Nicolas Stupak – 405178

# Bandycki Streaming

## Opis ćwiczenia

Celem laboratorium było przetestowanie różnych podejść do rozwiązywania problemu k-rękich bandytów. Nasz wariant tego problemu polegał na przewidywaniu, który z utworów może stać się przyszłym hitem. Nagroda 1 oznacza słuchacza, który przesłuchał go w całości, nagroda 0 oznacza, że utwór został pominięty. Do zaimplementowania były następujące algorytmy:

- Explore-Then Commit

- Greedy

- UCB1

- Gradient

- Thompson Sampling

## Implementacja metod

### Explore-Then Commit

1. class ExploreThenCommitLearner(BanditLearner):

2. def \_\_init\_\_(self, m: int = 10):

3. self.name = "ETC"

4. self.color = "blue"

5. self.arms: list[str] = []

6. self.time\_step = 0

7. self.k\_arms: int = None

8. self.m = m

9. self.arms\_stats: dict[str] = {}

10.

11. def reset(self, arms: list[str], time\_steps: int):

12. self.arms = arms

13. self.k\_arms = len(arms)

14. self.arms\_stats = {arm: 0 for arm in self.arms}

15. self.time\_step = 0

16.

17. def pick\_arm(self) -> str:

18. if self.time\_step < self.m \* self.k\_arms:

19. self.time\_step += 1

20. return self.arms[(self.time\_step % self.k\_arms)]

21. else:

22. return max(self.arms\_stats, key=self.arms\_stats.get)

23.

24. def acknowledge\_reward(self, arm: str, reward: float) -> None:

25. self.arms\_stats[arm] += reward

26.

### Greedy

1. class GreedyLearner(BanditLearner):

2. def \_\_init\_\_(self, strategy=None, Q=None, epsilon=None):

3. self.color = "purple"

4. self.arms: list[str] = []

5. self.time\_step = 0

6. self.k\_arms: int = None

7. self.arms\_stats: dict[dict[int, int]] = {}

8.

9. if not strategy:

10. self.strategy = random.choice(["Greedy", "ε-Greedy", "Optimistic-Greedy"])

11. else:

12. self.strategy = strategy

13.

14. # Initialize based on the chosen strategy

15. if self.strategy == "Greedy":

16. self.init\_pure\_greedy()

17. elif self.strategy == "ε-Greedy":

18. self.init\_epsilon\_greedy()

19. else:

20. self.init\_optimistic\_greedy()

21.

22. def init\_pure\_greedy(self):

23. self.name = self.strategy

24.

25. def init\_epsilon\_greedy(self, epsilon=np.random.rand()):

26. self.name = self.strategy

27. self.epsilon = epsilon

28.

29. def init\_optimistic\_greedy(self, Q=random.randint(1, 11)):

30. self.name = self.strategy

31. self.Q = Q

32.

33. def reset(self, arms: list[str], time\_steps: int):

34. self.arms = arms

35. self.k\_arms = len(arms)

36. self.arms\_stats = {

37. arm: {

38. "n\_pulls": 0,

39. "mean": self.Q if self.strategy == "Optimistic-Greedy" else 0,

40. }

41. for arm in self.arms

42. }

43. self.time\_step = 0

44.

45. def pick\_arm(self) -> str:

46. if self.time\_step < self.k\_arms:

47. self.time\_step += 1

48. return self.arms[(self.time\_step - 1)]

49. else:

50. if self.strategy != "ε-Greedy":

51. return max(

52. self.arms\_stats, key=lambda arm: self.arms\_stats[arm]["mean"]

53. )

54. else:

55. return np.random.choice(

56. [

57. random.choice(self.arms),

58. max(

59. self.arms\_stats,

60. key=lambda arm: self.arms\_stats[arm]["mean"],

61. ),

62. ],

63. p=[self.epsilon, 1 - self.epsilon],

64. )

65.

### UCB1

1. class UpperConfidenceBoundLearner(BanditLearner):

2. def \_\_init\_\_(self, c: int = 2):

3. self.name = "UCB1"

4. self.color = "red"

5. self.arms: list[str] = []

6. self.time\_step = 0

7. self.k\_arms: int = None

8. self.c = c

9. self.arms\_pulls: dict[str] = {}

10. self.arms\_rewards: dict[str] = {}

11. self.arms\_means: dict[str] = {}

12. self.arms\_ubc: dict[str] = {}

13.

14. def reset(self, arms: list[str], time\_steps: int):

15. self.arms = arms

16. self.k\_arms = len(arms)

17. self.arms\_pulls = {arm: 0 for arm in self.arms}

18. self.arms\_rewards = {arm: 0 for arm in self.arms}

19. self.arms\_means = {arm: 0 for arm in self.arms}

20. self.arms\_ubc = {arm: 0 for arm in self.arms}

21. self.time\_step = 0

22.

23. def pick\_arm(self) -> str:

24. self.time\_step += 1

25. if self.time\_step <= self.k\_arms:

26. return self.arms[self.time\_step - 1]

27. else:

28. self.calculate\_ubc()

29. return max(self.arms\_ubc, key=self.arms\_ubc.get)

30.

31. def acknowledge\_reward(self, arm: str, reward: float) -> None:

32. self.arms\_rewards[arm] += reward

33. self.arms\_pulls[arm] += 1

34. self.arms\_means[arm] = self.arms\_rewards[arm] / self.arms\_pulls[arm]

35.

36. def calculate\_ubc(self):

37. for arm in self.arms:

38. if self.arms\_pulls[arm] == 0:

39. self.arms\_ubc[arm] = float("inf")

40. else:

41. self.arms\_ubc[arm] = self.arms\_means[arm] + (

42. self.c \* np.sqrt(np.log(self.time\_step) / self.arms\_pulls[arm])

43. )

44.

### Gradient

1. class GradientLearner(BanditLearner):

2. def \_\_init\_\_(self, alpha: float = 0.1, random\_alpha: bool = False):

3. self.name = "Gradient"

4. self.color = "cyan"

5. self.alpha = np.random.rand() if random\_alpha else alpha

6. self.arms: list[str] = []

7. self.time\_step: int = 0

8. self.reward\_baseline: float = 0.0

9. self.preferences: dict[str, float] = {}

10.

11. def reset(self, arms: list[str], time\_steps: int):

12. self.arms = arms

13. self.preferences = {arm: 0.0 for arm in self.arms}

14. self.time\_step = 0

15. self.reward\_baseline = 0.0

16.

17. def pick\_arm(self) -> str:

18. probabilities = softmax(

19. list(self.preferences.values())

20. ) # Compute action probabilities

21. return np.random.choice(self.arms, p=probabilities)

22.

23. def acknowledge\_reward(self, arm: str, reward: float) -> None:

24. """Update preferences based on the received reward"""

25. self.time\_step += 1

26. self.reward\_baseline += (1 / self.time\_step) \* (

27. reward - self.reward\_baseline

28. ) # Update baseline

29.

30. probabilities = softmax(

31. list(self.preferences.values())

32. ) # Compute probabilities before updating

33.

34. for i, a in enumerate(self.arms):

35. if a == arm:

36. self.preferences[a] += (

37. self.alpha

38. \* (reward - self.reward\_baseline)

39. \* (1 - probabilities[i])

40. )

41. else:

42. self.preferences[a] -= (

43. self.alpha \* (reward - self.reward\_baseline) \* probabilities[i])

44.

### Thompson Sampling

1. class ThompsonLearner(BanditLearner):

2. def \_\_init\_\_(self):

3. self.name = "Thompson"

4. self.color = "olive"

5. self.arms: list[str] = []

6. self.time\_step = 0

7. self.k\_arms: int = None

8. self.arms\_stats: dict[str] = {}

9.

10. def reset(self, arms: list[str], time\_steps: int):

11. self.arms = arms

12. self.k\_arms = len(arms)

13. self.arms\_stats = {arm: {"a": 1, "b": 1} for arm in self.arms}

14.

15. def pick\_arm(self) -> str:

16. return max(

17. self.arms,

18. key=lambda arm: beta.rvs(

19. self.arms\_stats[arm]["a"], self.arms\_stats[arm]["b"]

20. ),

21. )

22.

23. def acknowledge\_reward(self, arm: str, reward: float) -> None:

24. self.arms\_stats[arm]["a"] += reward

25. self.arms\_stats[arm]["b"] += 1 - reward

26.

## Realizacja eksperymentów

Realizacja eksperymentów polegała na uruchomieniu istniejącego już szkieletu kodu i badaniu, jak będą kumulowały się w czasie nagrody otrzymane przez poszczególne algorytmy.

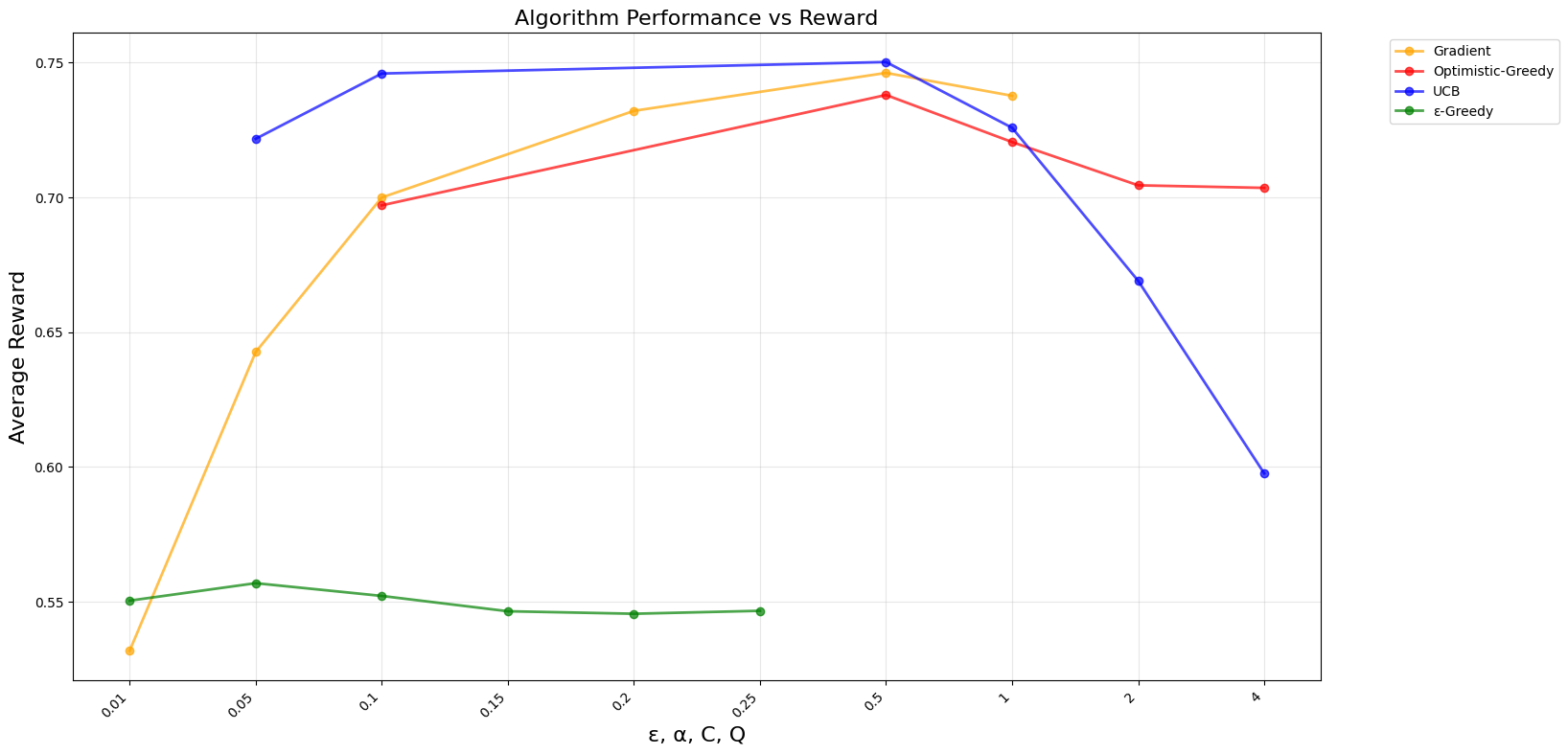
A graph with different colored lines

AI-generated content may be incorrect.

Obraz 1. Porównanie nagród zgromadzonych w czasie przez różne algorytmy

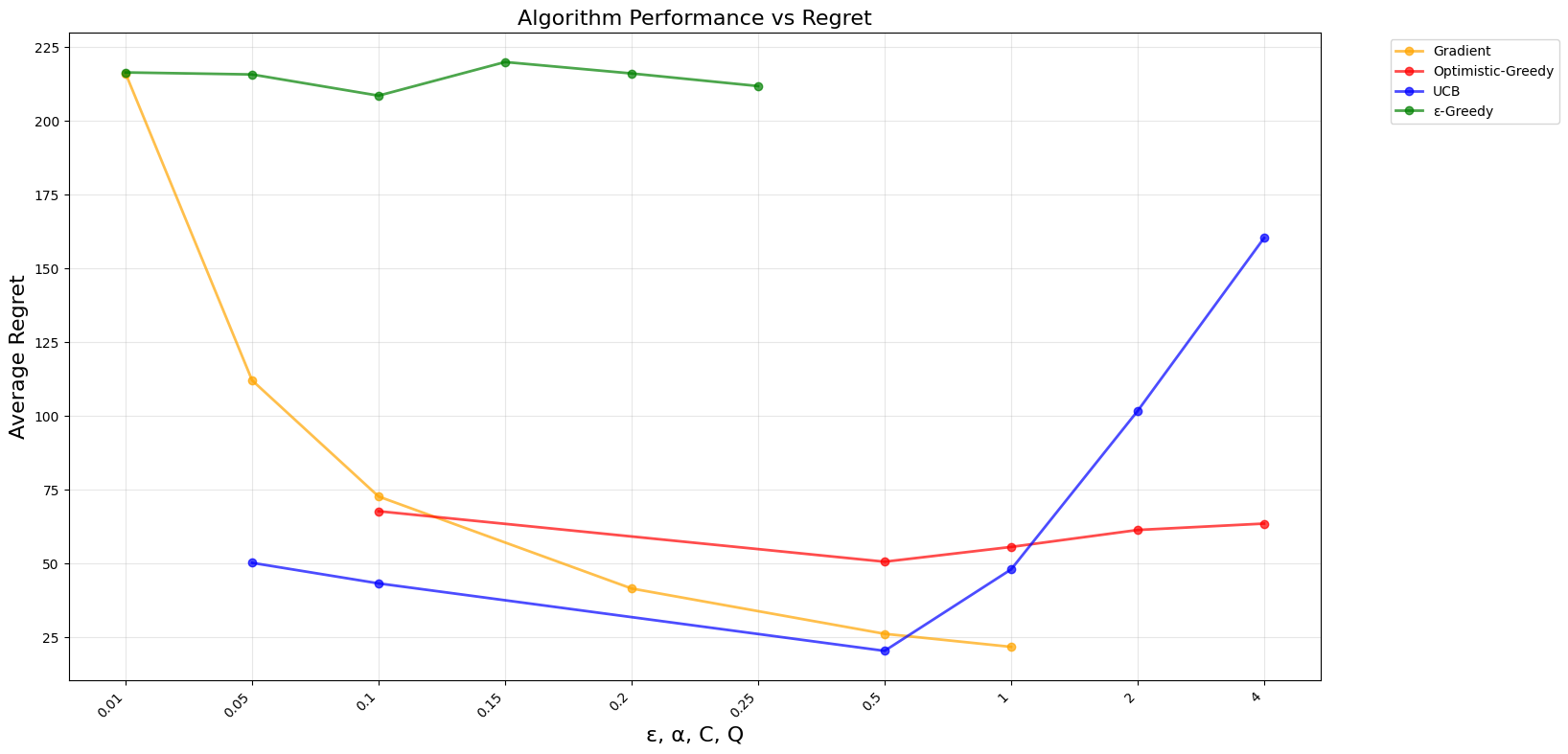
## Studium parametryczne

### Strategie vs Nagroda



Obraz 2. Studium parametryczne: Strategie vs Nagroda

### Strategie vs Regret

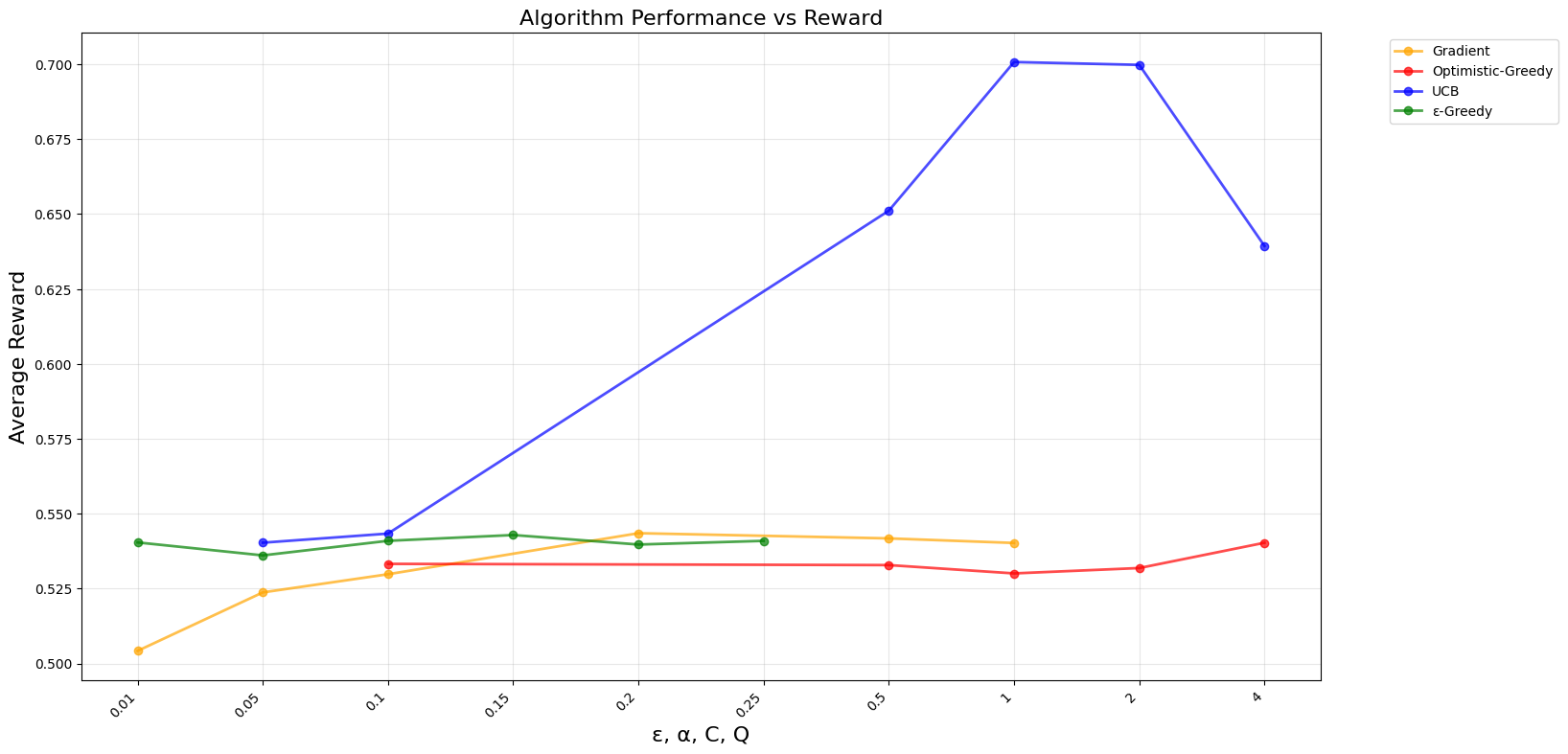


Obraz 3. Studium parametryczne: Strategie vs Regret

## Bandyci niestacjonarni

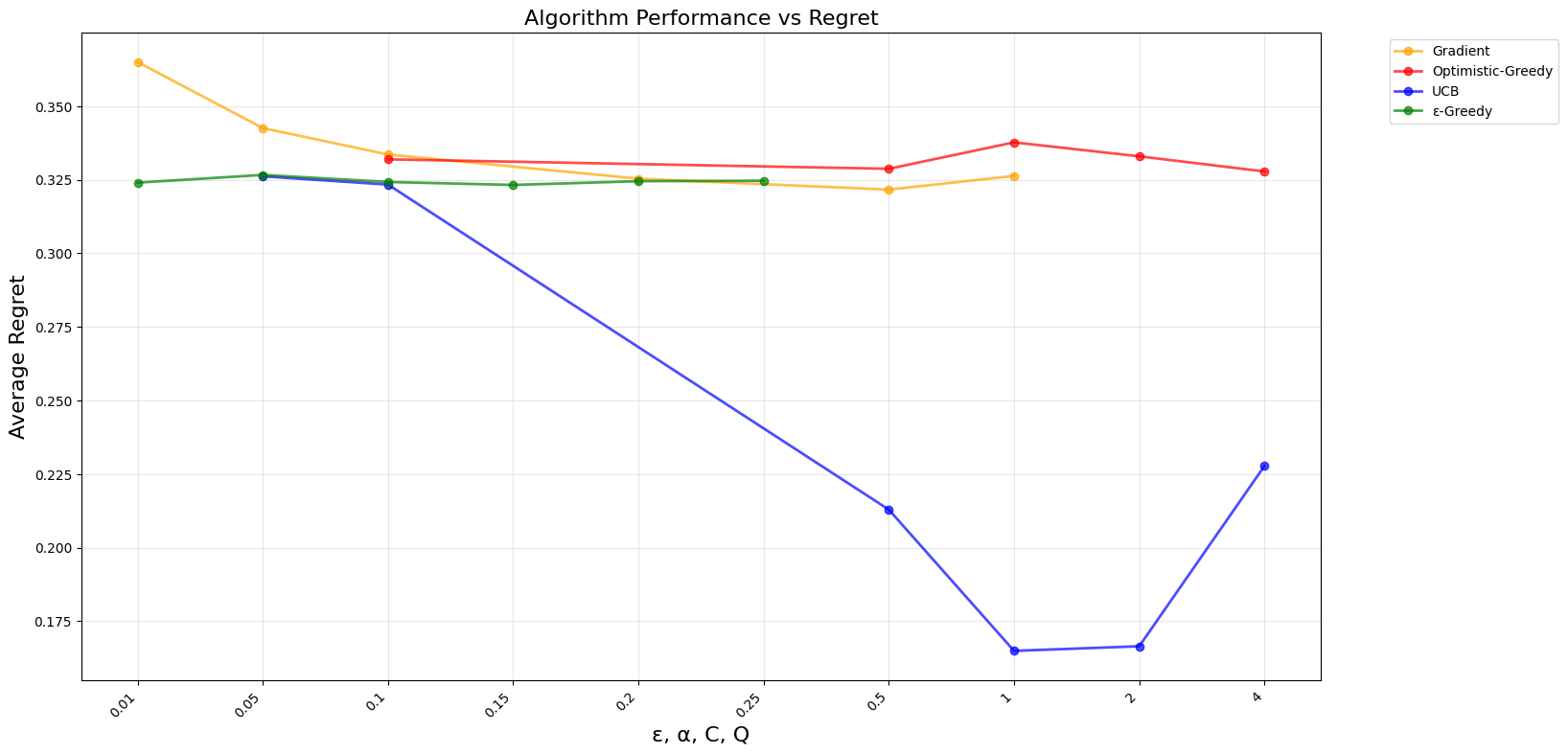
Dla wersji niestacjonarnej, w każdym kroku wartości nagród zostały zaktualizowane przy pomocy funkcji np.random.normal(0, 0.1, 1) o losową wartość z rozkładu normalnego o odchyleniu standardowym 0.1. Jeżeli po aktualizacji wartość nie mieściły się w przedziale [0,1], zostaje im przypisana wartość krańcowa przedziału przy użyciu np.clip(a\_min=0, a\_max=1).

### Strategie vs Nagroda



Obraz 4. Niestacjonarne studium parametryczne: Strategie vs Nagroda

### Strategie vs Regret



Obraz 5. Niestacjonarne studium parametryczne: Strategie vs Regret

## Wnioski

Zarówno w wersji stacjonarnej jak i niestacjonarnej algorytmem, który uzyskiwał najlepsze wyniki była strategia Upper Confidence Bound. Najlepszymi wartościami parametru **c** tego algorytmu były wartości 0.1 i 0.5 dla wersji stacjonarnej i 1 oraz 2 dla testu niestacjonarnego. Drugim najlepiej radzącym sobie algorytmem był Gradient, dla którego 0.5 oraz 1, okazały się najlepszymi wartościami parametru **learning rate**. Należy również zaznaczyć, że w teście niestacjonarnym wydajność i przewaga algorytmu gradientowego nad strategiami zachłannymi drastycznie spada. Najgorzej radzącymi sobie algorytmami były strategie zachłanne: e-greedy oraz optimistic-greedy. Co ciekawe, algorytm optimistic-greedy radził sobie całkiem nieźle w teście stacjonarnym, natomiast okazał się najgorszy w studium niestacjonarnym. Dla wersji niestacjonarnej, wyniki algorytmu e-greedy nieznacznie zwiększają się wraz z wzrostem parametru **e**, co ma sens, ponieważ częściej eksplorujemy więc możemy poznać nowy rozkład. Wszystkie algorytmy poradziły sobie gorzej w teście niestacjonarnym.