**Experiment No.02**

PART A

(PART A: TO BE REFFERED BY STUDENTS)

**A.1 Aim: Implementation of Multi-Arm Bandit (MAB) Problem**

1. To implement MAB problem using pure exploitation algorithm.
2. To implement MAB problem using pure exploration algorithm
3. To implement MAB problem using Fixed Exploration followed by Exploitation
4. To implement MAB problem using Greedy algorithm
5. To apply Upper Confidence Bound(UCB) in the implementation done in part d
6. To analyse the algorithms and compare them in terms of rewards, regrets and complexity

**A.2 Prerequisite:**

Concept of Multi-Arm Bandit, Exploration, Exploitation, Greedy, UCB

**A.3 Learning Outcome:**

After completing thisexperiment you will be able to-

* Comprehend the fundamental concept of Multi-Arm Bandit Problem
* Implementation of various strategies
* Fine-tuning of to balance exploration and exploitation
* Comparative analysis of all the algorithms to identify algorithm with maximum award

**A.4 Theory:**

**A.4.1 Multi-Arm Bandit Problem**

The Multi-Arm Bandit (MAB) problem is a classic reinforcement-learning problem that explores the trade-off between exploitation (choosing the best-known option) and exploration (trying new options to discover their potential). In this problem, an agent is faced with multiple slot machines (referred to as arms), each with an unknown probability of winning. The agent's goal is to maximize the total reward over a series of trials by deciding which arm to pull in each trial.

**Exploration**

**Exploration** refers to the strategy where an agent tries out different actions to gather more information about their potential rewards. In the context of the Multi-Arm Bandit problem, this means pulling different arms to learn more about their reward distributions.

**Exploitation**

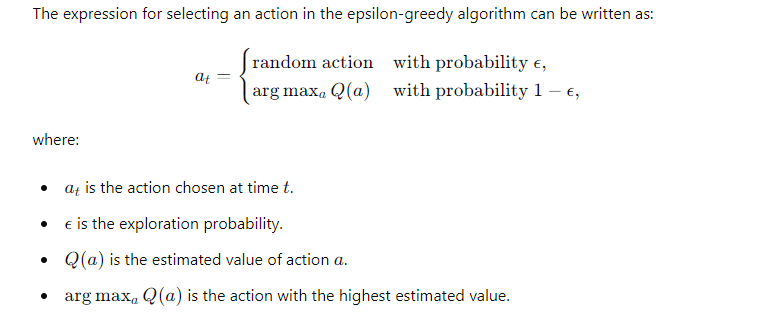
**Exploitation** involves selecting the action that is currently believed to be the best based on experiences. In the Multi-Arm Bandit problem, this means always pulling the arm with the highest estimated reward.

**Epsilon-Greedy Algorithm**

The **epsilon-greedy algorithm** is a simple and effective strategy that combines both exploration and exploitation. The key idea is to choose the best-known action most of the time, but occasionally explore other actions. This balance is controlled by a parameter, epsilon (ε)

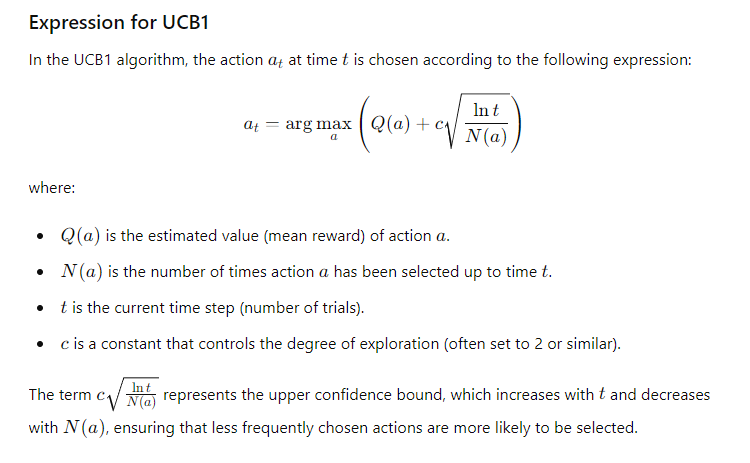
**How Epsilon-Greedy Works**

* **Parameter ε**: A value between 0 and 1 that determines the likelihood of exploration versus exploitation. For example, ε = 0.1 means that there is a 10% chance of exploring and a 90% chance of exploiting.
* **Decision Rule**:
  + With probability ϵ, choose a random action (exploration).
  + With probability 1−ϵ, choose the action with the highest estimated reward (exploitation).
* **Update**: After selecting an action and receiving a reward, update the estimated rewards based on the observed outcomes.



**UCB**

The UCB algorithm balances exploration and exploitation by considering the uncertainty in the estimates of the rewards. It selects the arm with the highest upper confidence bound.



**A.5 Task to be completed:**

1. Take value of arm as n=5 and number of trials t =500, Positive Reward =1, Negative Reward =0
   1. Implement Exploration
   2. Implement Exploitation
   3. Implement Fixed Exploration and then Exploitation
   4. Implement greedy
   5. Fine tune the value of and encapsulate your observations
   6. Compare the algorithms in terms or complexity, value and regret
   7. Incorporate UCB in greedy and state your opinion
2. Take different values of n and t and summarize your observations
3. Give two real world applications of MAB

**References**

[**https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-python/**](https://www.analyticsvidhya.com/blog/2018/09/reinforcement-multi-armed-bandit-scratch-python/)

[**https://www.geeksforgeeks.org/multi-armed-bandit-problem-in-reinforcement-learning/**](https://www.geeksforgeeks.org/multi-armed-bandit-problem-in-reinforcement-learning/)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**PART B**

(PART B: TO BE COMPLETED BY STUDENTS)

**(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Portal/MS Teams assignment link at the end of the practical)**

|  |  |
| --- | --- |
| Roll No. C050 | Name: Nisha Kini |
| Program: BTI | Division: B |
| Batch: B2 | Date of Experiment: 8.1.25 |
| Date of Submission: 8.1.25 | Grade : |

**B.1 Tasks given in PART A to be completed here**

**Implementation of Multi-Arm Bandit (MAB) Problem**

1. To implement MAB problem using pure exploitation algorithm.

CODE: import numpy as np

import matplotlib.pyplot as plt

np.random.seed(42)

class MultiArmBandit:

def \_\_init\_\_(self, n\_arms, means, std\_devs):

self.n\_arms = n\_arms

self.means = means

self.std\_devs = std\_devs

def pull(self, arm):

"""Simulate pulling the arm by sampling from a normal distribution"""

return np.random.normal(self.means[arm], self.std\_devs[arm])

Screenshot:



1. To implement MAB problem using pure exploration algorithm

CODE:

def pure\_exploitation(bandit, n\_steps):

total\_reward = 0

rewards = []

arm\_counts = np.zeros(bandit.n\_arms)

arm\_rewards = np.zeros(bandit.n\_arms)

for t in range(n\_steps):

# Choose the arm with the highest average reward

best\_arm = np.argmax(arm\_rewards / (arm\_counts + 1e-5))

reward = bandit.pull(best\_arm)

arm\_counts[best\_arm] += 1

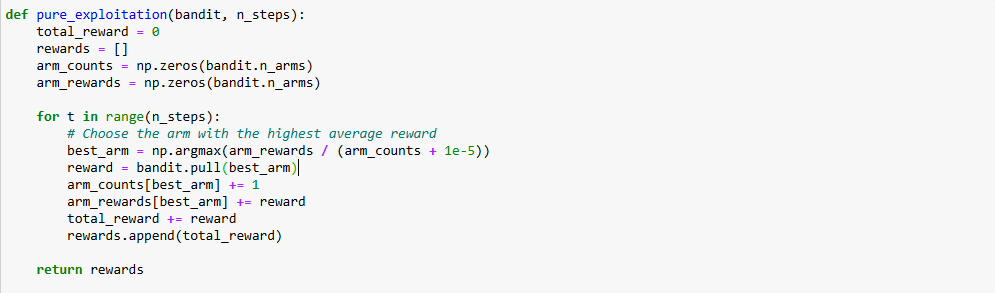
arm\_rewards[best\_arm] += reward

total\_reward += reward

rewards.append(total\_reward)

return rewards

Screenshot:



1. To implement MAB problem using Fixed Exploration followed by Exploitation

CODE:

def pure\_exploration(bandit, n\_steps):

total\_reward = 0

rewards = []

for t in range(n\_steps):

# Randomly select an arm to explore

arm = np.random.randint(0, bandit.n\_arms)

reward = bandit.pull(arm)

total\_reward += reward

rewards.append(total\_reward)

return rewards

def fixed\_exploration\_then\_exploitation(bandit, n\_steps, n\_explore):

total\_reward = 0

rewards = []

arm\_counts = np.zeros(bandit.n\_arms)

arm\_rewards = np.zeros(bandit.n\_arms)

# Exploration phase

for t in range(n\_explore):

arm = np.random.randint(0, bandit.n\_arms)

reward = bandit.pull(arm)

total\_reward += reward

rewards.append(total\_reward)

# Exploitation phase

for t in range(n\_explore, n\_steps):

best\_arm = np.argmax(arm\_rewards / (arm\_counts + 1e-5))

reward = bandit.pull(best\_arm)

arm\_counts[best\_arm] += 1

arm\_rewards[best\_arm] += reward

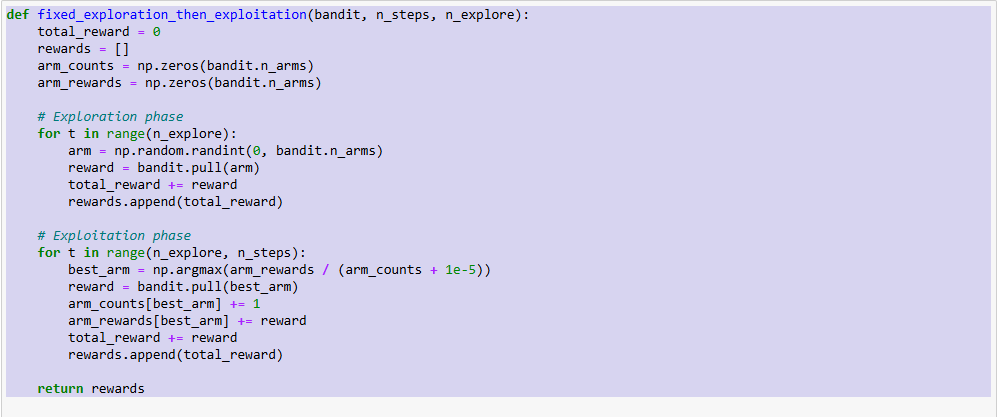
total\_reward += reward

rewards.append(total\_reward)

return rewards

Screenshot:





1. To implement MAB problem using Greedy algorithm

CODE: def epsilon\_greedy(bandit, n\_steps, epsilon):

total\_reward = 0

rewards = []

arm\_counts = np.zeros(bandit.n\_arms)

arm\_rewards = np.zeros(bandit.n\_arms)

for t in range(n\_steps):

if np.random.rand() < epsilon:

# Explore: choose a random arm

arm = np.random.randint(0, bandit.n\_arms)

else:

# Exploit: choose the arm with the highest estimated reward

arm = np.argmax(arm\_rewards / (arm\_counts + 1e-5))

reward = bandit.pull(arm)

arm\_counts[arm] += 1

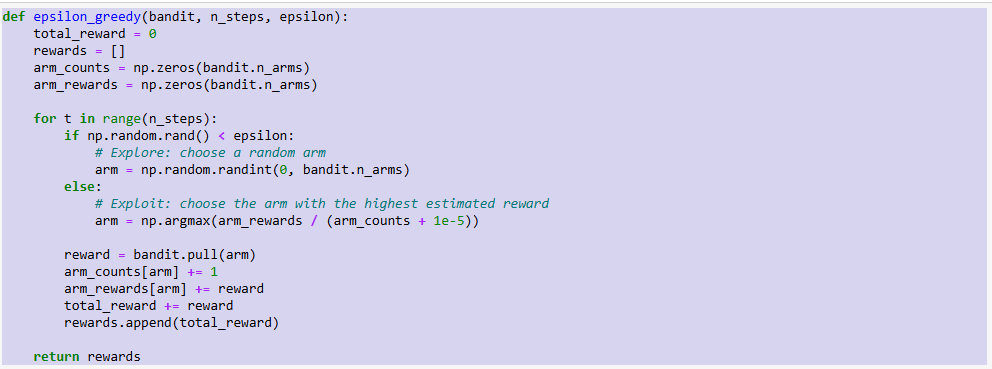
arm\_rewards[arm] += reward

total\_reward += reward

rewards.append(total\_reward)

return rewards

Screenshot:



1. To apply Upper Confidence Bound(UCB) in the implementation done in part d

CODE: def ucb(bandit, n\_steps):

total\_reward = 0

rewards = []

arm\_counts = np.zeros(bandit.n\_arms)

arm\_rewards = np.zeros(bandit.n\_arms)

for t in range(n\_steps):

# Select arm with highest upper confidence bound

if t < bandit.n\_arms:

arm = t # Initially, pull each arm once

else:

ucb\_values = arm\_rewards / (arm\_counts + 1e-5) + np.sqrt(2 \* np.log(t) / (arm\_counts + 1e-5))

arm = np.argmax(ucb\_values)

reward = bandit.pull(arm)

arm\_counts[arm] += 1

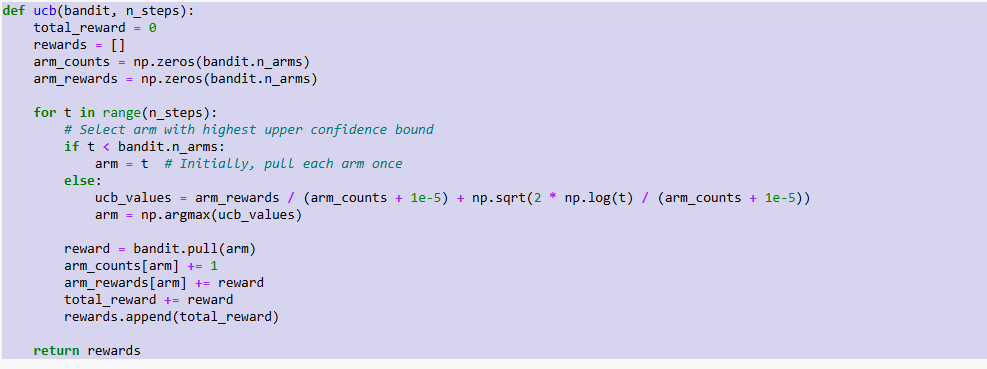
arm\_rewards[arm] += reward

total\_reward += reward

rewards.append(total\_reward)

return rewards

Screenshot:



1. To analyse the algorithms and compare them in terms of rewards, regrets and complexity

CODE:

def simulate\_bandit(strategy, bandit, n\_steps, \*\*kwargs):

rewards = strategy(bandit, n\_steps, \*\*kwargs)

return np.array(rewards)

def plot\_comparison(rewards\_dict, n\_steps):

plt.figure(figsize=(10, 6))

for label, rewards in rewards\_dict.items():

plt.plot(rewards, label=label)

plt.xlabel("Steps")

plt.ylabel("Cumulative Reward")

plt.title("Cumulative Reward for Different Strategies")

plt.legend()

plt.show()

def calculate\_regret(bandit, n\_steps, rewards):

optimal\_reward = np.max(bandit.means) \* n\_steps # Max possible reward for the best arm

regret = optimal\_reward - rewards[-1]

return regret

n\_arms = 5

means = np.random.uniform(0, 1, n\_arms) # Random means for each arm

std\_devs = np.ones(n\_arms) \* 0.1 # Standard deviation for rewards

bandit = MultiArmBandit(n\_arms, means, std\_devs)

n\_steps = 1000

n\_explore = 100 # For fixed exploration strategy

rewards\_pure\_exploitation = simulate\_bandit(pure\_exploitation, bandit, n\_steps)

rewards\_pure\_exploration = simulate\_bandit(pure\_exploration, bandit, n\_steps)

rewards\_fixed\_explore = simulate\_bandit(fixed\_exploration\_then\_exploitation, bandit, n\_steps, n\_explore=n\_explore)

rewards\_epsilon\_greedy = simulate\_bandit(epsilon\_greedy, bandit, n\_steps, epsilon=0.1)

rewards\_ucb = simulate\_bandit(ucb, bandit, n\_steps)

# Plot

rewards\_dict = {

"Pure Exploitation": rewards\_pure\_exploitation,

"Pure Exploration": rewards\_pure\_exploration,

"Fixed Exploration + Exploitation": rewards\_fixed\_explore,

"ε-Greedy (ε=0.1)": rewards\_epsilon\_greedy,

"UCB": rewards\_ucb

}

plot\_comparison(rewards\_dict, n\_steps)

# Calculate regret

regret\_pure\_exploitation = calculate\_regret(bandit, n\_steps, rewards\_pure\_exploitation)

regret\_pure\_exploration = calculate\_regret(bandit, n\_steps, rewards\_pure\_exploration)

regret\_fixed\_explore = calculate\_regret(bandit, n\_steps, rewards\_fixed\_explore)

regret\_epsilon\_greedy = calculate\_regret(bandit, n\_steps, rewards\_epsilon\_greedy)

regret\_ucb = calculate\_regret(bandit, n\_steps, rewards\_ucb)

# Print regrets

print(f"Regret - Pure Exploitation: {regret\_pure\_exploitation}")

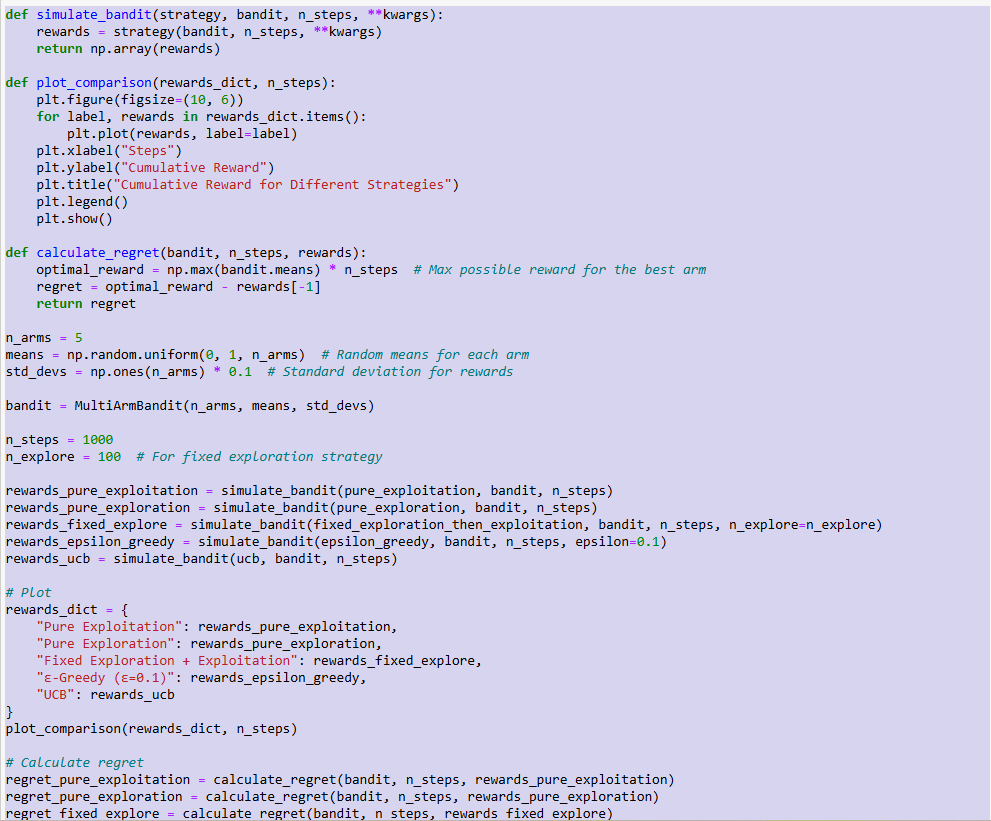
print(f"Regret - Pure Exploration: {regret\_pure\_exploration}")

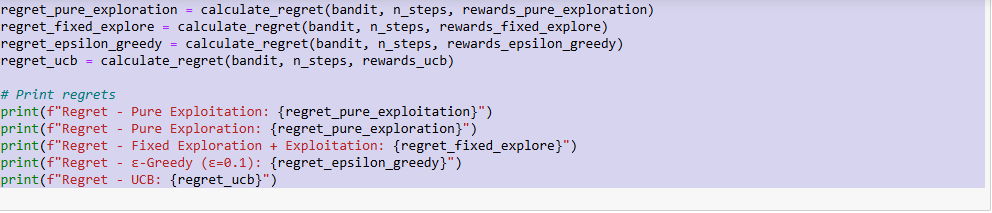
print(f"Regret - Fixed Exploration + Exploitation: {regret\_fixed\_explore}")

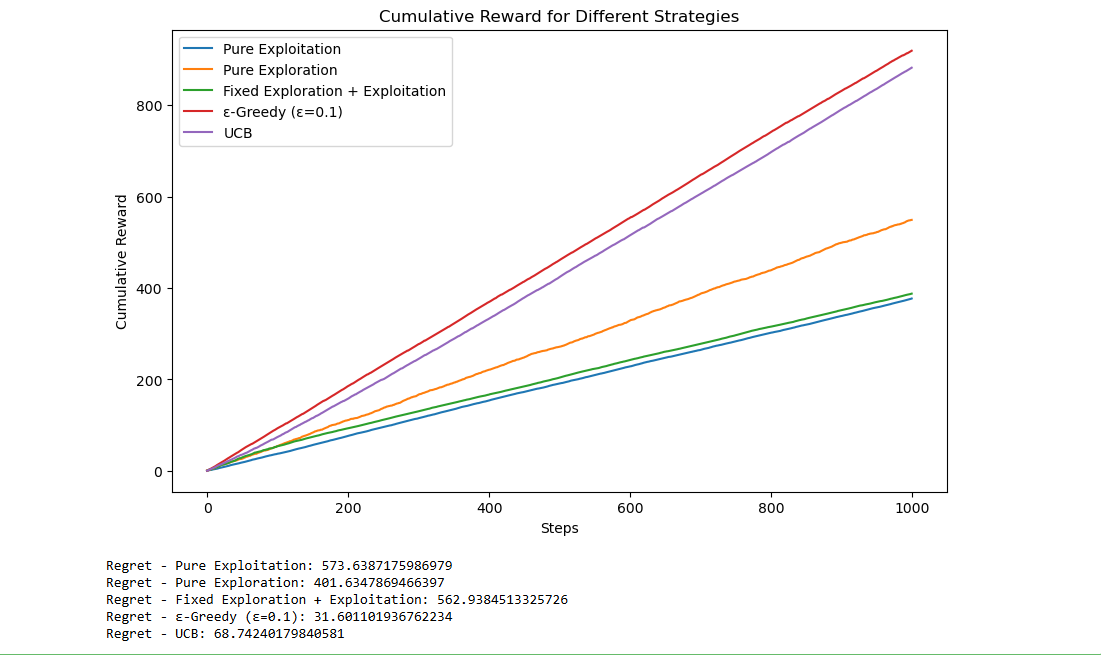
print(f"Regret - ε-Greedy (ε=0.1): {regret\_epsilon\_greedy}")

print(f"Regret - UCB: {regret\_ucb}")

Screenshot:





****

**B.2 Observations and Learning:**

 **Observation**: The **pure exploitation** strategy is highly efficient once enough data has been gathered to accurately estimate the rewards of each arm. However, it fails to explore any other arm once an arm is chosen as the optimal one. This can lead to suboptimal performance in scenarios where the environment changes or the initial estimates are inaccurate.

 **Learning**: Pure exploitation can be effective when you have high confidence in your estimates and have explored the arms sufficiently. However, in practice, it’s often better to balance exploration with exploitation to ensure robustness to uncertainty.

**B.3 Conclusion:**

 **Exploration vs. Exploitation**: The performance of any algorithm is largely dictated by how well it balances exploration and exploitation. Pure exploitation works well when the environment is known and stable but fails in more dynamic or uncertain environments.

 **Effectiveness of ε-Greedy and UCB**: Both the **ε-Greedy** and **UCB** algorithms effectively address the exploration-exploitation trade-off, with UCB being particularly advantageous in dynamic or less predictable environments. Proper parameter tuning (e.g., the value of ε) is critical for the performance of these algorithms.