1     16               0.007          0.000100 
2669.10
```






```
<Figure size 640x480 with 0 Axes>
```

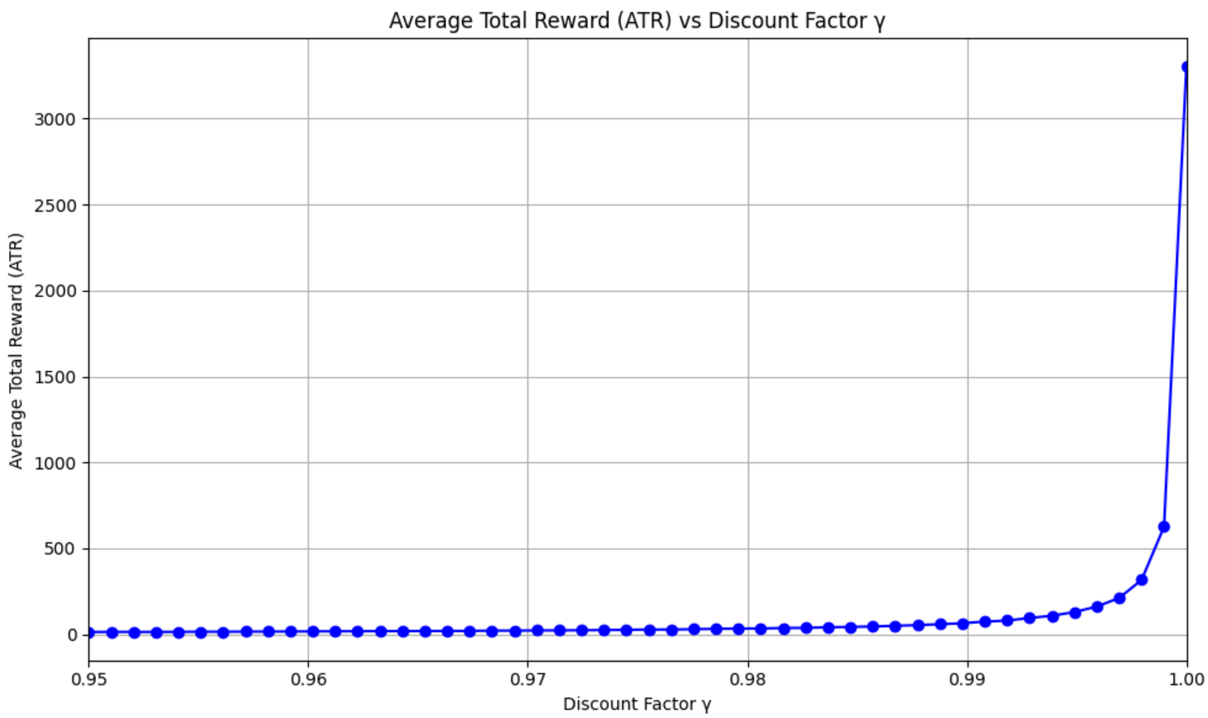
Part II

Problem 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 [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import beta

np.random.seed(42)
```

In [9]:
```
# Simulation parameters
true_theta = [0.7, 0.5]        # True success probabilities for arm 1 and arm 2
alpha_prior = [1, 1]           # Prior alpha parameters for Beta distributions
beta_prior = [1, 1]            # Prior beta parameters for Beta distributions
gamma_values = np.linspace(0.95, 1.0, 50)  # Gamma values from 0.5 to 1.0 in increm
time_steps = 5000              # Number of pulls per trial
repeat_time = 50               # Number of trials per gamma

# Initialize array to store average total rewards for each gamma
average_total_rewards = []

# Iterate over each gamma value
for gamma in gamma_values:
    total_rewards = []  # To store total rewards for each trial

    # Repeat the trial 'repeat_time' times for averaging
    for trial in range(repeat_time):
        # Initialize Beta parameters for each arm
        alpha = alpha_prior.copy()
        beta_params = beta_prior.copy()

        cumulative_reward = 0  # Total reward for this trial

        for t in range(1, time_steps + 1):
            # Calculate expected theta for each arm
            expected_theta = [alpha[i] / (alpha[i] + beta_params[i]) for i in range

            # Select the arm with the highest expected theta
            chosen_arm = np.argmax(expected_theta)

            # Simulate a pull: success with probability true_theta[chosen_arm]
            success = np.random.rand() < true_theta[chosen_arm]

            # Update Beta posterior
            if success:
                alpha[chosen_arm] += 1
                reward = gamma**(t-1)
            else:
                beta_params[chosen_arm] += 1
                reward = 0

            # Update cumulative reward
            cumulative_reward += reward

        total_rewards.append(cumulative_reward)

    # Calculate average total reward for this gamma
    avg_reward = np.mean(total_rewards)
    average_total_rewards.append(avg_reward)
```

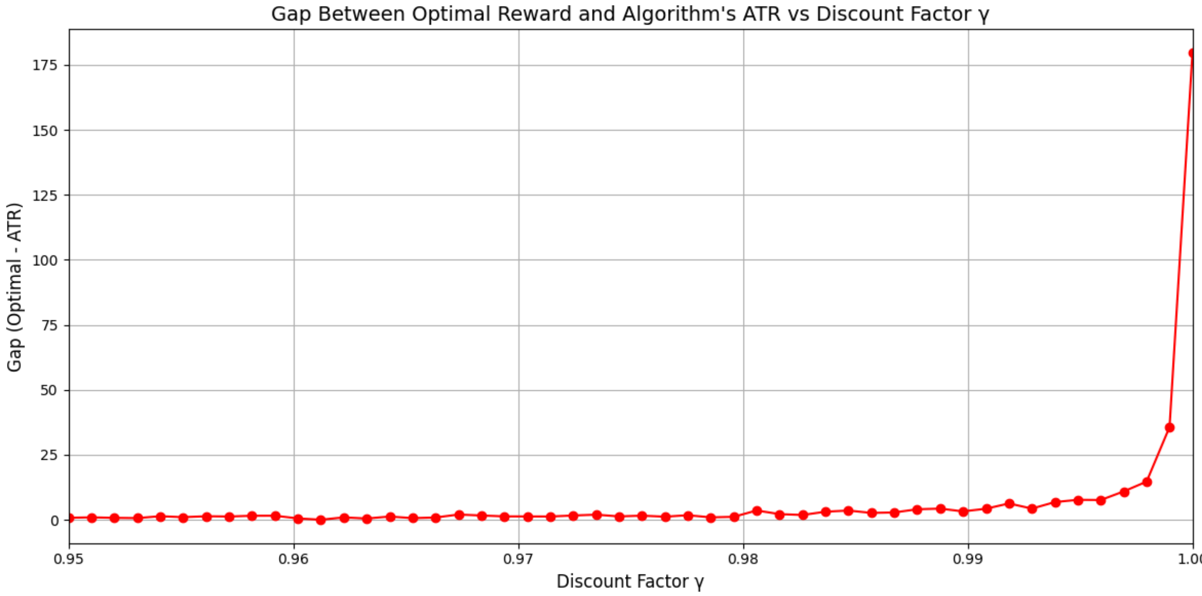
```
# Visualization: ATR vs Gamma
plt.figure(figsize=(10, 6))
plt.plot(gamma_values, average_total_rewards, marker='o', linestyle='-', color='blu
plt.title('Average Total Reward (ATR) vs Discount Factor γ')
plt.xlabel('Discount Factor γ')
plt.ylabel('Average Total Reward (ATR)')
plt.grid(True)
plt.xlim(0.95, 1.0)
plt.tight_layout()
plt.show()
```



In [10]:

```
# Identify the best arm (with the highest true_theta)
best_arm = np.argmax(true_theta)
theta_best = true_theta[best_arm]

# Initialize lists to store results
gamma_list = []
gap_list = []

# Iterate over each gamma value
for gamma in gamma_values:
    total_rewards = []  # To store total rewards for each trial

    # Repeat the trial 'repeat_time' times for averaging
    for trial in range(repeat_time):
        # Initialize Beta parameters for each arm
        alpha = alpha_prior.copy()
        beta_params = beta_prior.copy()

        cumulative_reward = 0.0  # Total reward for this trial

        for t in range(1, time_steps + 1):
            # Calculate expected theta for each arm using current Beta posterior
```

```
            expected_theta = [alpha[i] / (alpha[i] + beta_params[i]) for i in range

            # Select the arm with the highest expected theta
            chosen_arm = np.argmax(expected_theta)

            # Simulate a pull: success with probability true_theta[chosen_arm]
            success = np.random.rand() < true_theta[chosen_arm]

            # Update Beta posterior based on the outcome
            if success:
                alpha[chosen_arm] += 1
                reward = gamma**(t-1)
            else:
                beta_params[chosen_arm] += 1
                reward = 0.0

            # Accumulate the reward
            cumulative_reward += reward

        total_rewards.append(cumulative_reward)

    # Calculate average total reward for this gamma
    avg_reward = np.mean(total_rewards)

    # Compute theoretical optimal reward
    if gamma < 1.0:
        # Geometric series sum: theta_best * (1 - gamma^time_steps) / (1 - gamma)
        optimal_reward = theta_best * (1 - gamma**time_steps) / (1 - gamma)
    else:
        # Handle the case when gamma = 1.0
        optimal_reward = theta_best * time_steps

    # Compute the gap between optimal reward and algorithm's average reward
    gap = optimal_reward - avg_reward

    # Store the results
    gamma_list.append(gamma)
    gap_list.append(gap)

# Convert lists to numpy arrays for easier handling
gamma_array = np.array(gamma_list)
gap_array = np.array(gap_list)

# Visualization: Gap vs Gamma
plt.figure(figsize=(12, 6))
plt.plot(gamma_array, gap_array, marker='o', linestyle='-', color='red')
plt.title('Gap Between Optimal Reward and Algorithm\'s ATR vs Discount Factor γ', f
plt.xlabel('Discount Factor γ', fontsize=12)
plt.ylabel('Gap (Optimal - ATR)', fontsize=12)
plt.grid(True)
plt.xlim(0.95, 1.0)
plt.tight_layout()
plt.show()
```

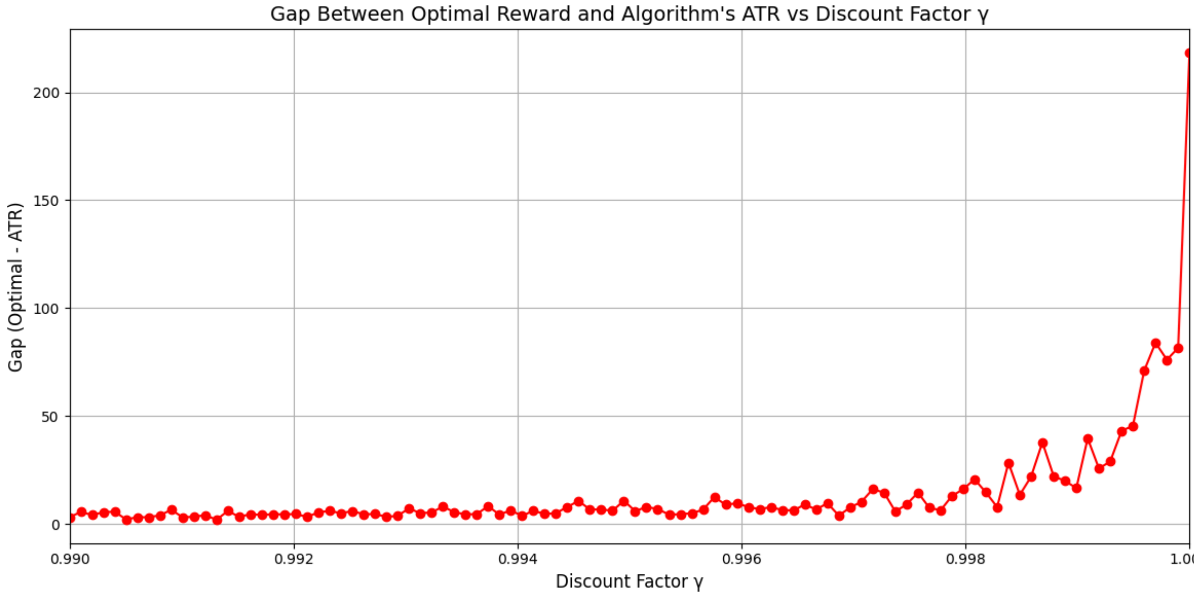


Problem 2

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

In [2]:
```
# Simulation parameters
true_theta = [0.3, 0.6]       # True success probabilities for arm 1 and arm 2
alpha_prior = [1, 1]          # Prior alpha parameters for Beta distributions
beta_prior = [1, 1]           # Prior beta parameters for Beta distributions
gamma_values = np.linspace(0.99, 1.00, 100)  # Gamma values
time_steps = 5000             # Number of pulls per trial
repeat_time = 50              # Number of trials per gamma

# Identify the best arm (with the highest true_theta)
best_arm = np.argmax(true_theta)
theta_best = true_theta[best_arm]

# Initialize lists to store results
gamma_list = []
gap_list = []

# Iterate over each gamma value
for gamma in gamma_values:
    total_rewards = []  # To store total rewards for each trial

    # Repeat the trial 'repeat_time' times for averaging
    for trial in range(repeat_time):
        # Initialize Beta parameters for each arm
        alpha = alpha_prior.copy()
        beta_params = beta_prior.copy()

        cumulative_reward = 0.0  # Total reward for this trial

        for t in range(1, time_steps + 1):
            # Calculate expected theta for each arm using current Beta posterior
            expected_theta = [alpha[i] / (alpha[i] + beta_params[i]) for i in range
```

```
            # Select the arm with the highest expected theta
            chosen_arm = np.argmax(expected_theta)

            # Simulate a pull: success with probability true_theta[chosen_arm]
            success = np.random.rand() < true_theta[chosen_arm]

            # Update Beta posterior based on the outcome
            if success:
                alpha[chosen_arm] += 1
                reward = gamma**(t-1)
            else:
                beta_params[chosen_arm] += 1
                reward = 0.0

            # Accumulate the reward
            cumulative_reward += reward

        total_rewards.append(cumulative_reward)

    # Calculate average total reward for this gamma
    avg_reward = np.mean(total_rewards)

    # Compute theoretical optimal reward
    if gamma < 1.0:
        # Geometric series sum: theta_best * (1 - gamma^time_steps) / (1 - gamma)
        optimal_reward = theta_best * (1 - gamma**time_steps) / (1 - gamma)
    else:
        # Handle the case when gamma = 1.0
        optimal_reward = theta_best * time_steps

    # Compute the gap between optimal reward and algorithm's average reward
    gap = optimal_reward - avg_reward

    # Store the results
    gamma_list.append(gamma)
    gap_list.append(gap)

# Convert lists to numpy arrays for easier handling
gamma_array = np.array(gamma_list)
gap_array = np.array(gap_list)

# Visualization: Gap vs Gamma
plt.figure(figsize=(12, 6))
plt.plot(gamma_array, gap_array, marker='o', linestyle='-', color='red')
plt.title('Gap Between Optimal Reward and Algorithm\'s ATR vs Discount Factor γ', f
plt.xlabel('Discount Factor γ', fontsize=12)
plt.ylabel('Gap (Optimal - ATR)', fontsize=12)
plt.grid(True)
plt.xlim(0.99, 1.0)
plt.tight_layout()
plt.show()
```

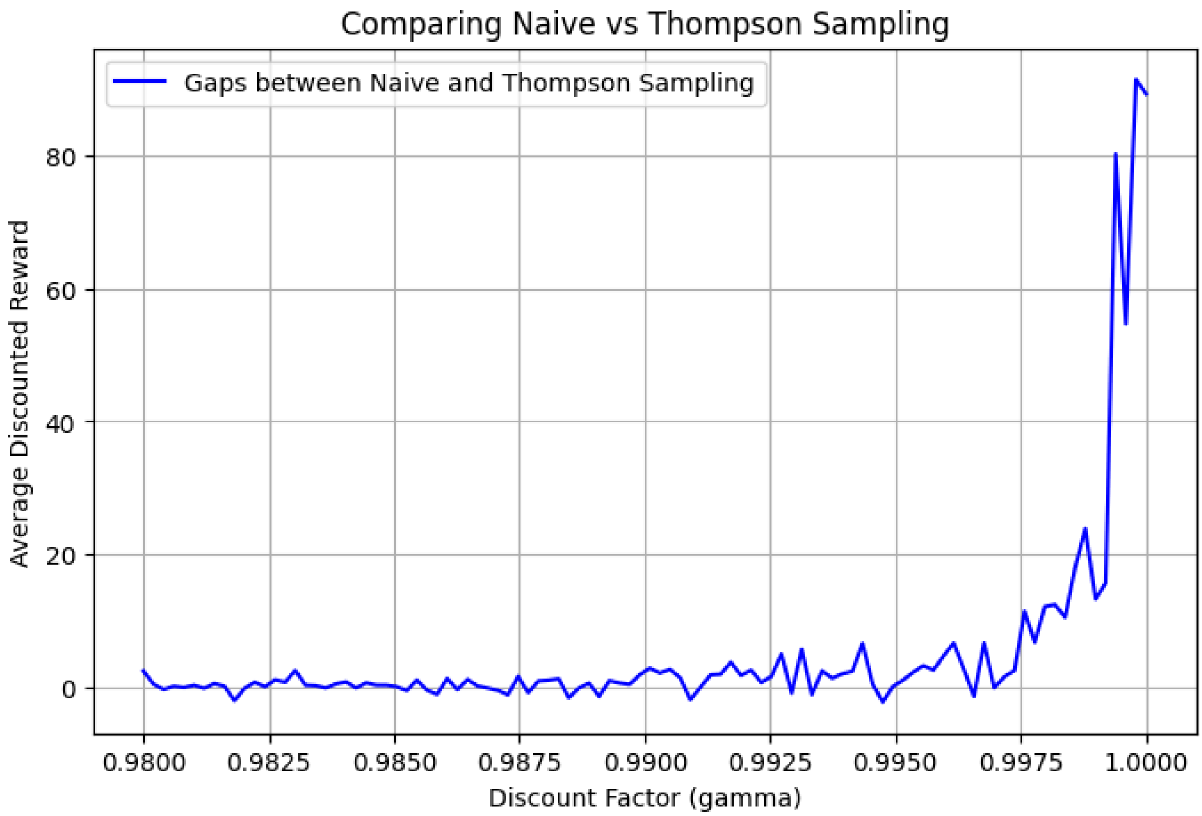


Compared with TS

In [6]:
```
# Simulation parameters
true_theta = [0.3, 0.6]        # True success probabilities for arm 1 and arm 2
alpha_prior = [1, 1]           # Prior alpha parameters for Beta distributions
beta_prior = [1, 1]            # Prior beta parameters for Beta distributions
gamma_values = np.linspace(0.98, 1.00, 100)  # Gamma values
time_steps = 5000              # Number of pulls per trial
repeat_time = 50               # Number of trials per gamma

# --------------------------------------------------------------
# Helper function: sample Bernoulli reward from an arm
# --------------------------------------------------------------
def draw_reward(arm_idx):
    """Simulate pulling arm_idx and return reward (1 or 0)."""
    return 1 if (np.random.rand() < true_theta[arm_idx]) else 0
# --------------------------------------------------------------
# Naive Strategy: Always pick arm with highest posterior mean
# --------------------------------------------------------------
def run_naive_strategy(gamma, alpha0, beta0):
    """
    Runs the naive strategy for 'time_steps' pulls with discount factor gamma.
    alpha0, beta0 are the prior parameters for each arm (list of length 2).
    Returns the total discounted reward.
    """
    # Initialize alpha, beta
    alpha = np.array(alpha0, dtype=float)
    beta  = np.array(beta0,  dtype=float)

    total_discounted_reward = 0.0
    discount_power = 0      # exponent for gamma^(t-1)

    for t in range(1, time_steps + 1):
        # Posterior means for each arm
        posterior_means = alpha / (alpha + beta)
```

```
        # Choose the arm with the highest posterior mean
        chosen_arm = np.argmax(posterior_means)

        # Draw a Bernoulli reward
        reward = draw_reward(chosen_arm)

        # Update posterior
        alpha[chosen_arm] += reward
        beta[chosen_arm]  += (1 - reward)

        # Accumulate discounted reward
        total_discounted_reward += (gamma ** discount_power) * reward
        discount_power += 1

    return total_discounted_reward
# -------------------------------------------------------------
# Thompson Sampling Strategy
# -------------------------------------------------------------
def thompson_sampling(gamma, alpha0, beta0):
    """
    Runs Thompson Sampling for 'time_steps' pulls with discount factor gamma.
    alpha0, beta0 are the prior parameters for each arm.
    Returns the total discounted reward.
    """
    alpha = np.array(alpha0, dtype=float)
    beta  = np.array(beta0,  dtype=float)

    total_discounted_reward = 0.0
    discount_power = 0

    for t in range(1, time_steps + 1):
        # Sample theta-hat from current posterior for each arm
        sampled_thetas = np.random.beta(alpha, beta)

        # Choose the arm that maximizes the sampled theta
        chosen_arm = np.argmax(sampled_thetas)

        # Draw reward
        reward = draw_reward(chosen_arm)

        # Update posterior
        alpha[chosen_arm] += reward
        beta[chosen_arm]  += (1 - reward)

        # Accumulate discounted reward
        total_discounted_reward += (gamma ** discount_power) * reward
        discount_power += 1

    return total_discounted_reward

# -------------------------------------------------------------
# Main Experiment Loop
# -------------------------------------------------------------
gap_means = []

for gamma in gamma_values:
```

```
    gap_results = []

    for _ in range(repeat_time):
        # Run Naive
        naive_reward = run_naive_strategy(
            gamma,
            alpha_prior,
            beta_prior
        )


        # Run Thompson Sampling
        ts_reward = thompson_sampling(
            gamma,
            alpha_prior,
            beta_prior
        )
        gap_results.append(ts_reward - naive_reward)

    gap_means.append(np.mean(gap_results))

# ---------------------------------------------------------------
# Plotting Results
# ---------------------------------------------------------------
plt.figure(figsize=(8, 5))
plt.plot(gamma_values, gap_means, 'b-', label='Gaps between Naive and Thompson Samp
plt.xlabel('Discount Factor (gamma)')
plt.ylabel('Average Discounted Reward')
plt.title('Comparing Naive vs Thompson Sampling')
plt.legend()
plt.grid(True)
plt.show()
```



Problem 5

Find the optimal policy (approximately).

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

# -------------------------------------------------------------
# Dynamic Programming Function
# -------------------------------------------------------------
def solve_2armed_bandit_dp(M=10, gamma=0.95, tol=1e-8, max_iter=50):
    """
    Solve the 2-armed Beta-Bernoulli bandit using 4D dynamic programming.

    Arguments:
    ----------
    M : int
        Truncation level for alpha_i, beta_i.
    gamma : float
        Discount factor in (0,1).
    tol : float
        Convergence tolerance for the value iteration.
    max_iter : int
        Maximum number of iterations to run.

    Returns:
    --------
    R : 4D numpy array, shape (M+1, M+1, M+1, M+1)
```

```
        The approximate value function.
    policy : 4D numpy array of 0 or 1
        Optimal action: 0 for arm1, 1 for arm2.
    """
    # Initialize value function and policy arrays
    R = np.zeros((M+1, M+1, M+1, M+1), dtype=np.float64)
    policy = np.zeros((M+1, M+1, M+1, M+1), dtype=int)

    def clamp(x):
        return min(x, M)

    for it in range(max_iter):
        delta = 0.0  # Maximum change in this iteration

        # Iterate over all possible states
        for alpha1 in range(1, M+1):
            for beta1 in range(1, M+1):
                for alpha2 in range(1, M+1):
                    for beta2 in range(1, M+1):
                        # Compute expected reward for choosing arm 1
                        p1 = alpha1 / (alpha1 + beta1)
                        R_success_1 = R[clamp(alpha1 + 1), beta1, alpha2, beta2]
                        R_fail_1 = R[alpha1, clamp(beta1 + 1), alpha2, beta2]
                        R1 = p1 * (1.0 + gamma * R_success_1) + (1.0 - p1) * (gamma

                        # Compute expected reward for choosing arm 2
                        p2 = alpha2 / (alpha2 + beta2)
                        R_success_2 = R[alpha1, beta1, clamp(alpha2 + 1), beta2]
                        R_fail_2 = R[alpha1, beta1, alpha2, clamp(beta2 + 1)]
                        R2 = p2 * (1.0 + gamma * R_success_2) + (1.0 - p2) * (gamma

                        # Choose the action with the higher expected reward
                        new_val = max(R1, R2)

                        # Update the value function
                        old_val = R[alpha1, beta1, alpha2, beta2]
                        diff = abs(new_val - old_val)
                        if diff > delta:
                            delta = diff
                        R[alpha1, beta1, alpha2, beta2] = new_val

                        # Update the policy
                        if R1 > R2:
                            policy[alpha1, beta1, alpha2, beta2] = 0  # Choose arm1
                        else:
                            policy[alpha1, beta1, alpha2, beta2] = 1  # Choose arm2

        if delta < tol:
            break

    return R, policy

# ------------------------------------------------------------
# Simulation Parameters
# ------------------------------------------------------------
true_theta = [0.3, 0.6]        # True success probabilities for arm 1 and arm 2
```

```python
alpha_prior = [1, 1]          # Prior alpha parameters for Beta distributions
beta_prior = [1, 1]           # Prior beta parameters for Beta distributions
gamma_values = np.linspace(0.9, 1, 100)  # Gamma values
time_steps = 5000             # Number of pulls per trial
repeat_time = 10              # Number of trials per gamma
M = 17                        # Truncation level for DP

# ------------------------------------------------------------
# Simulation Function for DP-Based Policy
# ------------------------------------------------------------
def simulate_trial(policy, M, true_theta, alpha_prior, beta_prior, gamma, time_step
    """
    Simulate a single trial of the bandit problem using the provided policy.

    Arguments:
    ----------
    policy : 4D numpy array
        Optimal policy derived from DP.
    M : int
        Truncation level.
    true_theta : list of float
        True success probabilities for each arm.
    alpha_prior : list of int
        Prior alpha parameters for Beta distributions.
    beta_prior : list of int
        Prior beta parameters for Beta distributions.
    gamma : float
        Discount factor.
    time_steps : int
        Number of pulls in the trial.

    Returns:
    --------
    total_reward : float
        Total discounted reward accumulated in the trial.
    """
    # Initialize Beta parameters
    alpha = [alpha_prior[0], alpha_prior[1]]
    beta = [beta_prior[0], beta_prior[1]]
    total_reward = 0.0
    current_gamma_power = 1.0  # gamma^{t-1}, starts at t=1

    for t in range(1, time_steps + 1):
        # Current state with truncation
        a1 = min(alpha[0], M)
        b1 = min(beta[0], M)
        a2 = min(alpha[1], M)
        b2 = min(beta[1], M)

        # Determine action from policy
        action = policy[a1, b1, a2, b2]

        # Pull the selected arm
        arm = action  # 0 or 1
        success = np.random.rand() < true_theta[arm]
        if success:
```

```
            total_reward += current_gamma_power  # Reward is gamma^{t-1}
            alpha[arm] += 1
        else:
            beta[arm] += 1

        # Update the discount factor for the next time step
        current_gamma_power *= gamma

    return total_reward

# ------------------------------------------------------------
# Thompson Sampling Simulation Function
# ------------------------------------------------------------
def thompson_sampling_simulation(true_theta, alpha_prior, beta_prior, gamma, time_s
    """
    Simulate a single trial of the bandit problem using Thompson Sampling.

    Arguments:
    ----------
    true_theta : list of float
        True success probabilities for each arm.
    alpha_prior : list of int
        Prior alpha parameters for Beta distributions.
    beta_prior : list of int
        Prior beta parameters for Beta distributions.
    gamma : float
        Discount factor.
    time_steps : int
        Number of pulls in the trial.

    Returns:
    --------
    total_reward : float
        Total discounted reward accumulated in the trial.
    """
    alpha = np.array(alpha_prior, dtype=float)
    beta = np.array(beta_prior, dtype=float)

    total_discounted_reward = 0.0
    discount_power = 0

    for t in range(1, time_steps + 1):
        # Sample theta-hat from current posterior for each arm
        sampled_thetas = np.random.beta(alpha, beta)

        # Choose the arm that maximizes the sampled theta
        chosen_arm = np.argmax(sampled_thetas)

        # Draw reward
        reward = 1 if np.random.rand() < true_theta[chosen_arm] else 0

        # Update posterior
        alpha[chosen_arm] += reward
        beta[chosen_arm] += (1 - reward)

        # Accumulate discounted reward
```

```
        total_discounted_reward += (gamma ** discount_power) * reward
        discount_power += 1

    return total_discounted_reward

# ---------------------------------------------------------------
# Gamma Evaluation Function
# ---------------------------------------------------------------
def evaluate_gamma(gamma):
    """
    Evaluate a single gamma value by solving DP and running simulations for both DP

    Arguments:
    ----------
    gamma : float
        Discount factor.

    Returns:
    --------
    gamma : float
        The gamma value evaluated.
    average_reward_dp : float
        Average total discounted reward over all DP trials.
    average_reward_ts : float
        Average total discounted reward over all TS trials.
    """
    print(f"Evaluating gamma = {gamma:.4f}")

    # Solve DP to get the policy
    R, policy = solve_2armed_bandit_dp(M=M, gamma=gamma)

    # Initialize total rewards for all trials
    total_rewards_dp = np.zeros(repeat_time, dtype=np.float64)
    total_rewards_ts = np.zeros(repeat_time, dtype=np.float64)

    # Simulate all trials for DP-based policy
    for trial in range(1, repeat_time + 1):
        reward = simulate_trial(
            policy, M, true_theta, alpha_prior, beta_prior, gamma, time_steps
        )
        total_rewards_dp[trial - 1] = reward

    # Simulate all trials for Thompson Sampling policy
    for trial in range(1, repeat_time + 1):
        reward_ts = thompson_sampling_simulation(
            true_theta, alpha_prior, beta_prior, gamma, time_steps
        )
        total_rewards_ts[trial - 1] = reward_ts

    # Calculate average rewards
    average_reward_dp = np.mean(total_rewards_dp)
    average_reward_ts = np.mean(total_rewards_ts)

    print(f"Gamma={gamma:.4f}: DP Avg Reward={average_reward_dp:.2f}, TS Avg Reward

    return gamma, average_reward_dp, average_reward_ts
```

```
# ------------------------------------------------------------
# Main Evaluation Loop
# ------------------------------------------------------------
# Initialize lists to store results
results_dp = []
results_ts = []

# Total number of gamma values
total_gammas = len(gamma_values)

# Iterate over gamma_values and collect results
for idx, gamma in enumerate(gamma_values, 1):
    print(f"\nProcessing gamma {idx}/{total_gammas}: gamma = {gamma:.4f}")
    gamma_result = evaluate_gamma(gamma)
    _, avg_dp, avg_ts = gamma_result
    results_dp.append(avg_dp)
    results_ts.append(avg_ts)

print("\nAll gamma values have been evaluated.\n")

# Convert results to numpy arrays for easier processing
gamma_evaluated = np.array(gamma_values)
avg_rewards_dp = np.array(results_dp)
avg_rewards_ts = np.array(results_ts)

# Compute the gap between DP and TS
gap = avg_rewards_dp - avg_rewards_ts

# ------------------------------------------------------------
# Find the gamma with the highest average reward for DP
# ------------------------------------------------------------
optimal_index = np.argmax(avg_rewards_dp)
optimal_gamma = gamma_evaluated[optimal_index]
optimal_reward_dp = avg_rewards_dp[optimal_index]
optimal_reward_ts = avg_rewards_ts[optimal_index]

print(f"Optimal gamma for DP: {optimal_gamma:.4f}")
print(f"DP Reward at Optimal Gamma: {optimal_reward_dp:.2f}")
print(f"TS Reward at Optimal Gamma: {optimal_reward_ts:.2f}")

# ------------------------------------------------------------
# Plot the Results
# ------------------------------------------------------------
plt.figure(figsize=(14, 6))

# Plot Average Rewards for DP and TS
plt.subplot(1, 2, 1)
plt.plot(gamma_evaluated, avg_rewards_dp, linestyle='-', color='blue', label='DP Op
plt.plot(gamma_evaluated, avg_rewards_ts, linestyle='--', color='green', label='Tho
plt.xlabel('Gamma')
plt.ylabel('Average Discounted Reward')
plt.title('Average Discounted Reward vs Gamma')
plt.axvline(optimal_gamma, color='red', linestyle='--', label=f'Optimal Gamma: {opt
plt.legend()
plt.grid(True)
```

```python
# Plot the Gap between DP and TS
plt.subplot(1, 2, 2)
plt.plot(gamma_evaluated, gap, linestyle='-', color='purple')
plt.xlabel('Gamma')
plt.ylabel('Reward Gap (DP - TS)')
plt.title('Gap Between DP Optimal Policy and Thompson Sampling')
plt.axvline(optimal_gamma, color='red', linestyle='--', label=f'Optimal Gamma: {opt
plt.legend()
plt.grid(True)

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

```
Processing gamma 1/100: gamma = 0.9000
Evaluating gamma = 0.9000
Gamma=0.9000: DP Avg Reward=4.23, TS Avg Reward=5.18

Processing gamma 2/100: gamma = 0.9010
Evaluating gamma = 0.9010
Gamma=0.9010: DP Avg Reward=5.87, TS Avg Reward=4.99

Processing gamma 3/100: gamma = 0.9020
Evaluating gamma = 0.9020
Gamma=0.9020: DP Avg Reward=5.06, TS Avg Reward=5.79

Processing gamma 4/100: gamma = 0.9030
Evaluating gamma = 0.9030
Gamma=0.9030: DP Avg Reward=6.04, TS Avg Reward=5.97

Processing gamma 5/100: gamma = 0.9040
Evaluating gamma = 0.9040
Gamma=0.9040: DP Avg Reward=5.80, TS Avg Reward=5.14

Processing gamma 6/100: gamma = 0.9051
Evaluating gamma = 0.9051
Gamma=0.9051: DP Avg Reward=5.22, TS Avg Reward=5.12

Processing gamma 7/100: gamma = 0.9061
Evaluating gamma = 0.9061
Gamma=0.9061: DP Avg Reward=5.56, TS Avg Reward=5.59

Processing gamma 8/100: gamma = 0.9071
Evaluating gamma = 0.9071
Gamma=0.9071: DP Avg Reward=6.20, TS Avg Reward=5.20

Processing gamma 9/100: gamma = 0.9081
Evaluating gamma = 0.9081
Gamma=0.9081: DP Avg Reward=6.30, TS Avg Reward=5.83

Processing gamma 10/100: gamma = 0.9091
Evaluating gamma = 0.9091
Gamma=0.9091: DP Avg Reward=6.32, TS Avg Reward=5.02

Processing gamma 11/100: gamma = 0.9101
Evaluating gamma = 0.9101
Gamma=0.9101: DP Avg Reward=5.29, TS Avg Reward=5.70

Processing gamma 12/100: gamma = 0.9111
Evaluating gamma = 0.9111
Gamma=0.9111: DP Avg Reward=6.39, TS Avg Reward=6.32

Processing gamma 13/100: gamma = 0.9121
Evaluating gamma = 0.9121
Gamma=0.9121: DP Avg Reward=7.09, TS Avg Reward=5.13

Processing gamma 14/100: gamma = 0.9131
Evaluating gamma = 0.9131
Gamma=0.9131: DP Avg Reward=6.68, TS Avg Reward=6.59
```

```
Processing gamma 15/100: gamma = 0.9141
Evaluating gamma = 0.9141
Gamma=0.9141: DP Avg Reward=6.42, TS Avg Reward=6.76

Processing gamma 16/100: gamma = 0.9152
Evaluating gamma = 0.9152
Gamma=0.9152: DP Avg Reward=6.82, TS Avg Reward=6.35

Processing gamma 17/100: gamma = 0.9162
Evaluating gamma = 0.9162
Gamma=0.9162: DP Avg Reward=6.66, TS Avg Reward=5.86

Processing gamma 18/100: gamma = 0.9172
Evaluating gamma = 0.9172
Gamma=0.9172: DP Avg Reward=6.40, TS Avg Reward=5.62

Processing gamma 19/100: gamma = 0.9182
Evaluating gamma = 0.9182
Gamma=0.9182: DP Avg Reward=7.05, TS Avg Reward=6.29

Processing gamma 20/100: gamma = 0.9192
Evaluating gamma = 0.9192
Gamma=0.9192: DP Avg Reward=5.76, TS Avg Reward=6.04

Processing gamma 21/100: gamma = 0.9202
Evaluating gamma = 0.9202
Gamma=0.9202: DP Avg Reward=6.67, TS Avg Reward=6.43

Processing gamma 22/100: gamma = 0.9212
Evaluating gamma = 0.9212
Gamma=0.9212: DP Avg Reward=6.55, TS Avg Reward=6.90

Processing gamma 23/100: gamma = 0.9222
Evaluating gamma = 0.9222
Gamma=0.9222: DP Avg Reward=7.24, TS Avg Reward=6.80

Processing gamma 24/100: gamma = 0.9232
Evaluating gamma = 0.9232
Gamma=0.9232: DP Avg Reward=7.34, TS Avg Reward=7.15

Processing gamma 25/100: gamma = 0.9242
Evaluating gamma = 0.9242
Gamma=0.9242: DP Avg Reward=7.07, TS Avg Reward=6.72

Processing gamma 26/100: gamma = 0.9253
Evaluating gamma = 0.9253
Gamma=0.9253: DP Avg Reward=8.15, TS Avg Reward=6.17

Processing gamma 27/100: gamma = 0.9263
Evaluating gamma = 0.9263
Gamma=0.9263: DP Avg Reward=8.09, TS Avg Reward=7.71

Processing gamma 28/100: gamma = 0.9273
Evaluating gamma = 0.9273
Gamma=0.9273: DP Avg Reward=8.47, TS Avg Reward=6.94
```

```
Processing gamma 29/100: gamma = 0.9283
Evaluating gamma = 0.9283
Gamma=0.9283: DP Avg Reward=7.42, TS Avg Reward=6.37

Processing gamma 30/100: gamma = 0.9293
Evaluating gamma = 0.9293
Gamma=0.9293: DP Avg Reward=7.28, TS Avg Reward=7.12

Processing gamma 31/100: gamma = 0.9303
Evaluating gamma = 0.9303
Gamma=0.9303: DP Avg Reward=7.80, TS Avg Reward=6.92

Processing gamma 32/100: gamma = 0.9313
Evaluating gamma = 0.9313
Gamma=0.9313: DP Avg Reward=7.46, TS Avg Reward=8.34

Processing gamma 33/100: gamma = 0.9323
Evaluating gamma = 0.9323
Gamma=0.9323: DP Avg Reward=8.42, TS Avg Reward=8.02

Processing gamma 34/100: gamma = 0.9333
Evaluating gamma = 0.9333
Gamma=0.9333: DP Avg Reward=7.52, TS Avg Reward=8.12

Processing gamma 35/100: gamma = 0.9343
Evaluating gamma = 0.9343
Gamma=0.9343: DP Avg Reward=8.76, TS Avg Reward=8.06

Processing gamma 36/100: gamma = 0.9354
Evaluating gamma = 0.9354
Gamma=0.9354: DP Avg Reward=8.03, TS Avg Reward=7.71

Processing gamma 37/100: gamma = 0.9364
Evaluating gamma = 0.9364
Gamma=0.9364: DP Avg Reward=8.45, TS Avg Reward=7.29

Processing gamma 38/100: gamma = 0.9374
Evaluating gamma = 0.9374
Gamma=0.9374: DP Avg Reward=9.14, TS Avg Reward=8.19

Processing gamma 39/100: gamma = 0.9384
Evaluating gamma = 0.9384
Gamma=0.9384: DP Avg Reward=10.54, TS Avg Reward=8.47

Processing gamma 40/100: gamma = 0.9394
Evaluating gamma = 0.9394
Gamma=0.9394: DP Avg Reward=8.79, TS Avg Reward=8.70

Processing gamma 41/100: gamma = 0.9404
Evaluating gamma = 0.9404
Gamma=0.9404: DP Avg Reward=8.27, TS Avg Reward=9.21

Processing gamma 42/100: gamma = 0.9414
Evaluating gamma = 0.9414
Gamma=0.9414: DP Avg Reward=9.20, TS Avg Reward=9.22
```

```
Processing gamma 43/100: gamma = 0.9424
Evaluating gamma = 0.9424
Gamma=0.9424: DP Avg Reward=9.35, TS Avg Reward=8.63

Processing gamma 44/100: gamma = 0.9434
Evaluating gamma = 0.9434
Gamma=0.9434: DP Avg Reward=9.59, TS Avg Reward=9.23

Processing gamma 45/100: gamma = 0.9444
Evaluating gamma = 0.9444
Gamma=0.9444: DP Avg Reward=9.20, TS Avg Reward=10.17

Processing gamma 46/100: gamma = 0.9455
Evaluating gamma = 0.9455
Gamma=0.9455: DP Avg Reward=10.27, TS Avg Reward=9.44

Processing gamma 47/100: gamma = 0.9465
Evaluating gamma = 0.9465
Gamma=0.9465: DP Avg Reward=9.82, TS Avg Reward=9.38

Processing gamma 48/100: gamma = 0.9475
Evaluating gamma = 0.9475
Gamma=0.9475: DP Avg Reward=9.56, TS Avg Reward=10.37

Processing gamma 49/100: gamma = 0.9485
Evaluating gamma = 0.9485
Gamma=0.9485: DP Avg Reward=9.20, TS Avg Reward=9.30

Processing gamma 50/100: gamma = 0.9495
Evaluating gamma = 0.9495
Gamma=0.9495: DP Avg Reward=10.23, TS Avg Reward=9.72

Processing gamma 51/100: gamma = 0.9505
Evaluating gamma = 0.9505
Gamma=0.9505: DP Avg Reward=11.19, TS Avg Reward=9.61

Processing gamma 52/100: gamma = 0.9515
Evaluating gamma = 0.9515
Gamma=0.9515: DP Avg Reward=10.85, TS Avg Reward=10.47

Processing gamma 53/100: gamma = 0.9525
Evaluating gamma = 0.9525
Gamma=0.9525: DP Avg Reward=11.53, TS Avg Reward=11.00

Processing gamma 54/100: gamma = 0.9535
Evaluating gamma = 0.9535
Gamma=0.9535: DP Avg Reward=12.21, TS Avg Reward=11.62

Processing gamma 55/100: gamma = 0.9545
Evaluating gamma = 0.9545
Gamma=0.9545: DP Avg Reward=9.53, TS Avg Reward=10.93

Processing gamma 56/100: gamma = 0.9556
Evaluating gamma = 0.9556
Gamma=0.9556: DP Avg Reward=11.30, TS Avg Reward=11.66
```

```
Processing gamma 57/100: gamma = 0.9566
Evaluating gamma = 0.9566
Gamma=0.9566: DP Avg Reward=12.15, TS Avg Reward=12.40

Processing gamma 58/100: gamma = 0.9576
Evaluating gamma = 0.9576
Gamma=0.9576: DP Avg Reward=12.47, TS Avg Reward=12.68

Processing gamma 59/100: gamma = 0.9586
Evaluating gamma = 0.9586
Gamma=0.9586: DP Avg Reward=13.73, TS Avg Reward=12.77

Processing gamma 60/100: gamma = 0.9596
Evaluating gamma = 0.9596
Gamma=0.9596: DP Avg Reward=14.52, TS Avg Reward=12.84

Processing gamma 61/100: gamma = 0.9606
Evaluating gamma = 0.9606
Gamma=0.9606: DP Avg Reward=14.34, TS Avg Reward=13.61

Processing gamma 62/100: gamma = 0.9616
Evaluating gamma = 0.9616
Gamma=0.9616: DP Avg Reward=14.11, TS Avg Reward=13.29

Processing gamma 63/100: gamma = 0.9626
Evaluating gamma = 0.9626
Gamma=0.9626: DP Avg Reward=14.08, TS Avg Reward=13.75

Processing gamma 64/100: gamma = 0.9636
Evaluating gamma = 0.9636
Gamma=0.9636: DP Avg Reward=15.20, TS Avg Reward=14.47

Processing gamma 65/100: gamma = 0.9646
Evaluating gamma = 0.9646
Gamma=0.9646: DP Avg Reward=16.59, TS Avg Reward=15.02

Processing gamma 66/100: gamma = 0.9657
Evaluating gamma = 0.9657
Gamma=0.9657: DP Avg Reward=17.22, TS Avg Reward=15.81

Processing gamma 67/100: gamma = 0.9667
Evaluating gamma = 0.9667
Gamma=0.9667: DP Avg Reward=15.93, TS Avg Reward=15.92

Processing gamma 68/100: gamma = 0.9677
Evaluating gamma = 0.9677
Gamma=0.9677: DP Avg Reward=18.08, TS Avg Reward=16.25

Processing gamma 69/100: gamma = 0.9687
Evaluating gamma = 0.9687
Gamma=0.9687: DP Avg Reward=18.17, TS Avg Reward=17.32

Processing gamma 70/100: gamma = 0.9697
Evaluating gamma = 0.9697
Gamma=0.9697: DP Avg Reward=18.94, TS Avg Reward=16.87
```

```
Processing gamma 71/100: gamma = 0.9707
Evaluating gamma = 0.9707
Gamma=0.9707: DP Avg Reward=18.94, TS Avg Reward=19.12

Processing gamma 72/100: gamma = 0.9717
Evaluating gamma = 0.9717
Gamma=0.9717: DP Avg Reward=21.93, TS Avg Reward=19.63

Processing gamma 73/100: gamma = 0.9727
Evaluating gamma = 0.9727
Gamma=0.9727: DP Avg Reward=21.67, TS Avg Reward=19.69

Processing gamma 74/100: gamma = 0.9737
Evaluating gamma = 0.9737
Gamma=0.9737: DP Avg Reward=20.85, TS Avg Reward=19.93

Processing gamma 75/100: gamma = 0.9747
Evaluating gamma = 0.9747
Gamma=0.9747: DP Avg Reward=19.46, TS Avg Reward=22.04

Processing gamma 76/100: gamma = 0.9758
Evaluating gamma = 0.9758
Gamma=0.9758: DP Avg Reward=25.31, TS Avg Reward=22.38

Processing gamma 77/100: gamma = 0.9768
Evaluating gamma = 0.9768
Gamma=0.9768: DP Avg Reward=25.60, TS Avg Reward=22.36

Processing gamma 78/100: gamma = 0.9778
Evaluating gamma = 0.9778
Gamma=0.9778: DP Avg Reward=24.24, TS Avg Reward=25.46

Processing gamma 79/100: gamma = 0.9788
Evaluating gamma = 0.9788
Gamma=0.9788: DP Avg Reward=24.13, TS Avg Reward=25.34

Processing gamma 80/100: gamma = 0.9798
Evaluating gamma = 0.9798
Gamma=0.9798: DP Avg Reward=28.80, TS Avg Reward=25.61

Processing gamma 81/100: gamma = 0.9808
Evaluating gamma = 0.9808
Gamma=0.9808: DP Avg Reward=28.95, TS Avg Reward=27.10

Processing gamma 82/100: gamma = 0.9818
Evaluating gamma = 0.9818
Gamma=0.9818: DP Avg Reward=29.40, TS Avg Reward=30.41

Processing gamma 83/100: gamma = 0.9828
Evaluating gamma = 0.9828
Gamma=0.9828: DP Avg Reward=34.18, TS Avg Reward=32.91

Processing gamma 84/100: gamma = 0.9838
Evaluating gamma = 0.9838
Gamma=0.9838: DP Avg Reward=34.37, TS Avg Reward=34.12
```

```
Processing gamma 85/100: gamma = 0.9848
Evaluating gamma = 0.9848
Gamma=0.9848: DP Avg Reward=34.12, TS Avg Reward=37.57

Processing gamma 86/100: gamma = 0.9859
Evaluating gamma = 0.9859
Gamma=0.9859: DP Avg Reward=39.85, TS Avg Reward=41.39

Processing gamma 87/100: gamma = 0.9869
Evaluating gamma = 0.9869
Gamma=0.9869: DP Avg Reward=43.05, TS Avg Reward=43.11

Processing gamma 88/100: gamma = 0.9879
Evaluating gamma = 0.9879
Gamma=0.9879: DP Avg Reward=46.48, TS Avg Reward=47.55

Processing gamma 89/100: gamma = 0.9889
Evaluating gamma = 0.9889
Gamma=0.9889: DP Avg Reward=50.42, TS Avg Reward=51.01

Processing gamma 90/100: gamma = 0.9899
Evaluating gamma = 0.9899
Gamma=0.9899: DP Avg Reward=57.54, TS Avg Reward=57.58

Processing gamma 91/100: gamma = 0.9909
Evaluating gamma = 0.9909
Gamma=0.9909: DP Avg Reward=61.79, TS Avg Reward=67.40

Processing gamma 92/100: gamma = 0.9919
Evaluating gamma = 0.9919
Gamma=0.9919: DP Avg Reward=70.39, TS Avg Reward=71.11

Processing gamma 93/100: gamma = 0.9929
Evaluating gamma = 0.9929
Gamma=0.9929: DP Avg Reward=78.76, TS Avg Reward=79.83

Processing gamma 94/100: gamma = 0.9939
Evaluating gamma = 0.9939
Gamma=0.9939: DP Avg Reward=97.94, TS Avg Reward=96.37

Processing gamma 95/100: gamma = 0.9949
Evaluating gamma = 0.9949
Gamma=0.9949: DP Avg Reward=110.69, TS Avg Reward=116.20

Processing gamma 96/100: gamma = 0.9960
Evaluating gamma = 0.9960
Gamma=0.9960: DP Avg Reward=133.00, TS Avg Reward=145.14

Processing gamma 97/100: gamma = 0.9970
Evaluating gamma = 0.9970
Gamma=0.9970: DP Avg Reward=194.02, TS Avg Reward=193.00

Processing gamma 98/100: gamma = 0.9980
Evaluating gamma = 0.9980
Gamma=0.9980: DP Avg Reward=289.31, TS Avg Reward=286.59
```

```
Processing gamma 99/100: gamma = 0.9990
Evaluating gamma = 0.9990
Gamma=0.9990: DP Avg Reward=555.59, TS Avg Reward=582.36

Processing gamma 100/100: gamma = 1.0000
Evaluating gamma = 1.0000
Gamma=1.0000: DP Avg Reward=2996.70, TS Avg Reward=3000.90

All gamma values have been evaluated.

Optimal gamma for DP: 1.0000
DP Reward at Optimal Gamma: 2996.70
TS Reward at Optimal Gamma: 3000.90
```