Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

UMA
Dokumentacja końcowa

Zastosowanie uczenia ze wzmocnieniem do ustawienia sposobu mutacji i współczynnika skalowania (F) w algorytmie ewolucji różnicowej

Natalia Iwańska, Filip Szczygielski

Warszawa, 2024

1. Polecenie

Zastosowanie uczenia ze wzmocnieniem do ustawienia sposobu mutacji i współczynnika skalowania (F) w algorytmie ewolucji różnicowej. Niech stanem będzie procent sukcesów oraz średnia odległość pomiędzy osobnikami w aktualnej populacji, a akcją wybór wskazanych parametrów algorytmu. O sukcesie mówimy wtedy, gdy mutant przeżyje sukcesję. Liczbę sukcesów należy zsumować na przestrzeni X iteracji, gdzie X to np. 20, po czym podzielić przez liczbę prób. Zarówno stany jak i akcje można zdyskretyzować. Funkcje do optymalizacji należy pobrać z benchmarku CEC2017.

2. Opis algorytmów

Do realizacji projektu zaimplementowaliśmy algorytm ewolucji różnicowej oraz algorytm uczenia ze wzmocnieniem. Ten pierwszy jest algorytmem optymalizacji podobnym to klasycznej wersji algorytmu ewolucyjnego z tą różnicą, że na etapie krzyżowania nowe osobniki generowane są poprzez kombinowanie różnic pomiędzy wybranymi osobnikami.

2.1. Algorytm ewolucji różnicowej

Pseudokod klasycznego algorytmu ewolucji różnicowej

```
P = inicjacja()
ocena( P )
while( !koniec )
  for( i in 1:mu )
    Os_wybr = selekcjaOs( P )
    Os_k = losuj( P )
    Os_l = losuj( P )
    T_i = Os_wybr + F*(Os_k-Os_l)
    Os_j = krzyzow( T_i, P[i] )
    if( ocena( Os_j )<= ocena( P[i] ) )
      P[i] = Os_j
```

Rys. 2.1.

Parametry początkowe

— mu - rozmiar populacji
— Metoda generowania populacji początkowej
— Warunek stopu - (maksymalna liczba iteracji)
— F - parametr skalowania różnicowego
— Metoda selekcji rodziców
— Typ krzyżowania
— Liczba par różnicowanych punktów
— CR - prawdopodobieństwo krzyżowania

Inicjacja algorytmu
Polega na wygenerowaniu populacji początkowej, odbywa się to poprzez losowanie współrzędnych osobników. Następnie dopóki nie osiągneliśmy maksymalnej liczby iteracji wykonujemy pętle w której po kolei:

— dokonujemy selekcji jednego osobnika - Os_wybr. Selekcji można dokonywać na dwa sposoby: losowy (rand), albo najlepszy (best).

— losujemy parę kolejnych osobników Os_k, Os_l. (lub dwie pary)

— do wybranego osobnika Os_wybr dodajemy różnicę między osobnikami Os_k i Os_l przemnożonymi przez parametr skalowania różnicowego - F tworząc nowego osobnika T_i - mutanta.

— krzyżujemy w sposób dwumianowy ze sobą T_i oraz i-tego osobnika w populacji tworząc Os_j

— oceniamy na podstawie wartości funkcji i-tego osobnika oraz Os_j i jeśli osobnik Os_j okaże się lepszy to zastępuje on i-tego osobnika.

— następnie całość powtarzamy dla kolejnego (i) osobnika w populacji.

Naszym zadaniem jest dostosowanie metody mutacji (selekcja osobnika Os_wybr oraz Liczba par różnicowanych punktów) oraz współczynnika F, za pomocą uczenia ze wzmocnieniem.

W tym celu zaimplementowaliśmy również algorytm q-learning.

Data: t_{max}, γ, β
Result: Q
begin
  $t \leftarrow 0$
  $Q_0 \leftarrow$zainicjuj
  while $t \leq t_{max}$ **do**
    $a_t \leftarrow$wybierz akcję(x_t, Q_t)
    $r_t, x_{t+1} \leftarrow$wykonaj akcję a_t
    $\Delta \leftarrow r_t + \gamma \max_a Q_t(x_{t+1}, a) - Q_t(x_t, a_t)$
    $Q_{t+1}(x_t, a_t) \leftarrow Q_t(x_t, a_t) + \beta\Delta$
    $t \leftarrow t + 1$
  end
end

Rys. 2.2.

Parametry początkowe

— epsilon - $\epsilon = 0{,}5$
— Współczynnik dyskontowania - $\gamma = 0{,}9$
— Współczynnik uczenia się - $\beta = 0{,}5$
— liczba iteracji - $t_{max} = 1000$

Inicjacja algorytmu

Polega na wygenerowaniu wartości funkcji Q dla wszystkich stanów i akcji na początkowe zerowe wartości. Następnie dopóki nie przekroczymy liczby iteracji t_{max} wykonujemy pętle w której po kolei:

— wybieramy akcję na podstawie funkcji wartości Q.

— wykonujemy akcję i obserwujemy wyniki (nowy stan - x_{t+1} i nagroda - r_t)

— aktualizujemy funkcję wartości Q.

Przy wyborze akcji, agent z prawdopodobieństwem 1 - ϵ wybiera ją w sposób zachłanny, czyli na podstawie maksymalnej wartości Q (eksploatacja). W przypadku kilku akcji o maksymalnej wartości, wybierana jest losowa z nich. Z prawdopodobieństwem ϵ zaś, agent wybiera losową z możliwych w danym stanie akcję (eksploracja).

Algorytm ten uczy się podejmować decyzje w środowisku w celu maksymalizacji nagród. Naszym środowiskiem jest zbiór stanów, gdzie pojedynczy stan reprezentuje informację o procencie sukcesów oraz średniej odległości pomiędzy osobnikami w aktualnej populacji. Między stanami algorytm przechodzi wykonując akcje, w naszym przypadku są to wybory wskazanych parametrów algorytmu ewolucji różnicowej. Program dąży do maksymalizacji swojej nagrody czyli ilości sukcesów przeżycia mutantów w populacji.

3. Implementacje

Ostateczne implementacje wcześniej opisanych algorytmów i klas pomocniczych.
Ewolucja różnicowa:

```
class DifferentialEvolution:
    def __init__(
    self,
    objective_fun,
    popul_size,
    crossover_rate,
    F,
    max_iterations,
    bounds,
    dimension,
    selection,
    num_diff,
    train
    ):
        self.objective_fun = objective_fun
        self.popul_size = popul_size
        self.crossover_rate = crossover_rate
        self.F = F
        self.max_iterations = max_iterations
        self.bounds = bounds
        self.num_params = dimension
        self.selection = selection
        self.num_diff = num_diff
        self.population = []
        self.obj_val = []
        self.train = train

    def initialize_popul(self):
        self.population = np.random.uniform(
            low=self.bounds[0],
            high=self.bounds[1],
            size=(self.popul_size, self.num_params))
        self.obj_val = self.objective_fun(self.population)

    def mutate(self, popul, best_from_popul):
        mutated_popul = np.copy(popul)
        for i in range(self.popul_size):
            if self.selection == 'best':
                a = best_from_popul
            elif self.selection == 'rand':
                 a = np.random.randint(0, self.popul_size)
```

```
            if self.num_diff == 1:
                b, c = np.random.choice(self.popul_size, 2,
                                        replace=False)
                mutated_popul[i] = popul[a] + self.F * (popul[b]
                                                        - popul[c])
            elif self.num_diff == 2:
                b, c, d, f = np.random.choice(self.popul_size, 4,
                                              replace=False)
                mutated_popul[i] = popul[a] + self.F * (popul[b]
                        - popul[c]) + self.F * (popul[d] - popul[f])
            for coordinate in mutated_popul[i]:
                if coordinate < self.bounds[0]:
                    coordinate = self.bounds[0] + (self.bounds[0]
                                                   - coordinate)
                elif coordinate > self.bounds[1]:
                    coordinate = self.bounds[1] - (coordinate
                                                   - self.bounds[1])
        return mutated_popul



    def crossover(self, popul, mutated_popul):
        trial_popul = np.copy(popul)
        for i in range(self.popul_size):
            for j in range(self.num_params):
                if np.random.rand() < self.crossover_rate:
                    trial_popul[i, j] = mutated_popul[i, j]
        return trial_popul

    def avg_distance(self, popul):
        avg_distance = np.mean(np.linalg.norm(popul
                               - popul.mean(axis=0), axis=1))
        return avg_distance


    def evolve(self):
        success_rate = 0
        for _ in range(self.max_iterations):
            mutated_popul = self.mutate(self.population,
                                        np.argmin(self.obj_val))
            trial_popul = self.crossover(self.population,
                                         mutated_popul)
            obj_val_trial = self.objective_fun(trial_popul)

            for i in range(self.popul_size):

                if obj_val_trial[i] < self.obj_val[i]:
                    success_rate += 1
                    self.population[i] = trial_popul[i]
                    self.obj_val[i] = obj_val_trial[i]
            if not self.train:
```

```
                add_point(min(self.obj_val))
        avg_distance = self.avg_distance(self.population)
        success_rate = success_rate/(self.max_iterations
                                     *self.popul_size)
        state = (success_rate, avg_distance)

        return min(self.obj_val),
                    self.population[np.argmin(self.obj_val)], state

    def set_obj_function(self, fs):
        for i, f in enumerate(fs):
            if self.objective_fun == f:
                self.objective_fun = fs[(i + 1) % len(fs)]
                break
```

Q-learning:

```
class QLearningSolver:

    def __init__(
        self,
        observation_space: tuple,
        learning_rate: float = 0.1,
        gamma: float = 0.9,
        epsilon: float = 0.1,
    ):
        self.observation_space = observation_space
        self.learning_rate = learning_rate
        self.gamma = gamma
        self.epsilon = epsilon
        self.q_table =
            {(v1, v2, v3): 0 for v1 in observation_space[0] for v2 in
            observation_space[1] for v3 in observation_space[2]}

    def __call__(self, state: tuple, action: int) -> float:
        return self.q_table[(state[0], state[1], action)]

    def keys(self):
        return self.q_table.keys()

    def update(self, state: tuple, action: int, reward:
                                        float, next_state: tuple) -> None:
        current_q: float = self.q_table[(state[0], state[1], action)]
        temp_next = [self.q_table[(next_state[0], next_state[1], i)]
                        for i in range(6)]
        max_next_q: float = max(temp_next)
        new_q: float = current_q + self.learning_rate * (reward
                                + self.gamma * max_next_q - current_q)
        self.q_table[(state[0], state[1], action)] = new_q

    def get_best_action(self, state: tuple) -> int:
        filtered_keys = [k for k in self.q_table if k[0] == state[0]
```

```
                                                and k[1] == state[1]]

        max_value = max(self.q_table[k] for k in filtered_keys)
        best_keys = [k for k in filtered_keys if self.q_table[k]
                                                        == max_value]
        best_action = random.choice(best_keys)[2]
        return best_action
```

Funkcja do trenowania agenta QLearning:

```
def train_qsolver(q_solver: QLearningSolver,env_handler: Env,
                  learning_iter,fs, plots: bool, verbose:bool=False):
    num_episodes = learning_iter
    all_rewards = []
    for episode in range(num_episodes):
        env_handler.de.set_obj_function(fs)
        env_handler.reset()
        state = env_handler.get_first_state()
        closest_key_avg_dist = min(q_solver.keys(), key=lambda
                                           k: abs(k[1] - state[1]))
        state = (0, closest_key_avg_dist[1])
        done = False
        total_reward = 0
        while not done:

            if q_solver.epsilon > rd.random():
                action = env_handler.action_sample()
            else:
                action = q_solver.get_best_action(state)

            next_state, reward, done= env_handler.step(action)

            closest_key_suc_rate = min(q_solver.keys(), key=lambda
                                       k: abs(k[0] - next_state[0]))
            closest_key_avg_dist = min(q_solver.keys(), key=lambda
                                       k: abs(k[1] - next_state[1]))
            next_state = (closest_key_suc_rate[0],
                          closest_key_avg_dist[1])

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

            total_reward += reward
            state = next_state

        all_rewards.append(total_reward)
        if episode % 100 == 0 and verbose:
            print(f"Episode:{episode},Total Reward:{total_reward}")
    if plots:
        fig1 = plt.plot(all_rewards)
        plt.xlabel('Epizod')
        plt.ylabel('suma nagród')
        plt.title(f'Wykres sumy nagród uzyskanych w poszczególnych
```

```
--------iteracjach-uczących\nLiczba-iteracji-uczących:{num_episodes}')
        plt.show()
    return q_solver
```

Funkcja do testowania agenta QLearning:

```
def test_qsolver(q_solver: QLearningSolver,env_handler: Env):
    env_handler.reset()
    state = env_handler.get_first_state()
    closest_key_avg_dist = min(q_solver.keys(), key=lambda
                               k: abs(k[1] - state[1]))
    state = (0, closest_key_avg_dist[1])
    done = False
    while not done:
        action = q_solver.get_best_action(state)

        next_state, reward, done= env_handler.step(action)

        closest_key_suc_rate = min(q_solver.keys(), key=lambda
                                   k: abs(k[0] - next_state[0]))
        closest_key_avg_dist = min(q_solver.keys(), key=lambda
                                   k: abs(k[1] - next_state[1]))
        next_state = (closest_key_suc_rate[0],
                      closest_key_avg_dist[1])

        state = next_state


    return q_solver
```

Środowisko do uczenia agenta:

```
class Env:

    def __init__(self, func, population_size, iterrations_per_action,
                 dimensions, iterations_per_episode, train) -> None:
        self.func = func
        self.de = DifferentialEvolution(
            objective_fun=func,
            popul_size=population_size,
            crossover_rate=0.5,
            max_iterations=iterrations_per_action,
            bounds=(-100, 100),
            dimension=dimensions,
            F=0.5,
            selection = 'best', #'rand',
            num_diff = 1,  #2
            train = train
        )
        self.safed_de_state = None
        self.observation_space = self.create_obs_space(dimensions)
        self.actions_counter = 0
        self.prev_result = None
```

```
        self.iter_per_episode = iterations_per_episode

    def action(self, action_idx):
        match action_idx:
            case 0:
                if self.de.F < 1.81:
                    self.de.F += 0.2

            case 1:
                if self.de.F > 0.19:
                    self.de.F -= 0.2

            case 2:
                self.de.selection = 'best'

            case 3:
                self.de.selection = 'rand'

            case 4:
                self.de.num_diff = 1

            case 5:
                self.de.num_diff = 2

    def safe_de_state(self):
        self.safe_de_state = self.de.population

    def create_obs_space(self, dimensions):

        values1 = np.arange(0, 1.05, 0.05)
        values2 = np.linspace(0, 200*np.sqrt(dimensions), 20)
        values3 = np.arange(0, 6)
        return (values1, values2, values3)

    def reset(self):
        self.de.initialize_popul()
        self.actions_counter = 0

    def action_sample(self):
        return random.randint(0, 5)

    def step(self, action):
        done = False
        self.action(action)
        result, _, next_state = self.de.evolve()
        reward = (next_state[0]-0.2) * 10
        self.actions_counter += 1

        if self.actions_counter >= self.iter_per_episode:
            done = True
```

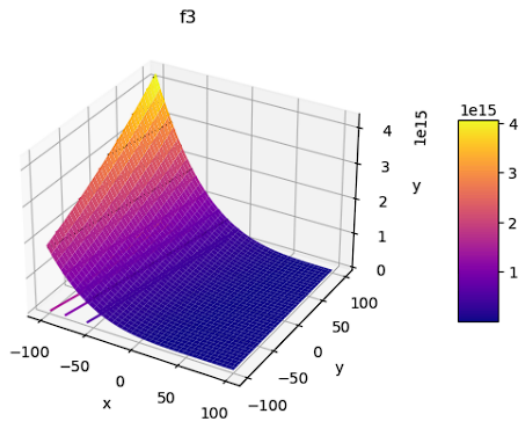
```
        return (next_state, reward, done)

    def get_first_state(self):
        avg_distance = self.de.avg_distance(self.de.population)
        return (0, avg_distance)
```

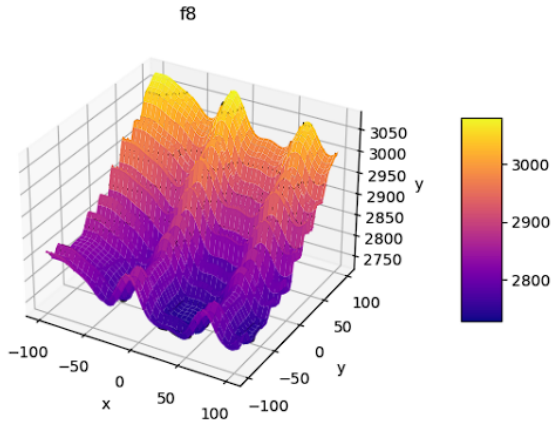
4. Dane do testowania

Nasze algorytmy przetestowaliśmy na 6 funkcjach z benchmarku CEC2017. Eksperymenty przeprowadzaliśmy na 2-wymiarowych i 10-wymiarowych funkcjach. Przy wyborze funkcji kierowaliśmy się tym żeby miały zróżnicowane kształty, ilość minimum lokalnych, rzędy wartości oraz punkty minimum. W dalszej części sprawozdania będziemy te funkcje odpowiednio nazywać: f1, f2, f3, f4, f5, f6.

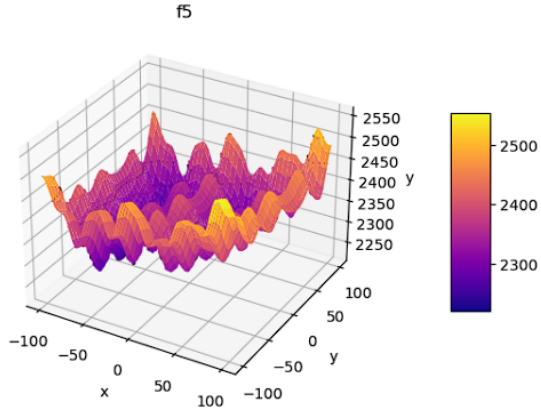
Wybrane funkcje:



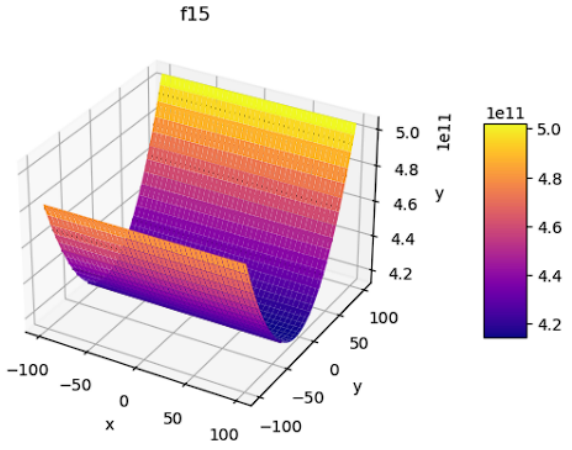
Rys. 4.1. Funkcja f1



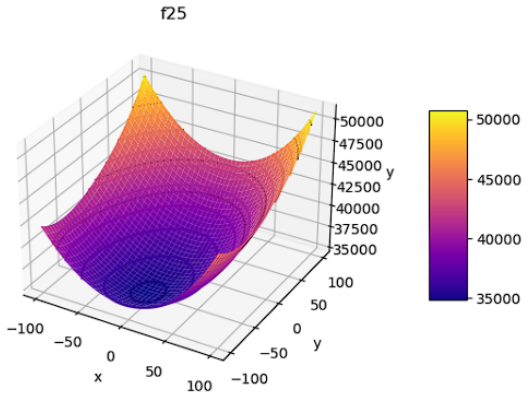
Rys. 4.2. Funkcja f2



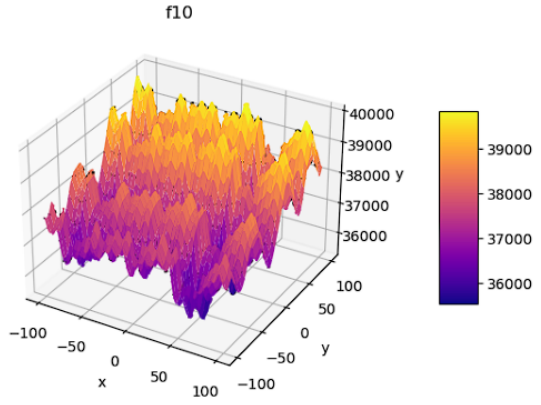
Rys. 4.3. Funkcja f3



Rys. 4.4. Funkcja f4



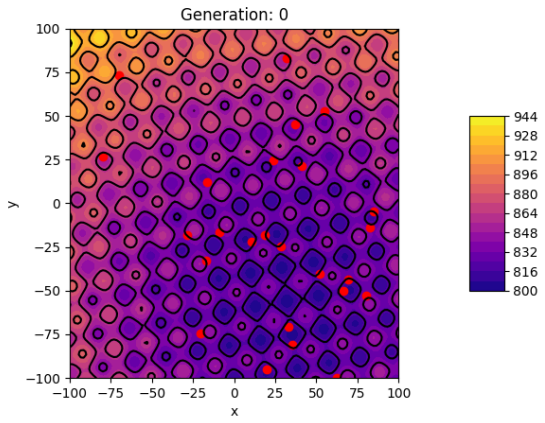
Rys. 4.5. Funkcja f5



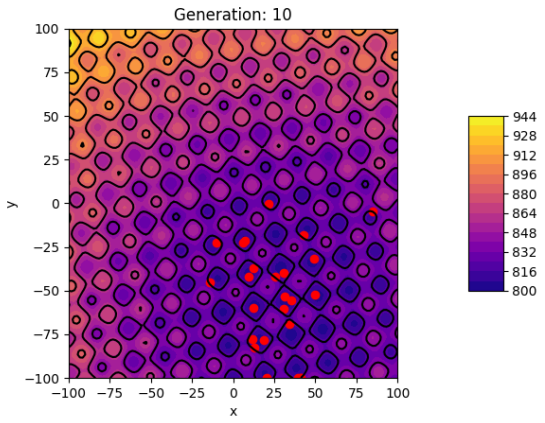
Rys. 4.6. Funkcja f6

5. Testowanie poprawności działania naszej implementacji ewolucji różnicowej

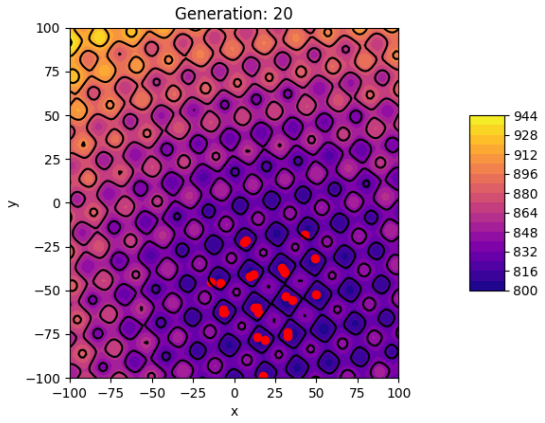
W celu przetestowania poprawności działania naszego algorytmu zwizualizowaliśmy jego działanie na tle funkcji, które minimalizowaliśmy. Kolejne rozkłady populacji podczas szukania minimum:



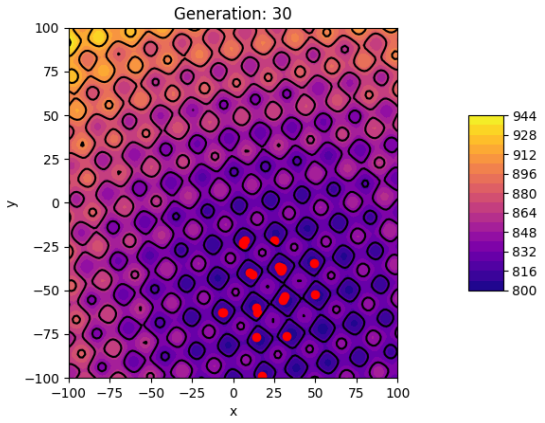
Rys. 5.1. Funkcja f2 początkowa populacja



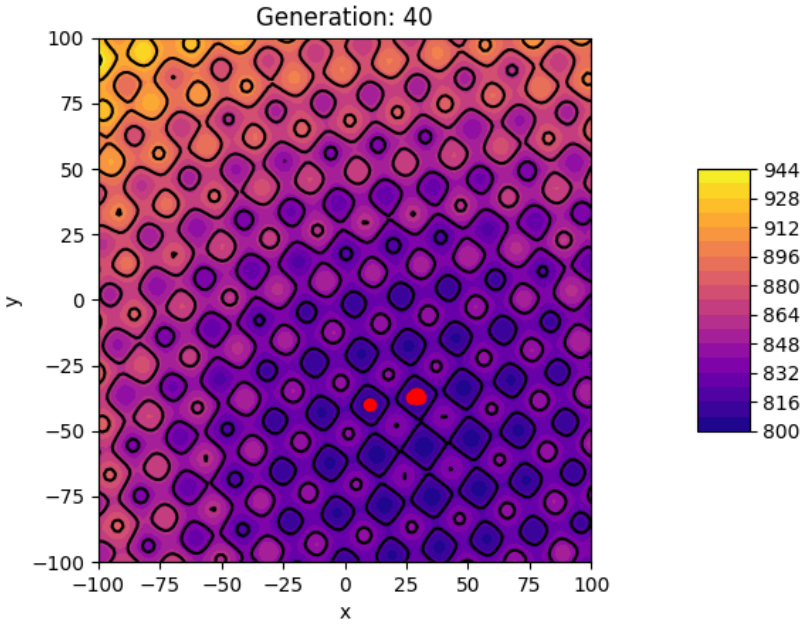
Rys. 5.2. Funkcja f2 populacja po 10 iteracjach



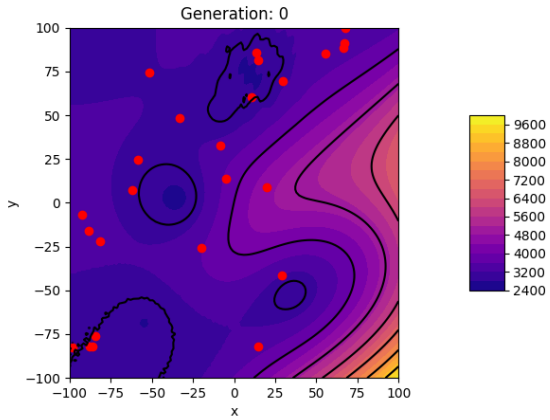
Rys. 5.3. Funkcja f2 populacja po 20 iteracjach



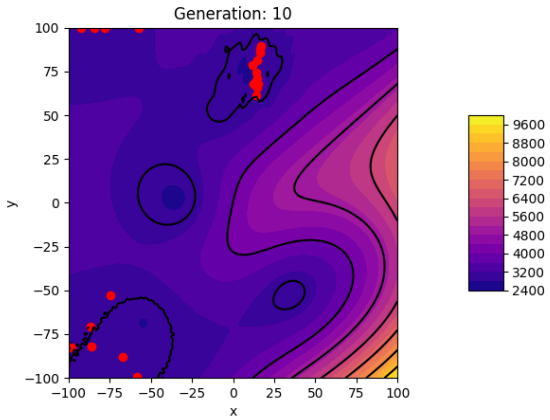
Rys. 5.4. Funkcja f2 populacja po 30 iteracjach



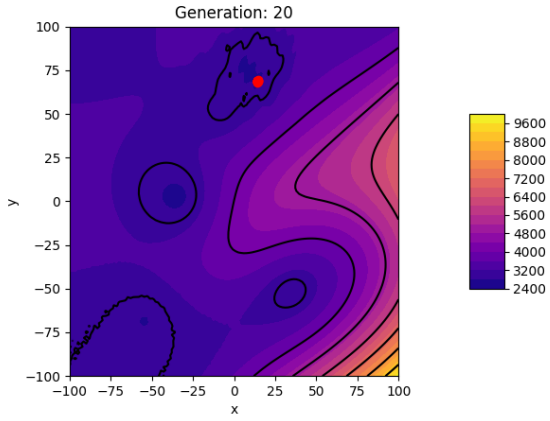
Rys. 5.5. Funkcja f2 populacja po 40 iteracjach



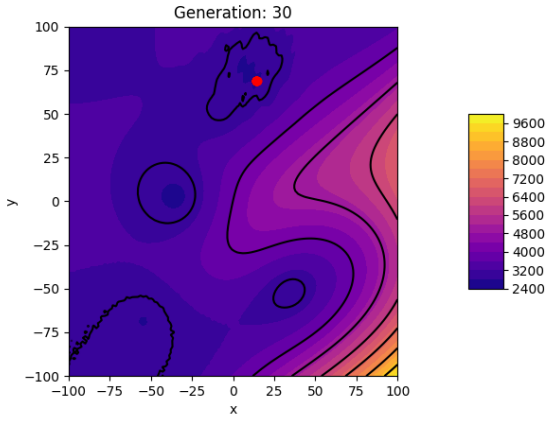
Rys. 5.6. Funkcja f5 początkowa populacja



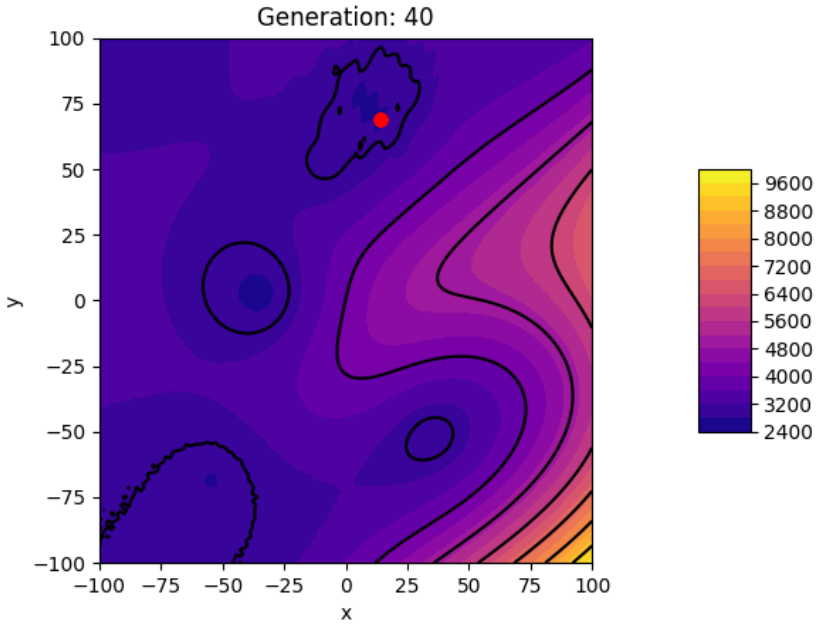
Rys. 5.7. Funkcja f5 populacja po 10 iteracjach



Rys. 5.8. Funkcja f5 populacja po 20 iteracjach



Rys. 5.9. Funkcja f5 populacja po 30 iteracjach



Rys. 5.10. Funkcja f5 populacja po 40 iteracjach

6. Testy i eksperymenty

6.1. Pliki Implementacyjne i Ich Opis

Implementację naszego rozwiązania załączyliśmy razem ze sprawozdaniem. Znajdują się tam pliki:

— differentialEvolution.py - w nim znajduje się implementacja algorytmu różnicowego.
— env.py - tutaj znajduje się implemenetacja środowiska wykorzystywanego w algorytmie QLearning
— main.py - tutaj uruchamiane jest testowanie funkcji bez Qlearningu i z, po wcześniejszym nauczeniu algorytmu.
— plot_conv.py - tutaj występują funkcje do rysowania krzywych zbiegania
— qLearningSolver.py - tutaj znajduje się główna klasa algorytmu QLearning
— training.py - tutaj uczony jest algorytm Qlearning a następnie zapisywany jest obiekt klasy zawierający nauczonego agenta.

6.2. Początkowe założenia

Docelowo wyuczony dzięki procesowi uczenia ze wzmocnieniem agent powinien poprawnie ustawiać metodę mutacji jak i współczynnik skalujący F, tak aby algorytm ewolucyjny uzyskał optymalny wynik. Algorytm będziemy uruchamiać wiele razy, tak aby otrzymać pewny wynik. W naszym przypadku miara jakości jest prosta do określenia i będzie to zdolność algorytmu do osiągnięcia minimum globalnego, będziemy ją oceniać głównie na podstawie krzywej uczenia.

6.3. Podział testów

Do eksperymentów przygotowaliśmy czterech agentów. Wszyscy byli uczeni na funkcjach f1, f2 i f3, a zmienialiśmy ilość wymiarów funkcji: 2 lub 10 oraz ilość iteracji uczących: 1000 lub 5000. Agentów będziemy rozróżniać korzystając z zapisu: Q(wymiar funkcji na których był uczony, ilość iteracji uczących). Następnie każdy z agentów porównywaliśmy z działaniem zwykłego algorytmu ewolucji różnicowej.

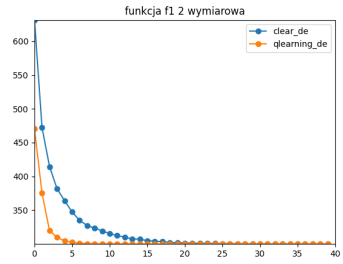
6.4. Eksperymenty

Dla wszystkich eksperymentów utrzymywaliśmy taki sam rozmiar populacji: 100, ograniczenia funkcji: (-100, 100), a wszystkie wykresy przedstawiają jaka w każdej iteracji była najlepszaw wartość optymalizacji. Ze względu na dużą ilość testów zdjęcia zostały wrzucone w mały rozmiarze i są mało czytelne, jednak postanowiliśmy tego nie zmieniać, ponieważ najważniejsza jest relacja między dwiema krzywymi a to bez problemu jesteśmy w stanie odczytać (na każdym wykresie niebieski kolor odpowiada ewolucji różnicowej a pomarańczowy ewolucji różnicowej ze strojeniem parametrów przy pomocy agenta QLearningu).

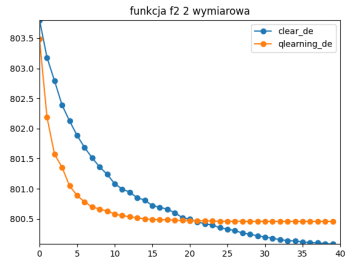
6.4.1. Eksperymenty na agencie Q(2, 1000)

W pierwszej kolejności porównaliśmy działanie algorytmów z agentem uczonym na funkcjach 2-wymiarowych przez 1000 iteracji.

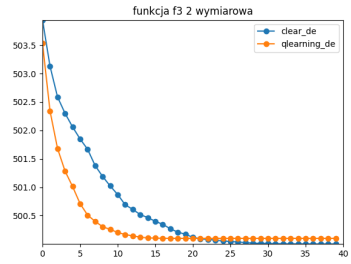
Testy na funkcjach 2-wymiarowych:



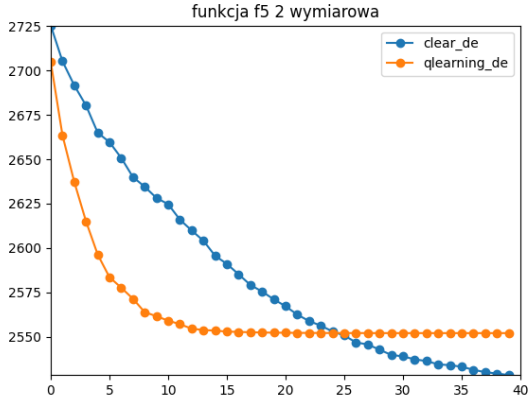
Rys. 6.1. Szukanie minimum funkcji f1 przez Q(2, 1000)



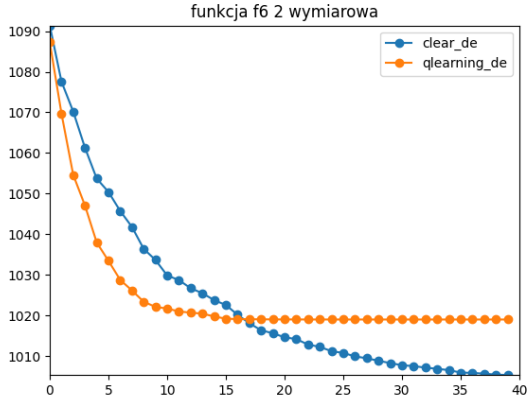
Rys. 6.2. Szukanie minimum funkcji f2 przez Q(2, 1000)



Rys. 6.3. Szukanie minimum funkcji f3 przez Q(2, 1000)

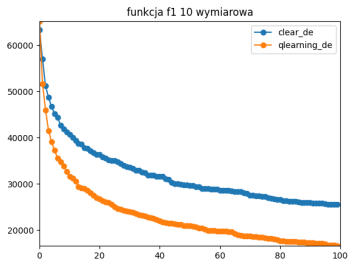


Rys. 6.4. Szukanie minimum funkcji f5 przez Q(2,

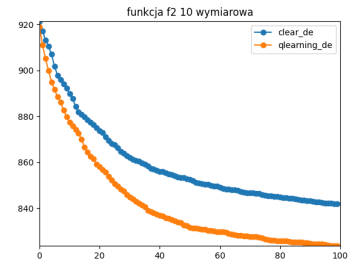


Rys. 6.5. Szukanie minimum funkcji f6 przez Q(2, 1000)

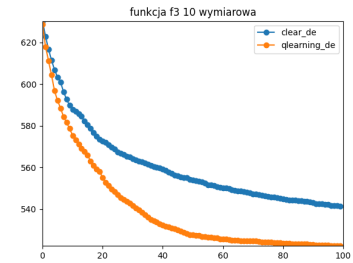
Testy na funkcjach 10-wymiarowych:



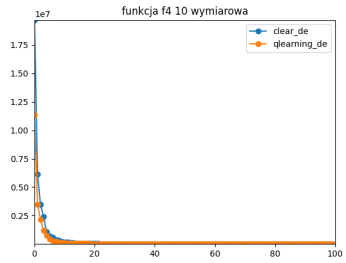
Rys. 6.6. Szukanie minimum funkcji f1 przez Q(2, 1000)



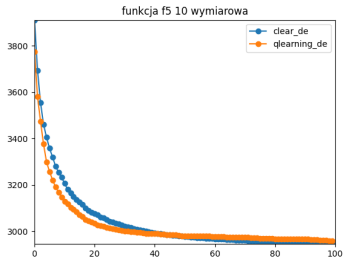
Rys. 6.7. Szukanie minimum funkcji f2 przez Q(2, 1000)



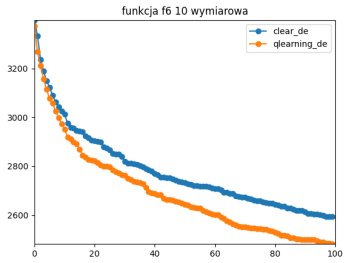
Rys. 6.8. Szukanie minimum funkcji f3 przez Q(2, 1000)



Rys. 6.9. Szukanie minimum funkcji f4 przez Q(2, 1000)



Rys. 6.10. Szukanie minimum funkcji f5 przez Q(2, 1000)

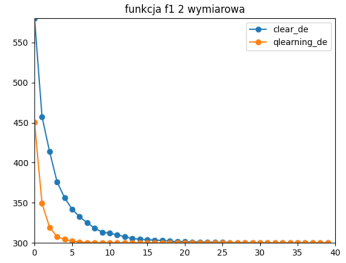


Rys. 6.11. Szukanie minimum funkcji f6 przez Q(2, 1000)

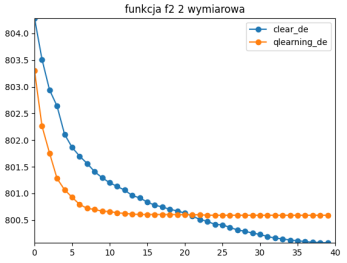
6.4.2. Eksperymenty na agencie Q(2, 5000)

Następnie przetestowaliśmy agenta uczonego na funkcjach dwuwymiarowych przez 5000 iteracji.

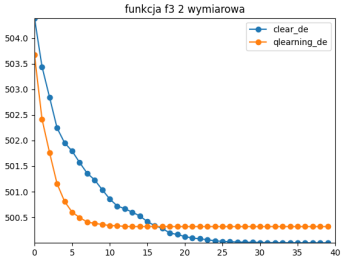
Testy na funkcjach 2-wymiarowych:



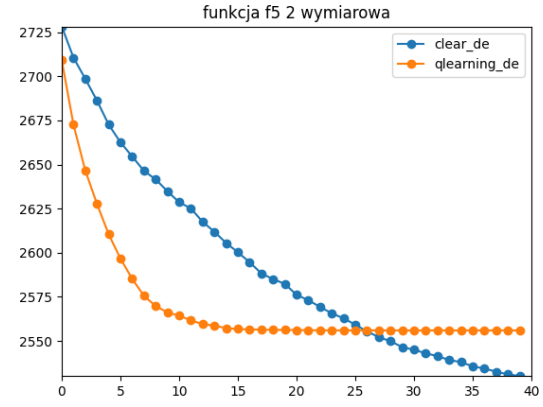
Rys. 6.12. Szukanie minimum funkcji f1 przez Q(2, 5000)

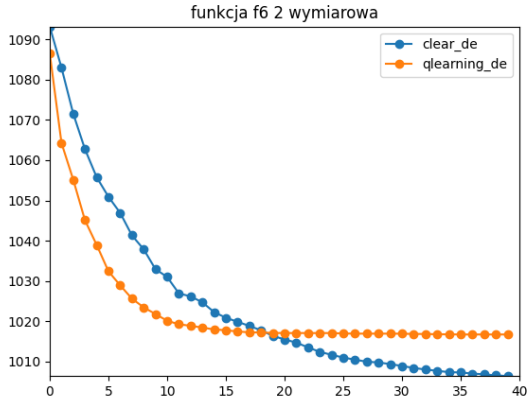


Rys. 6.13. Szukanie minimum funkcji f2 przez Q(2, 5000)



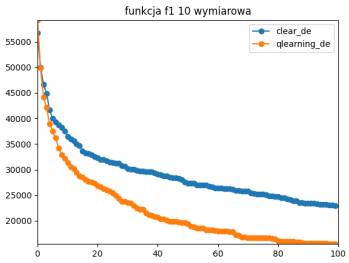
Rys. 6.14. Szukanie minimum funkcji f3 przez Q(2, 5000)



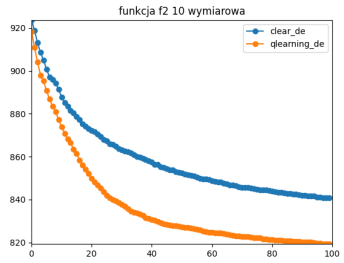


Rys. 6.15. Szukanie minimum funkcji f5 przez Q(2,Rys. 6.16. Szukanie minimum funkcji f6 przez Q(2, 5000)

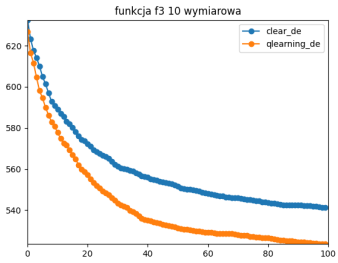
Testy na funkcjach 10-wymiarowych:



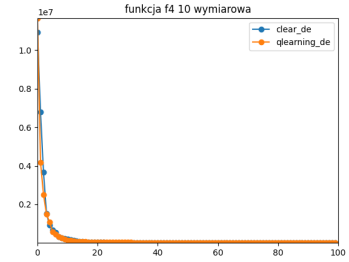
Rys. 6.17. Szukanie minimum funkcji f1 przez Q(2, 5000)



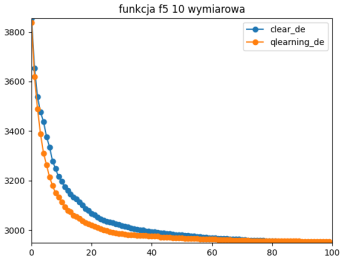
Rys. 6.18. Szukanie minimum funkcji f2 przez Q(2, 5000)



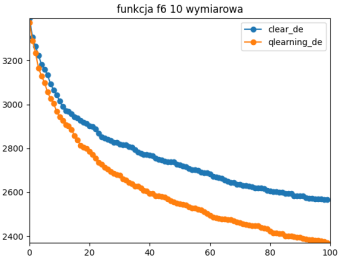
Rys. 6.19. Szukanie minimum funkcji f3 przez Q(2, 5000)



Rys. 6.20. Szukanie minimum funkcji f4 przez Q(2, 5000)



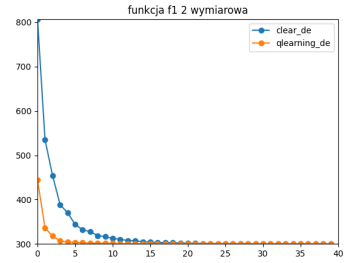
Rys. 6.21. Szukanie minimum funkcji f5 przez Q(2, 5000)



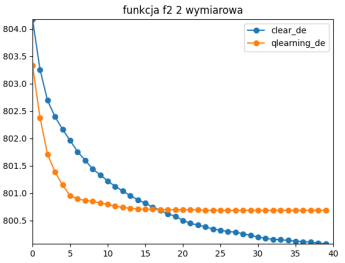
Rys. 6.22. Szukanie minimum funkcji f6 przez Q(2, 5000)

6.4.3. Eksperymenty na agencie Q(10, 1000)

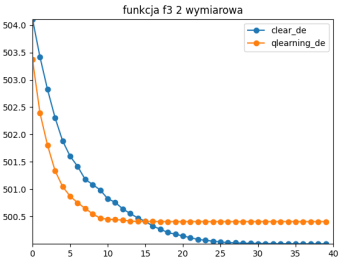
Kolejny był agent uczony na funkcjach 10-wymiarowych przez 1000 iteracji.
Testy na funkcjach 2-wymiarowych:



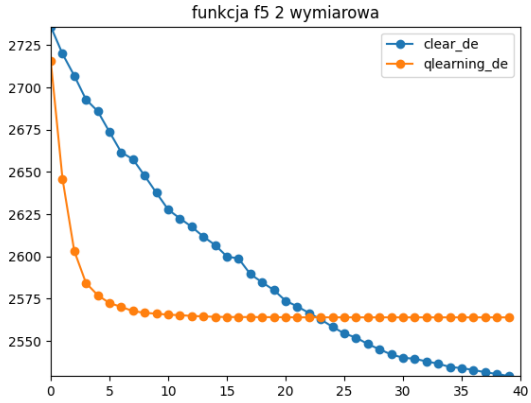
Rys. 6.23. Szukanie minimum funkcji f1 przez Q(10, 1000)



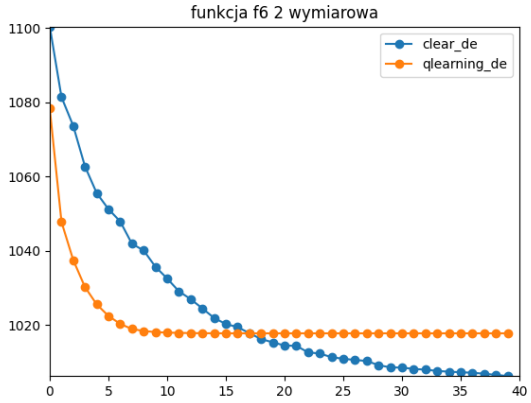
Rys. 6.24. Szukanie minimum funkcji f2 przez Q(10, 1000)



Rys. 6.25. Szukanie minimum funkcji f3 przez Q(10, 1000)

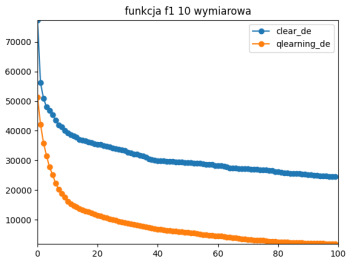


Rys. 6.26. Szukanie minimum funkcji f5 przez Q(10, 1000)

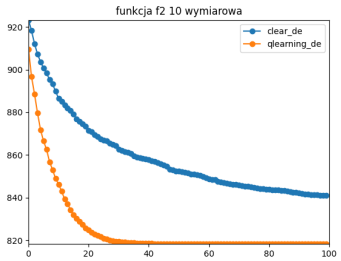


Rys. 6.27. Szukanie minimum funkcji f6 przez Q(10, 1000)

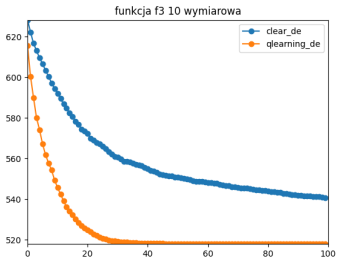
Testy na funkcjach 10-wymiarowych:



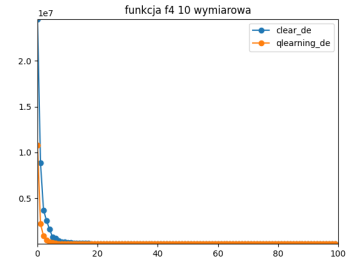
Rys. 6.28. Szukanie minimum funkcji f1 przez Q(2, 1000)



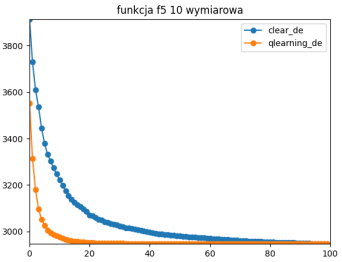
Rys. 6.29. Szukanie minimum funkcji f2 przez Q(2, 1000)



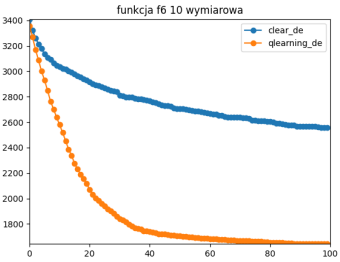
Rys. 6.30. Szukanie minimum funkcji f3 przez Q(2, 1000)



Rys. 6.31. Szukanie minimum funkcji f4 przez Q(2, 1000)



Rys. 6.32. Szukanie minimum funkcji f5 przez Q(2, 1000)

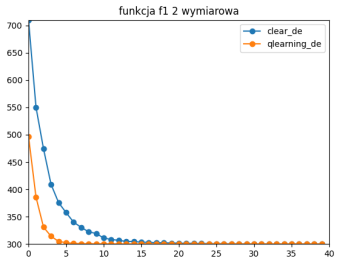


Rys. 6.33. Szukanie minimum funkcji f6 przez Q(2, 1000)

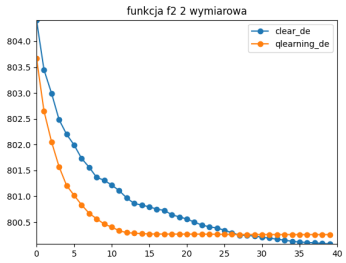
6.4.4. Eksperymenty na agencie Q(10, 5000)

W ostaniej kolejności przetestowaliśmy agenta uczonego na funkcjach 10-wymiarowych przez 5000 iteracji.

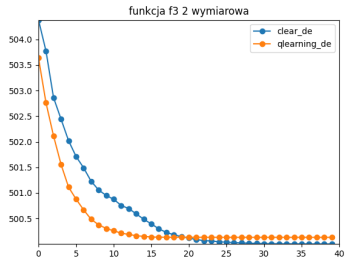
Testy na funkcjach 2-wymiarowych:



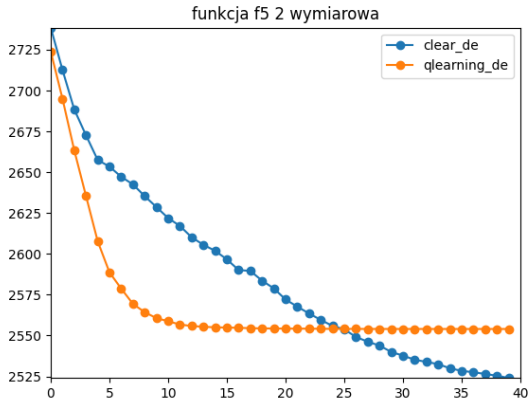
Rys. 6.34. Szukanie minimum funkcji f1 przez Q(10, 5000)



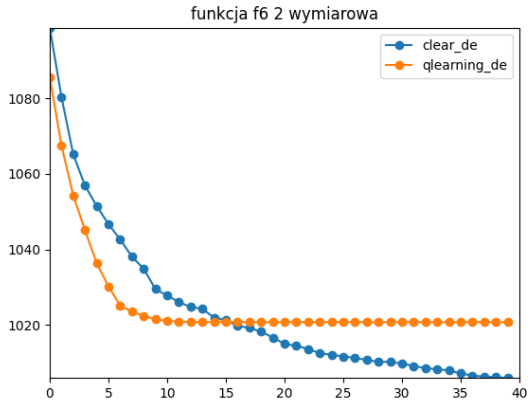
Rys. 6.35. Szukanie minimum funkcji f2 przez Q(10, 5000)



Rys. 6.36. Szukanie minimum funkcji f3 przez Q(10, 5000)

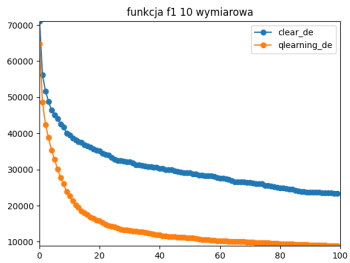


Rys. 6.37. Szukanie minimum funkcji f5 przez Q(10, 5000)

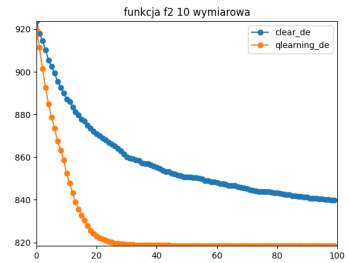


Rys. 6.38. Szukanie minimum funkcji f6 przez Q(10, 5000)

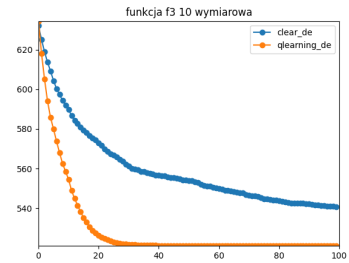
Testy na funkcjach 10-wymiarowych:



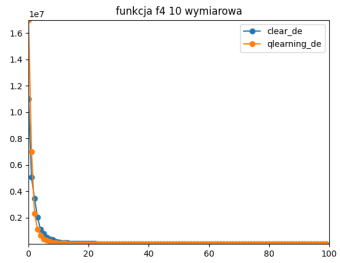
Rys. 6.39. Szukanie minimum funkcji f1 przez Q(10, 5000)



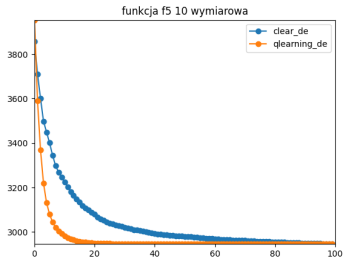
Rys. 6.40. Szukanie minimum funkcji f2 przez Q(10, 5000)



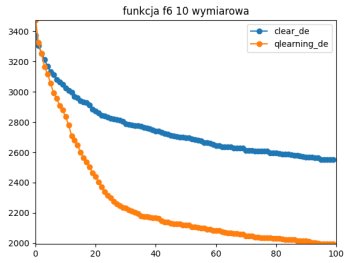
Rys. 6.41. Szukanie minimum funkcji f3 przez Q(10, 5000)



Rys. 6.42. Szukanie minimum funkcji f4 przez Q(10, 5000)



Rys. 6.43. Szukanie minimum funkcji f5 przez Q(10, 5000)



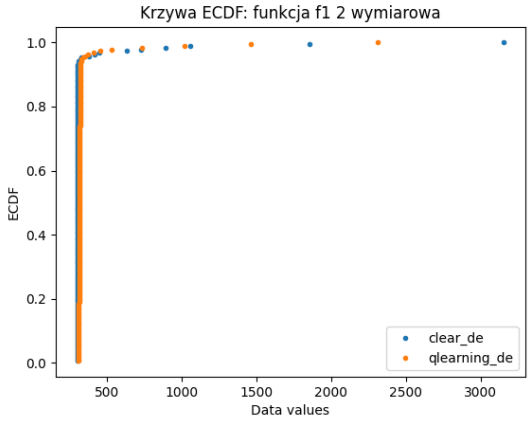
Rys. 6.44. Szukanie minimum funkcji f6 przez Q(10, 5000)

6.5. Krzywe ECDF

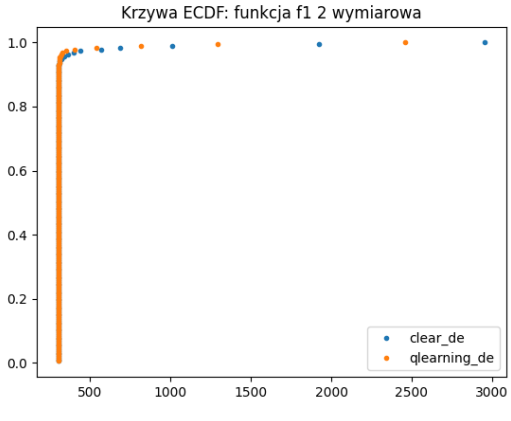
Po przetestowaniu algorytmu, postanowiliśmy zmienić podejście do nauczania algorytmu QLearning. Zmieniliśmy wielkość populacji na 25 - zgodnie z opisem z CEC2017. Zwiększyliśmy ilość epok przy uczeniu jak i przy testowaniu: Przy dwóch wymiarach do 200 epok, przy 10 wymiarach do 1000 epok. Większa ilość Epok powodowała bardzo długie wykonywanie testów i nie prowadziła do znacznych zmian wartości optymalizacji. Wprowadziliśmy również zmianę: teraz Q aktualizowane jest co 10 iteracje algorytmu DE.

Otrzymane wyniki postanowiliśmy reprezentować na Krzywych ECDF.

Testy na funkcjach 2-wymiarowych:

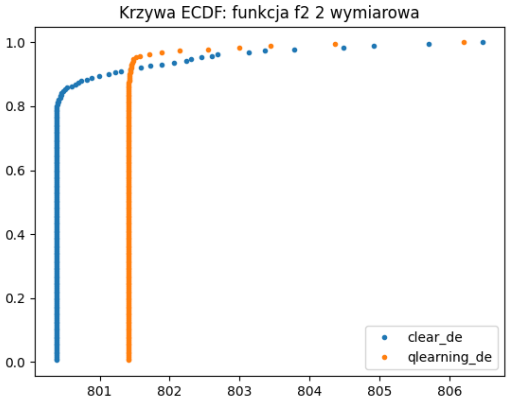


Rys. 6.45. Szukanie minimum funkcji f1 przez Q(2,1000)

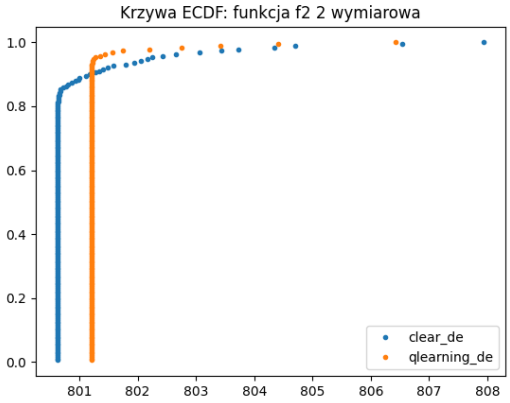


Rys. 6.46. Szukanie minimum funkcji f1 przez Q(10, 1000)

Funkcja f2

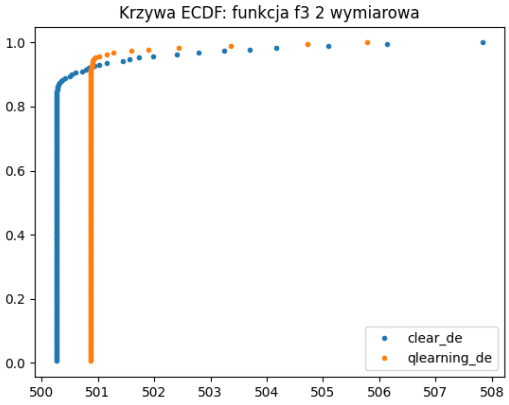


Rys. 6.47. Szukanie minimum funkcji f2 przez Q(2,1000)

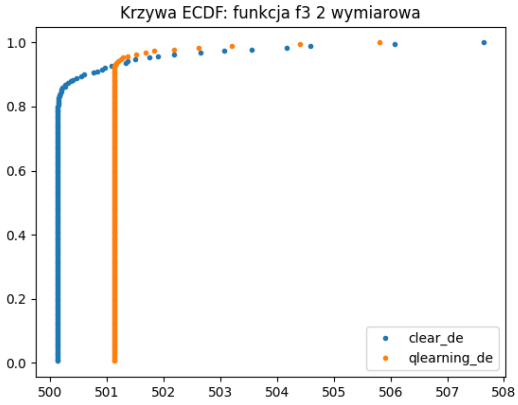


Rys. 6.48. Szukanie minimum funkcji f2 przez Q(10,1000)

Funkcja f3

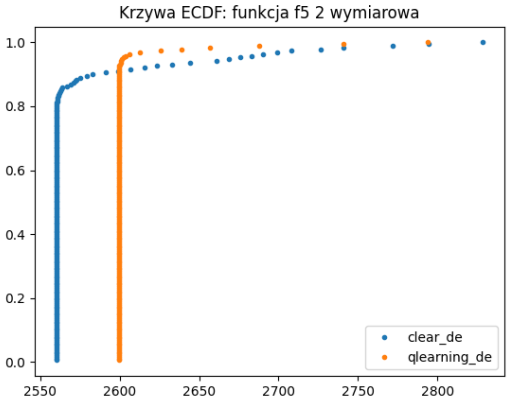


Rys. 6.49. Szukanie minimum funkcji f3 przez Q(2,1000)

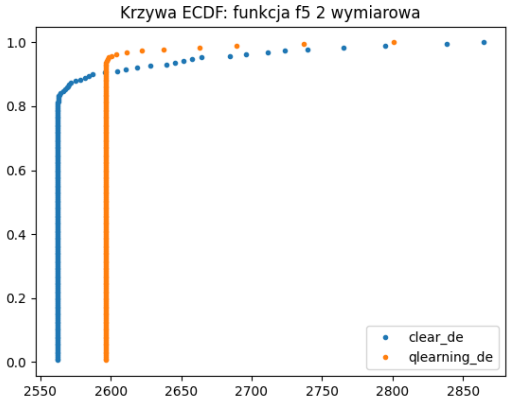


Rys. 6.50. Szukanie minimum funkcji f3 przez Q(10,1000)

Funkcja f5

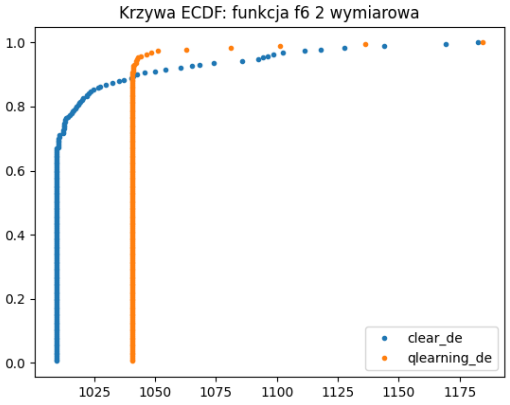


Rys. 6.51. Szukanie minimum funkcji f5 przez Q(2,

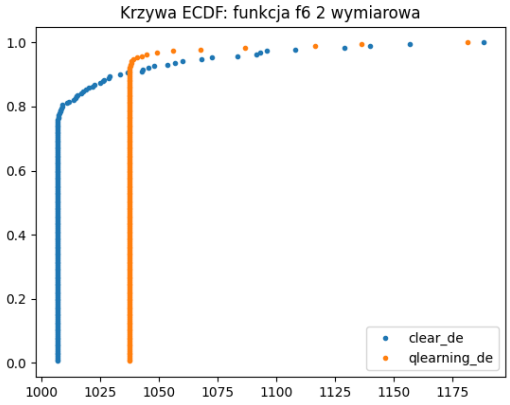


Rys. 6.52. Szukanie minimum funkcji f5 przez Q(10, 1000)

Funkcja f6

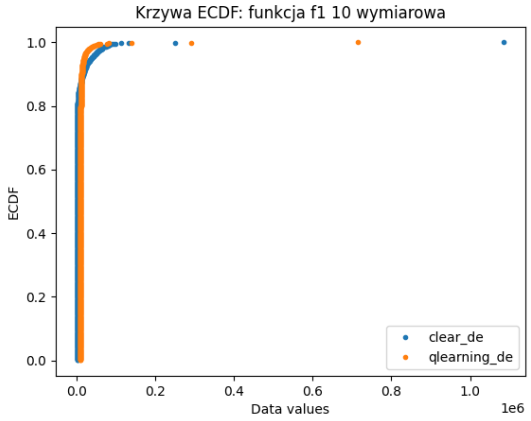


Rys. 6.53. Szukanie minimum funkcji f6 przez Q(2,

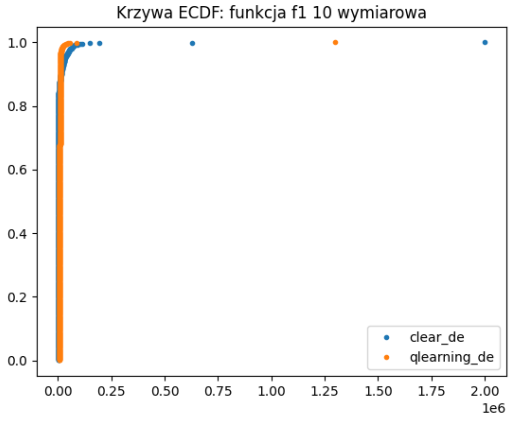


Rys. 6.54. Szukanie minimum funkcji f6 przez Q(10, 1000)

Testy na funkcjach 10-wymiarowych: Funkcja f1

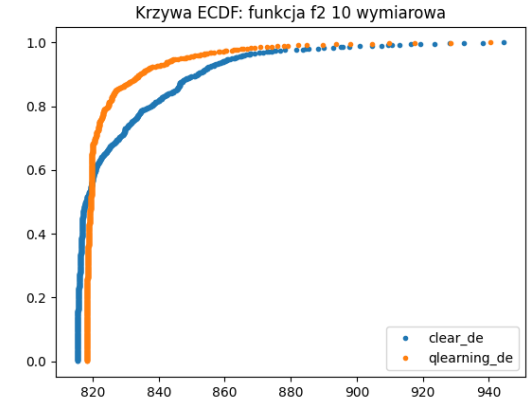


Rys. 6.55. Szukanie minimum funkcji f1 przez Q(2,

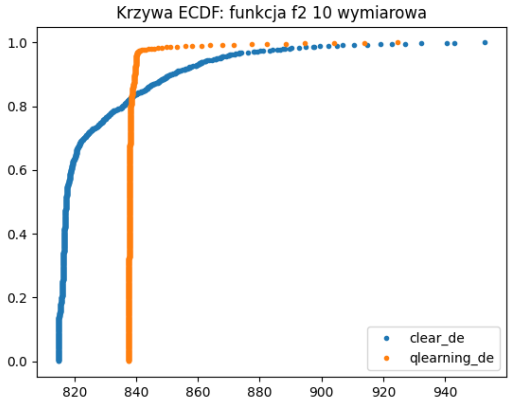


Rys. 6.56. Szukanie minimum funkcji f1 przez Q(10, 1000)

Funkcja f2

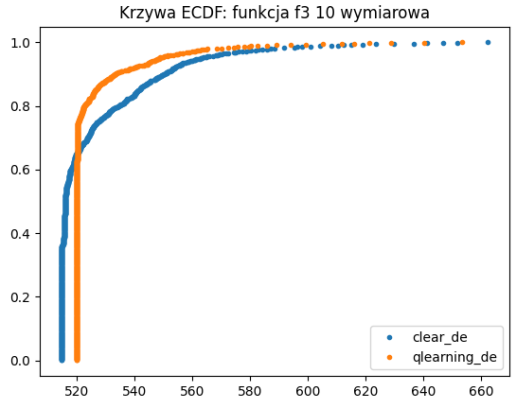


Rys. 6.57. Szukanie minimum funkcji f2 przez Q(2,

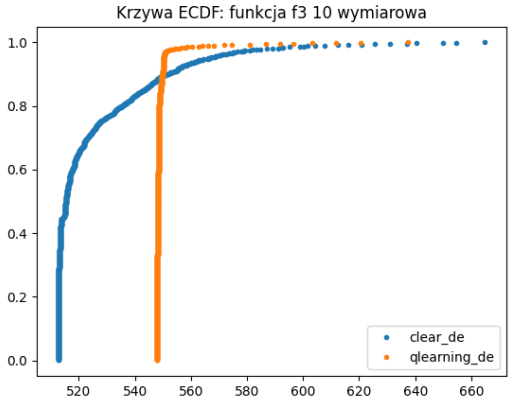


Rys. 6.58. Szukanie minimum funkcji f2 przez Q(10, 1000)

Funkcja f3

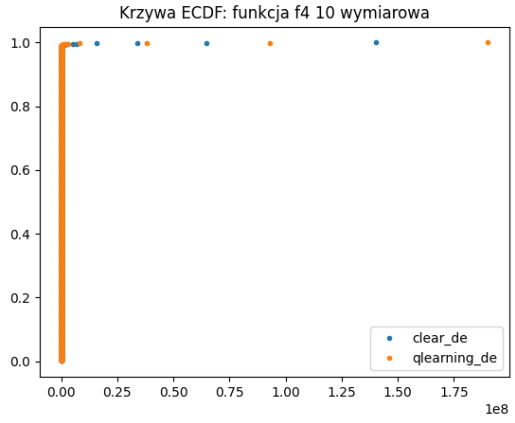


Rys. 6.59. Szukanie minimum funkcji f3 przez Q(2,

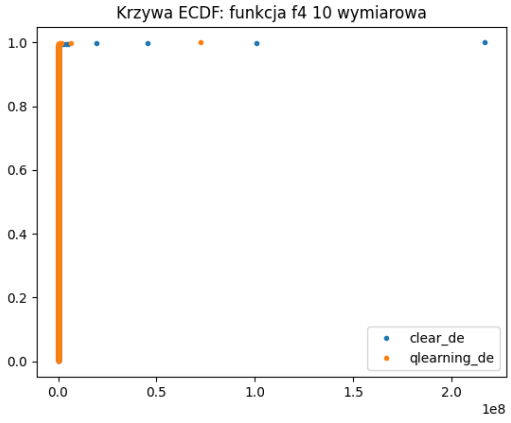


Rys. 6.60. Szukanie minimum funkcji f3 przez Q(10, 1000)

Funkcja f4

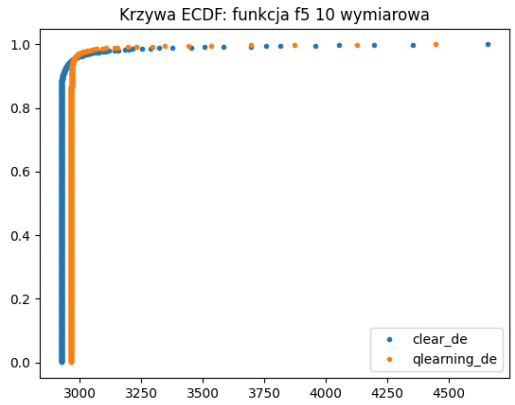


Rys. 6.61. Szukanie minimum funkcji f4 przez Q(2,

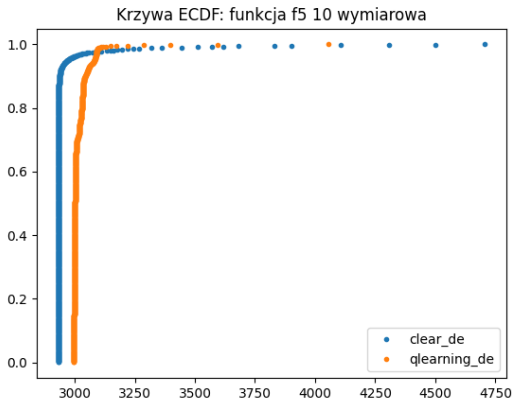


Rys. 6.62. Szukanie minimum funkcji f4 przez Q(10, 1000)

Funkcja f5

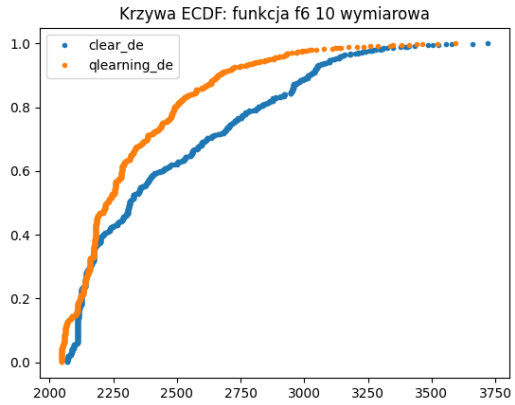


Rys. 6.63. Szukanie minimum funkcji f5 przez Q(2,1000)

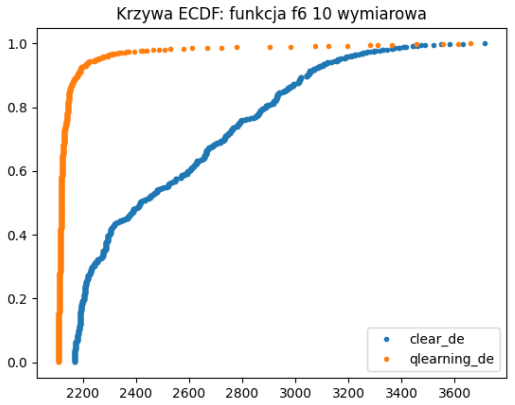


Rys. 6.64. Szukanie minimum funkcji f5 przez Q(10,1000)

Funkcja f6



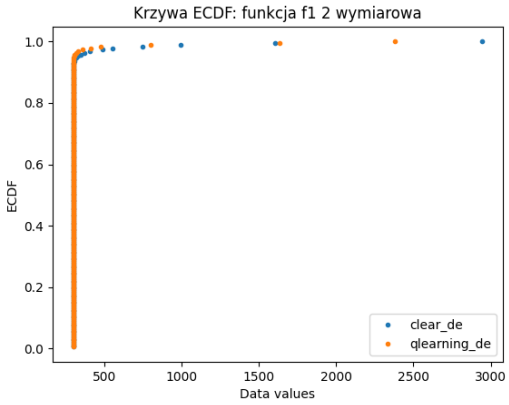
Rys. 6.65. Szukanie minimum funkcji f6 przez Q(2,1000)



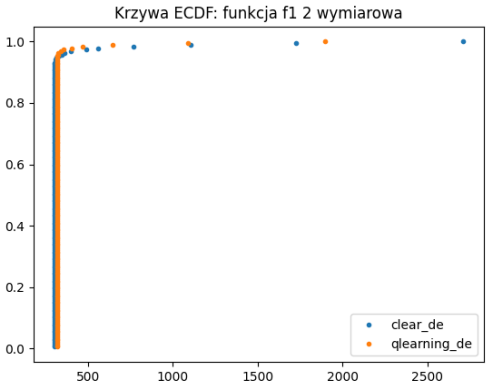
Rys. 6.66. Szukanie minimum funkcji f6 przez Q(10,1000)

Na powyższych wykresach możemy zauważyć, że nasz algorytm choć z początku jest szybszy niż zwykły algorytm DE często utyka w minimach lokalnych. Świadczy to o tym, że algorytm jest bardziej eksploatacyjny niż eksploracyjny. W związku z tym wprowadziliśmy zmianę w obliczaniu nagrody w algorytmie Qlearning. Teraz nagradzana jest również większa odległość między osobnikami w populacji. Przetestowaliśmy nowego agenta, a następnie porównaliśmy wyniki.

Testy na funkcjach 2-wymiarowych Funkcja f1

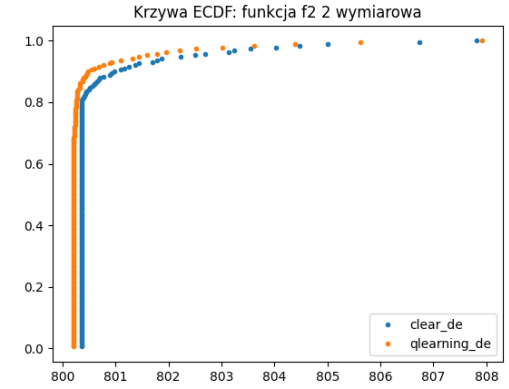


Rys. 6.67. Szukanie minimum funkcji f1 przez Q(2, 1000)

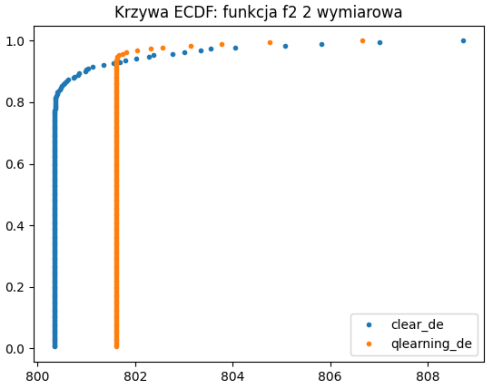


Rys. 6.68. Szukanie minimum funkcji f1 przez Q(10, 1000)

Funkcja f2

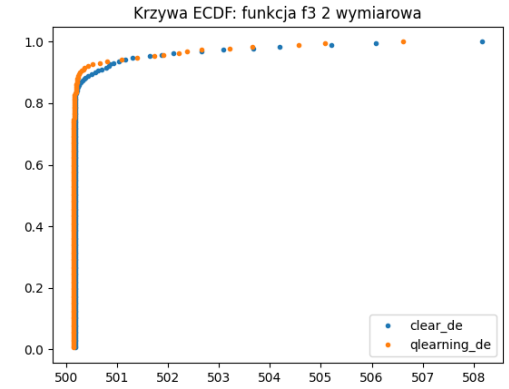


Rys. 6.69. Szukanie minimum funkcji f2 przez Q(2, 1000)

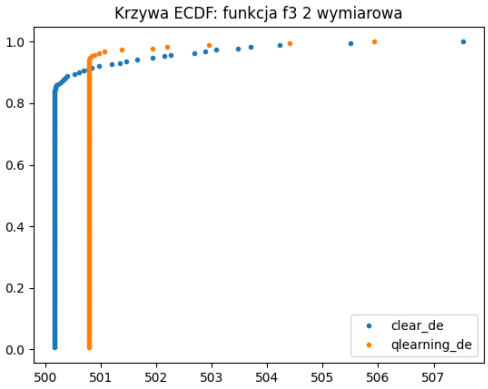


Rys. 6.70. Szukanie minimum funkcji f2 przez Q(10, 1000)

Funkcja f3

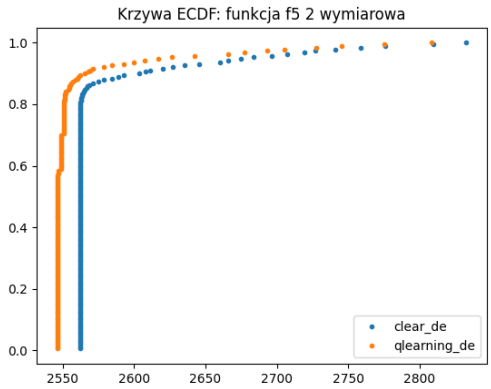


Rys. 6.71. Szukanie minimum funkcji f3 przez Q(2, 1000)

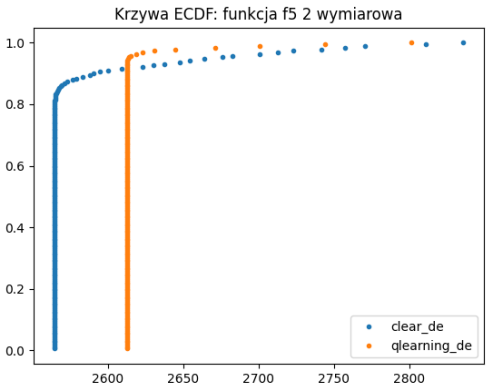


Rys. 6.72. Szukanie minimum funkcji f3 przez Q(10, 1000)

Funkcja f5

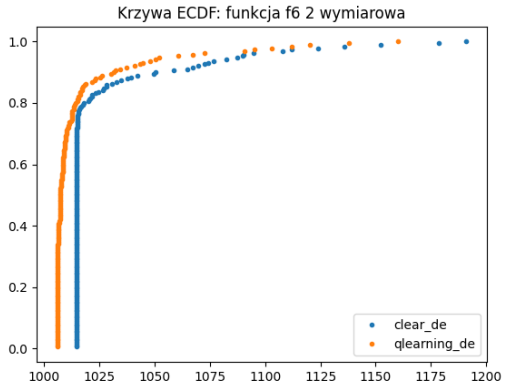


Rys. 6.73. Szukanie minimum funkcji f5 przez Q(2, 1000)

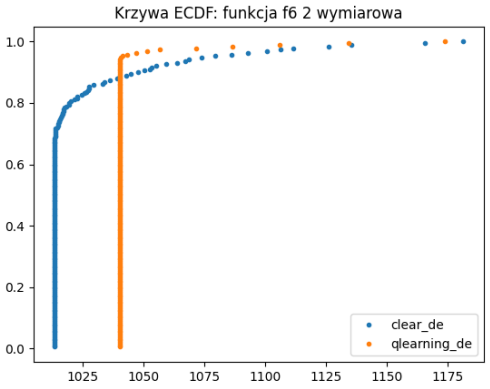


Rys. 6.74. Szukanie minimum funkcji f5 przez Q(10, 1000)

Funkcja f6

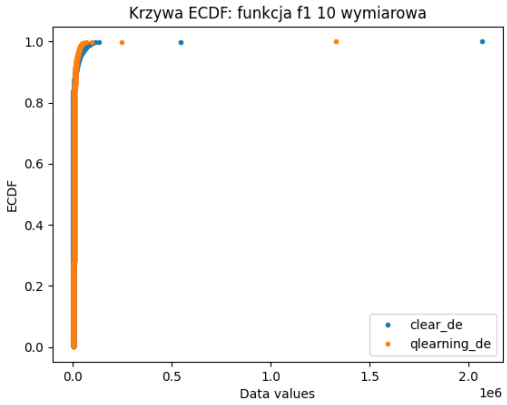


Rys. 6.75. Szukanie minimum funkcji f6 przez Q(2, 1000)

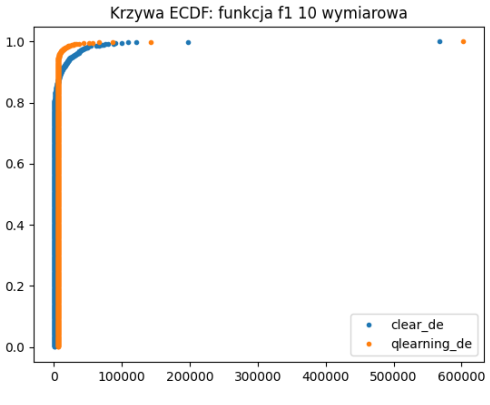


Rys. 6.76. Szukanie minimum funkcji f6 przez Q(10, 1000)

Testy na funkcjach 10-wymiarowych Funkcja f1

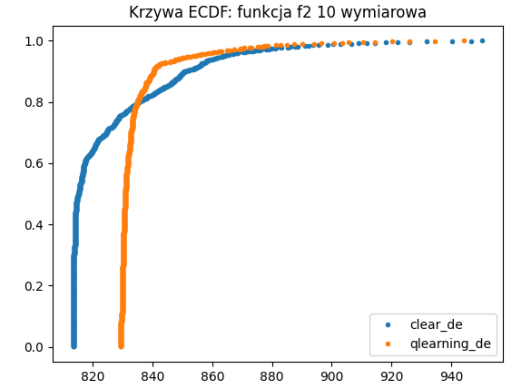


Rys. 6.77. Szukanie minimum funkcji f1 przez Q(2, 1000)

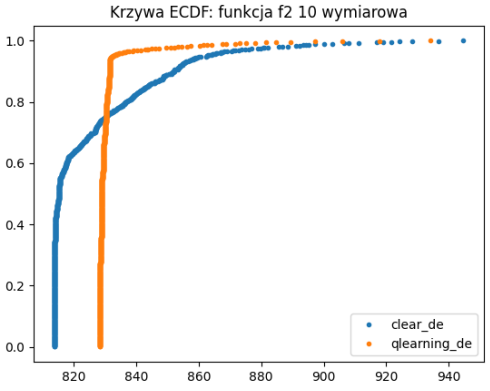


Rys. 6.78. Szukanie minimum funkcji f1 przez Q(10, 1000)

Funkcja f2

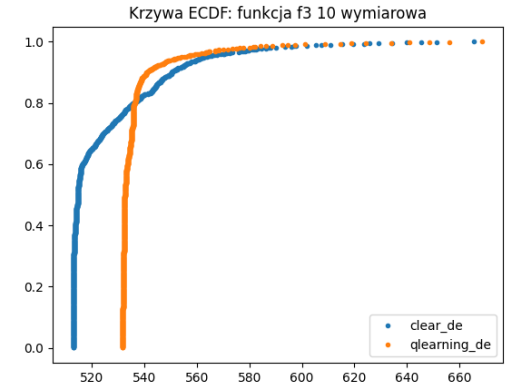


Rys. 6.79. Szukanie minimum funkcji f2 przez Q(2, 1000)

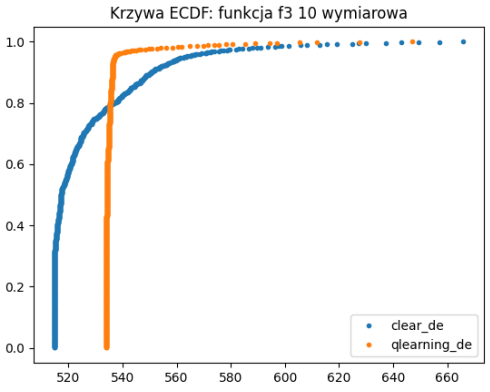


Rys. 6.80. Szukanie minimum funkcji f2 przez Q(10, 1000)

Funkcja f3

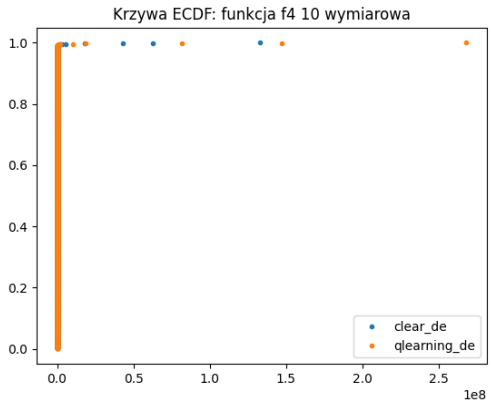


Rys. 6.81. Szukanie minimum funkcji f3 przez Q(2, 1000)

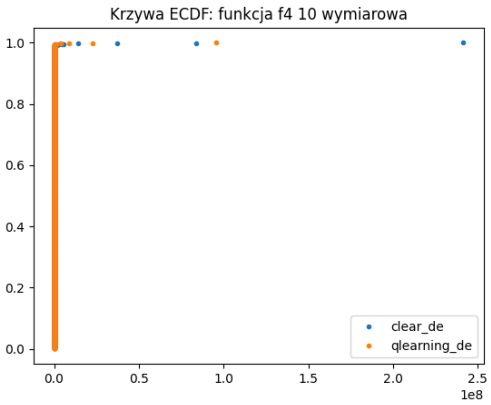


Rys. 6.82. Szukanie minimum funkcji f3 przez Q(10, 1000)

Funkcja f4

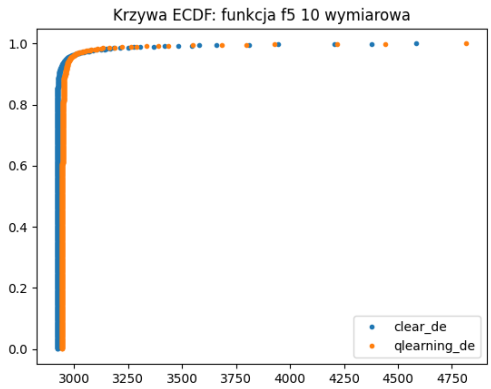


Rys. 6.83. Szukanie minimum funkcji f4 przez Q(2, 1000)

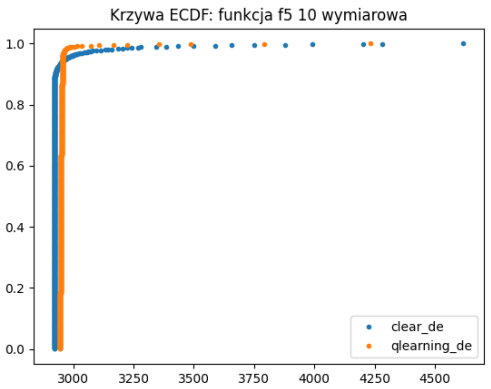


Rys. 6.84. Szukanie minimum funkcji f4 przez Q(10, 1000)

Funkcja f5

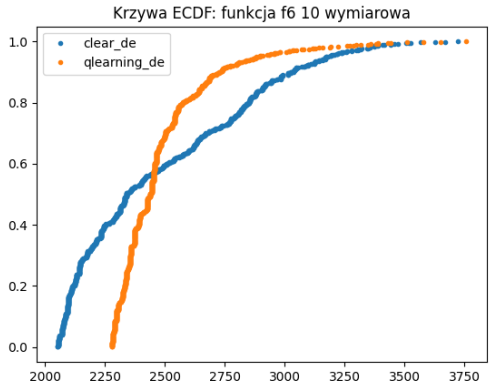


Rys. 6.85. Szukanie minimum funkcji f5 przez Q(2, 1000)

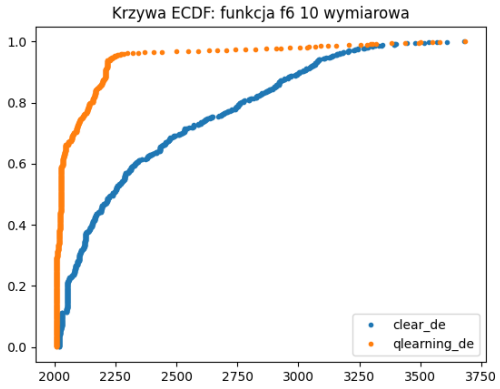


Rys. 6.86. Szukanie minimum funkcji f5 przez Q(10, 1000)

Funkcja f6



Rys. 6.87. Szukanie minimum funkcji f6 przez Q(2, 1000)



Rys. 6.88. Szukanie minimum funkcji f6 przez Q(10, 1000)

6.6. Porównanie wyników

Przedstawiliśmy wyniki w tabeli aby lepiej zobrazować różnice między zwykłym DE, DE z QLearninigiem nagradzającym tylko przeżywalność mutantów (stary) oraz DE z QLearninigiem nagradzającym przeżywalność mutantów oraz większą odległość między osobnikami (nowy).

6.6.1. Funkcje dwuwymiarowe

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	300,00	0,00	300,00	300,00
qlearning DE (stary)	310,98	73,23	300,00	822,97
qlearning DE (nowy)	315,96	111,73	300,00	1098,08

Tab. 6.1. Porównanie wyników dla funkcji f1(2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	300,00	0,00	300,00	300,00
qlearning DE (stary)	304,23	22,59	300,00	450,83
qlearning DE (nowy)	315,96	111,73	300,00	1098,08

Tab. 6.2. Porównanie wyników dla funkcji f1(10 wymiarów)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	800,36	0,52	800,00	801,99
qlearning DE (stary)	801,41	1,44	800,00	804,97
qlearning DE (nowy)	801,61	1,58	800,00	804,97

Tab. 6.3. Porównanie wyników dla funkcji f2(2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	800,34	0,47	800,00	800,99
qlearning DE (stary)	801,22	1,57	800,00	808,95
qlearning DE (nowy)	801,61	1,58	800,00	804,97

Tab. 6.4. Porównanie wyników dla funkcji f2(10 wymiarów)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	500,18	0,43	500,00	501,99
qlearning DE (stary)	500,88	1,10	500,00	504,97
qlearning DE (nowy)	500,78	0,87	500,00	504,97

Tab. 6.5. Porównanie wyników dla funkcji f3(2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	500,16	0,36	500,00	500,99
qlearning DE (stary)	501,12	1,29	500,00	504,97
qlearning DE (nowy)	500,78	0,87	500,00	504,97

Tab. 6.6. Porównanie wyników dla funkcji f3(10 wymiarów)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	2562,00	68,96	2500,00	2800,00
qlearning DE (stary)	2599,54	93,73	2500,00	2800,00
qlearning DE (nowy)	2612,62	81,75	2500,00	2800,00

Tab. 6.7. Porównanie wyników dla funkcji f5(2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	2562,00	68,96	2500,00	2800,00
qlearning DE (stary)	2596,37	89,50	2500,00	2909,63
qlearning DE (nowy)	2612,62	81,75	2500,00	2800,00

Tab. 6.8. Porównanie wyników dla funkcji f5(10 wymiarów)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	1014,80	24,81	1000,00	1143,06
qlearning DE (stary)	1040,52	48,23	1000,00	1178,58
qlearning DE (nowy)	1040,38	47,12	1000,00	1141,37

Tab. 6.9. Porównanie wyników dla funkcji f6(2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	1013,44	25,16	1000,00	1124,93
qlearning DE (stary)	1037,70	46,55	1000,00	1178,58
qlearning DE (nowy)	1040,38	47,12	1000,00	1141,37

Tab. 6.10. Porównanie wyników dla funkcji f6(10 wymiarów)

6.6.2. Funkcje dziesięciowymiarowe

Tab. 6.11. Porównanie wyników dla funkcji f1 (2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	300,00	0,00	300,00	300,00
qlearning DE (stary)	8561,41	8450,74	300,29	33348,81
qlearning DE (nowy)	7761,31	12579,09	300,00	55332,44

Tab. 6.12. Porównanie wyników dla funkcji f1 (10 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	300,00	0,00	300,00	300,00
qlearning DE (stary)	10561,65	14784,87	1505,66	77280,75
qlearning DE (nowy)	5684,75	5923,03	433,51	36569,22

Tab. 6.13. Porównanie wyników dla funkcji f2 (2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	813,45	6,72	803,98	833,59
qlearning DE (stary)	818,36	14,60	804,28	867,91
qlearning DE (nowy)	829,41	21,53	803,27	870,37

Tab. 6.14. Porównanie wyników dla funkcji f2 (10 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	813,71	7,43	801,39	838,32
qlearning DE (stary)	837,62	14,35	809,95	877,65
qlearning DE (nowy)	828,44	12,32	808,95	866,30

Tab. 6.15. Porównanie wyników dla funkcji f3 (2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	512,83	6,83	501,99	533,80
qlearning DE (stary)	519,72	15,54	504,74	584,79
qlearning DE (nowy)	531,82	20,89	504,00	563,32

Tab. 6.16. Porównanie wyników dla funkcji f3 (10 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	514,72	7,08	503,98	532,34
qlearning DE (stary)	547,87	17,91	515,92	586,56
qlearning DE (nowy)	534,10	15,66	507,96	586,70

Tab. 6.17. Porównanie wyników dla funkcji f4 (2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	1502,26	1,75	1500,00	1507,96
qlearning DE (stary)	2456,39	3336,95	1501,54	24497,08
qlearning DE (nowy)	2755,81	5364,14	1501,26	37056,50

Tab. 6.18. Porównanie wyników dla funkcji f4 (10 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	1502,45	1,86	1500,00	1509,03
qlearning DE (stary)	31653,18	56872,93	1583,38	329546,68
qlearning DE (nowy)	10561,65	14784,87	1505,66	77280,75

Tab. 6.19. Porównanie wyników dla funkcji f5 (2 wymiary)

Metoda	Średni wynik	Odchylenie standardowe	Minimum	Maksimum
clear DE	2921,98	25,72	2897,74	2981,14
qlearning DE (stary)	2963,75	113,93	2897,74	3667,55
qlearning DE (nowy)	2945,15	30,15	2897,76	3042,91

Tab. 6.20. Porównanie wyników dla funkcji f5 (10 wymiary)

Metoda	**Średni wynik**	**Odchylenie standardowe**	**Minimum**	**Maksimum**
clear DE	2920,97	23,83	2897,74	2967,00
qlearning DE (stary)	2996,72	108,05	2897,76	3461,53
qlearning DE (nowy)	2945,29	40,56	2897,74	3082,63

Tab. 6.21. Porównanie wyników dla funkcji f6 (2 wymiary)

Metoda	**Średni wynik**	**Odchylenie standardowe**	**Minimum**	**Maksimum**
clear DE	2049,53	465,72	1185,18	2961,52
qlearning DE (stary)	2045,11	500,35	1329,00	3113,73
qlearning DE (nowy)	2269,34	526,76	1185,67	2971,68

Tab. 6.22. Porównanie wyników dla funkcji f6 (10 wymiarów)

Metoda	**Średni wynik**	**Odchylenie standardowe**	**Minimum**	**Maksimum**
clear DE	2020,77	385,22	1309,16	2845,11
qlearning DE (stary)	2106,94	342,67	1382,92	3175,56
qlearning DE (nowy)	2006,48	387,03	1150,73	2753,83

7. Wnioski

Z przeprowadzonych eksperymentów wyciągnęliśmy wnioski dotyczące różnych aspektów.

7.1. Czas znajdowania minimum i jego dokładność

Zastosowanie uczenia ze wzmocnieniem do ustawienia sposobu mutacji i współczynnika skalowania (F) w algorytmie ewolucji różnicowej w większości przypadków przyspiesza znajdowanie minimum jednak tylko lokalnego. Algorytm nie posiada takiej samej zdolności globalnej minimalizacji jak oryginalna implementacja DE. Jest to wynik, którego się spodziewaliśmy. W naszych starych agentach nagroda była dawana tylko za procent sukcesów co bardzo zmniejszało zdolności eksploracyjne algorytmu i bardzo szybko dochodził do minimum lokalnego. Po wprowadzeniu poprawek i dodaniu do nagrody średniej odległości osobników w populacji, udało się osiągnąć większe rozproszenie populacji a co za tym idzie większą eksplorację. Poprawiło to wyniki względem starych agentów, jednak nie na tyle żeby uzyskiwać wyniki mniejsze od tradycyjnej ewolucji różnicowej. Podsumowując algorytm nadawałby się do użycia w przypadku gdzie naszym celem byłoby jak najszybsze uzyskanie minimum, niekoniecznie globalnego.

7.2. Działanie w zależności od kształtu i wymiarowości funkcji

W większości przypadków algorytm lepiej radzi sobie z minimalizacją funkcji 2-wymiarowych. Jest to oczywiste, ponieważ zadanie minimalizacji w przestrzeni 2-wymiarowej jest znacznie prostsze niż 10-wymiarowej. Wynika to z większej: złożoności przestrzeni poszukiwań, liczby zmiennych oraz ilości minimów lokalnych w przestrzeni 10 wymiarowej. Ciekawszym spostrzeżeniem jest fakt, że algorytmy uczone na funkcjach 10-wymiarowych radzą sobie lepiej i z minimalizacją funkcji 2-wymiarowych jak i 10-wymiarowych. Może to wynikać z tego, że dzięki wprowadzeniu większej ilości zmiennych nasz algorytm zachowuje większą zdolność eksploracji w przeciwieństwie do funkcji 2-wymiarowych

7.3. Tabela Q

Tabelę Q ograniczyliśmy do 2500 stanów $Q(v_1, v_2, v_3)$, gdzie:
v_1 -stosunek liczby sukcesów do liczby prób - od 0 do 1 z rozdzielczością `0,05`
v_2 - średnią odległość między osobnikami od 0 do $20\sqrt{dim}$ (to jest najdłuższa możliwa odległość między osobnikami w przestrzeni o wymiarowości dim) podzieliliśmy na 20 równych
v_3 - od 0 do 5 gdzie każda liczba całkowita odpowiada innej akcji.

Przy takiej tabeli Q powstaje wiele stanów, które nigdy nie zostaną odwiedzone przez algorytm uczenia. W naszym przypadku ilość stanów gdzie wartość Q została zmieniona była równa około 500 . Pierwszym powodem tego jest niemożliwość odwiedzenia stanów z wysokim procentem sukcesów, pomimo że w funkcji nagrody algorytm był zachęcany do maksymalizowania sukcesu nie chcieliśmy całkowicie rezygnować z eksploracji. Drugim powodem jest fakt że większość osobników w populacji jest stosunkowo blisko siebie co powoduje, że średnia odległość między osobnikami przez większość czasu działania algorytmu będzie podobna.

7.4. Podsumowanie

Z przeprowadzonych eksperymentów wynika, że zastosowanie uczenia ze wzmocnieniem do algorytmu ewolucji różnicowej przyspiesza znajdowanie minimum, choć nie zawsze jest to minimum globalne. Wprowadzenie nagrody za średnią odległość osobników w populacji poprawiło zdolności eksploracyjne algorytmu, jednak wciąż nie dorównuje to tradycyjnej ewolucji różnicowej pod względem zdolności do globalnej minimalizacji.

Analiza działania algorytmu w zależności od kształtu i wymiarowości funkcji pokazała, że algorytm lepiej radzi sobie z funkcjami 2-wymiarowymi. Algorytmy uczone na funkcjach 10-wymiarowych wykazują lepsze zdolności eksploracyjne.

Tabela Q, ograniczona do 2500 stanów, pokazała, że nie wszystkie stany są odwiedzane przez algorytm, a zmiany wartości Q występują w około 500 stanach. Powodem tego jest zarówno brak możliwości odwiedzenia stanów z wysokim procentem sukcesów, jak i fakt, że większość osobników w populacji znajduje się blisko siebie, co powoduje, że średnia odległość między osobnikami jest stosunkowo stała przez większość czasu działania algorytmu.