WSI: Laboratorium

Ćwiczenie 2: Algorytmy ewolucyjne i genetyczne

Paweł Skierś, 310895

Semestr zimowy 2021/22

ZADANIE

Celem ćwiczenia jest implementacja algorytmu genetycznego Hollanda oraz zastosowanie go do znalezienia minimum funkcji f dla wektora czterech liczb całkowitych z zakresu $[-16, 16)$. Implementacja powinna działać dla dowolnego problemu minimalizacji o reprezentacji odpowiedniej dla algorytmu genetycznego Hollanda.

Funkcja:

$$f(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2 + \sin^3(1.5x_3) + (x_3 - 1)^2 [1 + \sin^2(1.5x_4)] + (x_4 - 1)^2 [1 + \sin^2(x_4)]$$

DOKUMENTACJA ROZWIĄZANIA

Program będący napisany na potrzebę tego ćwiczenia podzielony został na trzy pliki: hollands_algorithm.py, target_function.py oraz main.py. W dalszej części zostanie omówione za co odpowiadają poszczególne pliki oraz zastosowane w nich rozwiązania.

hollands_algorithm.py

Plik ten odpowiedzialny jest za część algorytmiczną programu. Zawiera on implementację funkcję hollands_algorithm() będącą funkcją wykonującą algorytm Hollanda, a także funkcje cross_and_mutate(), reproduction(), find_best() i evaluate(), które są funkcjami pomocniczymi dla hollands_algorithm(). Poniżej zostaną omówione poszczególne funkcje.

hollands_algorithm()

Funkcja ta wykonuje algorytm genetyczny Hollanda (minimalizacja) dla zadanych argumentów. Funkcja przyjmuje następujące argumenty

- target_function – jest to funkcja, która wykorzystywana będzie jako funkcja celu. Może być to na przykład funkcja, którą chcemy minimalizować. Argument obowiązkowy, musi być wywoływalny.
- population – początkowa populacja, od której zacznie się ewolucja. Powinna być iterowalna i zawierać osobniki dla funkcji celu. Argument obowiązkowy.
- population_size – liczebność początkowej populacji. Argument obowiązkowy.
- mutation_probability – prawdopodobieństwo, z którym następuje mutacja każdego z genów każdego z osobników populacji. Powinna być to liczba z przedziału [0,1]. Argument obowiązkowy.
- crossing_probability – prawdopodobieństwo, z którym następuje krzyżowanie osobników. Powinna być to liczba z przedziału [0,1]. Argument obowiązkowy.
- max_iteration – liczba generacji, którą wytworzy algorytm. Po wytworzeniu podanej w tym argumencie liczby generacji algorytm kończy swoją pracę. Powinna to być liczba naturalna. Argument obowiązkowy.

Funkcja zwraca (best_x, best_ev, history, history_ev) gdzie:

- best_x – najlepszy znaleziony osobnik
- best_ev – najlepsza znaleziona wartość funkcji celu

- history – lista wszystkich stworzonych osobników, ustawionych w kolejności wytworzenia
- history_ev – lista wszystkich ocen osobników, ustawionych w kolejności wytworzenia

cross_and_mutate()

Funkcja ta odpowiada za krzyżowanie i mutowanie populacji algorytmu po reprodukcji. Zaimplementowane krzyżowanie jest krzyżowaniem jednopunktowym, natomiast mutowanie następuje niezależnie dla każdego genu każdego osobnika. Funkcja przyjmuje następujące argumenty:

- reproduced – populacja po reprodukcji, na której wykonane zostanie krzyżowanie i mutacja. Powinna być iterowalna i zawierać osobniki dla funkcji celu. Argument obowiązkowy.
- mutation_probability – prawdopodobieństwo, z którym następuje mutacja każdego z genów każdego z osobników populacji. Powinna być to liczba z przedziału [0,1]. Argument obowiązkowy.
- crossing_probability – prawdopodobieństwo, z którym następuje krzyżowanie osobników. Powinna być to liczba z przedziału [0,1]. Argument obowiązkowy.

Funkcja zwraca:

- crossed – populacja skrzyżowana i zmutowana

reproduction()

Funkcja ta odpowiada za reprodukcję populacji. Zaimplementowana reprodukcja jest reprodukcją ruletkową dla minimalizacji. W przypadku gdy jedna z podanych funkcji ocen jest ujemna następuje zwiększenie wszystkich ocen tak, by wszystkie oceny były dodatnie. Funkcja przyjmuje następujące argumenty:

- population – populacja, na której wykonana ma być reprodukcja. Powinna być iterowalna i zawierać osobniki dla funkcji celu. Argument obowiązkowy
- evaluations – oceny poszczególnych osobników uzyskane za pomocą funkcji celu, ustawione w takiej samej kolejności co osobniki w populacji. Argument powinien być typu float. Argument obowiązkowy
- population_size – rozmiar reprodukowanej populacji. Argument typu int, obowiązkowy.

Funkcja zwraca:

- new_population – populacja uzyskana w wyniku reprodukcji

find_best()

Funkcja zwracająca najlepszego osobnika i jego ocenę z podanej populacji. Funkcja przyjmuje następujące argumenty:

- population - populacja, którą przeszukujemy. Powinna być iterowalna i zawierać osobniki dla funkcji celu. Argument obowiązkowy
- evaluations – oceny poszczególnych osobników uzyskane za pomocą funkcji celu, ustawione w takiej samej kolejności co osobniki w populacji. Argument powinien być typu float. Argument obowiązkowy

Funkcja zwraca (population[best], evaluations[best]) gdzie:

- population[best] – najlepszy osobnik
- evaluations[best] – najlepsza ocean

evaluate()

Funkcja zwraca listę ocen osobników z populacji. Funkcja przyjmuje następujące argumenty:

- target_function – jest to funkcja, która wykorzystywana będzie jako funkcja celu. Może być to na przykład funkcja, którą chcemy minimalizować. Argument obowiązkowy, musi być wywoływalny.
- population – populacja, której ocen szukamy. Powinna być iterowalna i zawierać osobniki dla funkcji celu. Argument obowiązkowy

Funkcja zwraca listę ocen osobników.

target_funcion.py

Plik ten zawiera implementacje funkcji celu dla zadanego problemu. Zawiera funkcje target_function() i deode(). Poniżej omówione zostaną zasady działania tych funkcji.

target_function()
Funkcja odpowiedzialna za implementację funkcji celu dla zadanego problemu. Funkcja dekoduje otrzymanego osobnika i oblicza dla niego wartość zadanej funkcji. Funkcja przyjmuje argument:

- individual – osobnik, którego ocenę chcemy otrzymać. Reprezentowany powinien być przez listę 0 i 1 długości 20.

Funkcja zwraca ocenę osobnika.

decode()
Funkcja odpowiedzialna za dekodowanie postaci genowej do postaci wektora liczb całkowitych. Kodowanie użyte to kodowanie binarne z biasem = -16. Funkcja przyjmuje argument:

- individual – osobnik, którego ocenę chcemy otrzymać. Reprezentowany powinien być przez listę 0 i 1 długości 20.

Funkcja zwraca zdekodowanego osobnika do postaci wektora liczb 4 całkowitych (lista 4 liczb całkowitych z zakresu [-16, 15].

main.py

Jest to plik główny programu. Znajduje się w nim funkcja main() pozwalająca na wczytanie liczebności populacji startowej, prawdopodobieństwa mutacji i krzyżowania oraz liczby generacji które ma wygenerować algorytm. Następnie algorytm losuje początkową populację, wykonuje algorytm ewolucyjny Hollanda, wpisuje znalezione minimum funkcji oraz wizualizuje kilka z wytworzonych populacji i zmianę otrzymywanych ocen osobników.

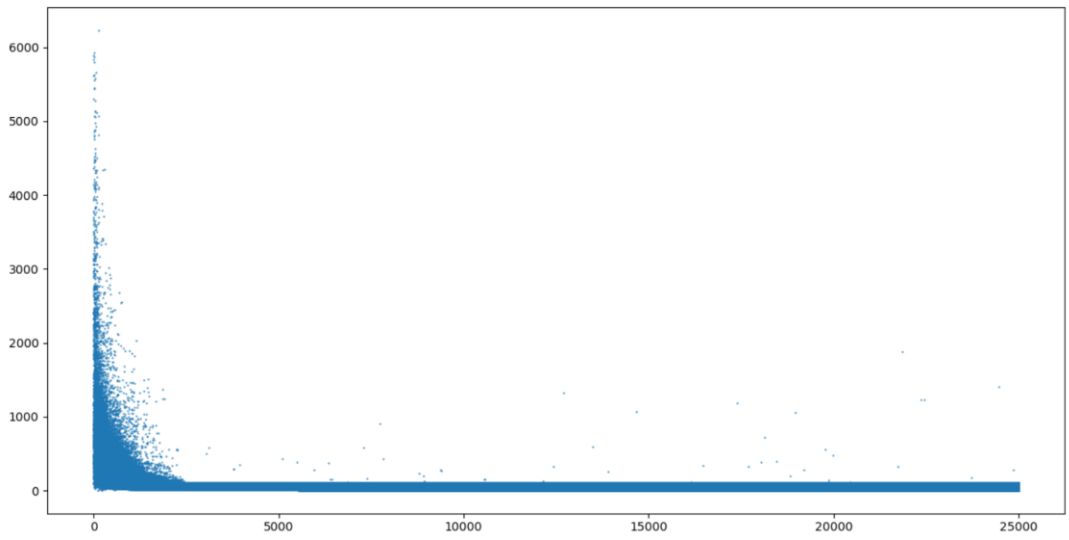
BADANIE WPŁYWU DOBORU PARAMETRÓW POCZĄTKOWYCH NA ALGORYTM GENETYCZNY HOLLANDA

Wpływ prawdopodobieństwa mutacji

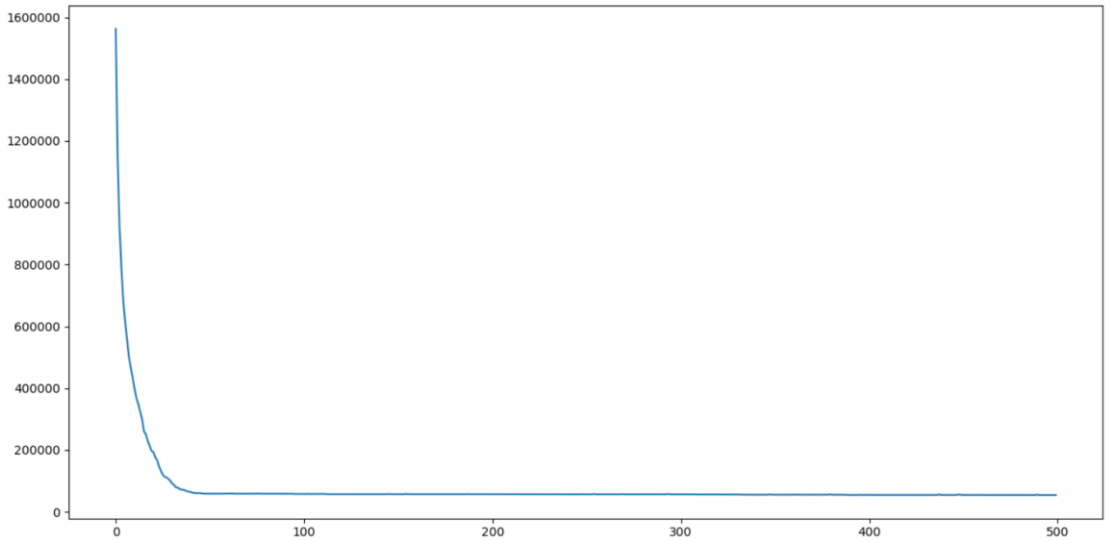
1. **Bardzo małe prawdopodobieństwo mutacji**

```
Enter population size: 50
Enter mutation probability: 0.00001
Enter crossing probability: 0.7
Enter number of iterations: 500
Best found x =  [0, 4, 1, 1] Best evaluation =  2.992503769369315
```

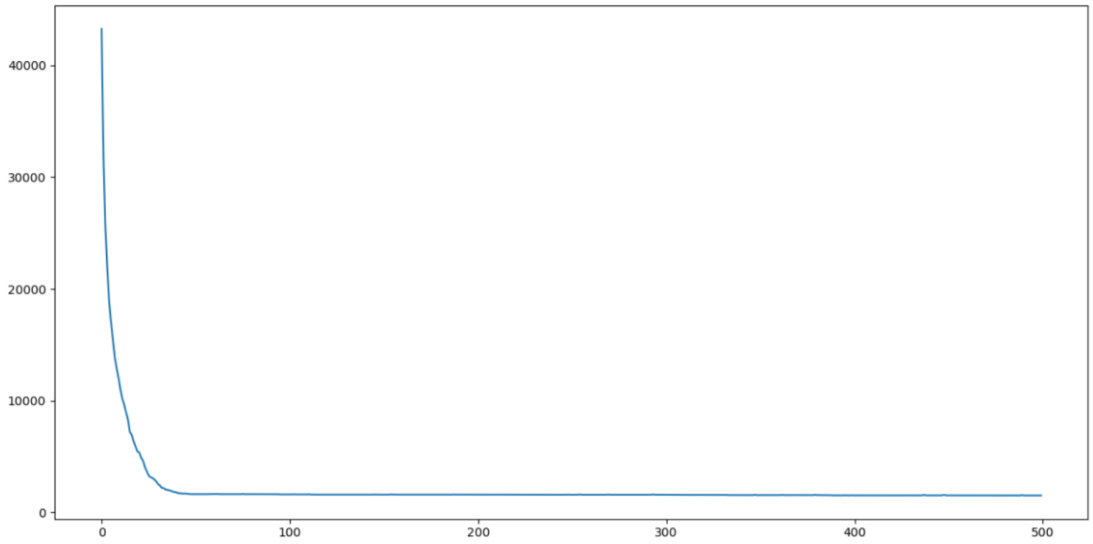
Dane startowe i znalezione minimum



Wytworzone osobniki i ich oceny na wykresie



Wykres średniej oceny populacji od numeru generacji

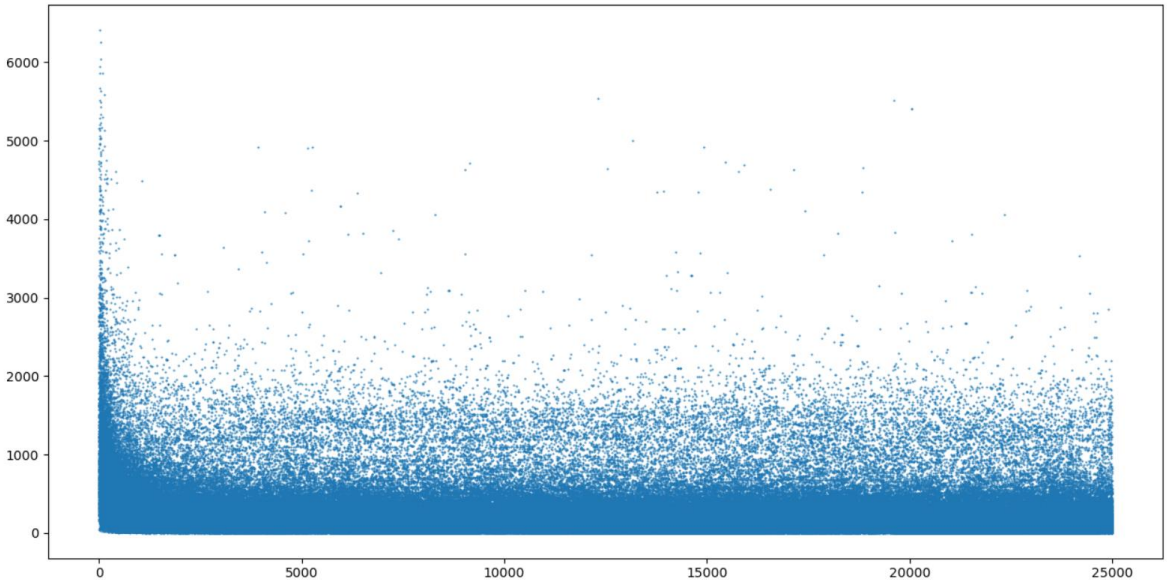


Wykres odchylenia standardowego oceny on numeru generacji

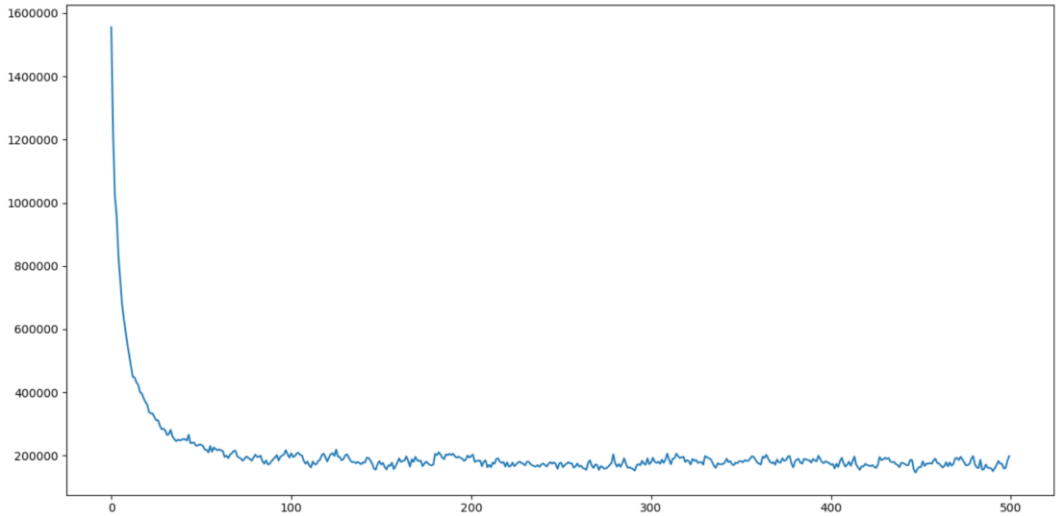
2. Małe prawdopodobieństwo mutacji

```
Enter population size: 50
Enter mutation probability: 0.01
Enter crossing probability: 0.7
Enter number of iterations: 500
Best found x =  [1, 3, 1, 1] Best evaluation =  0.9925037693693152
```

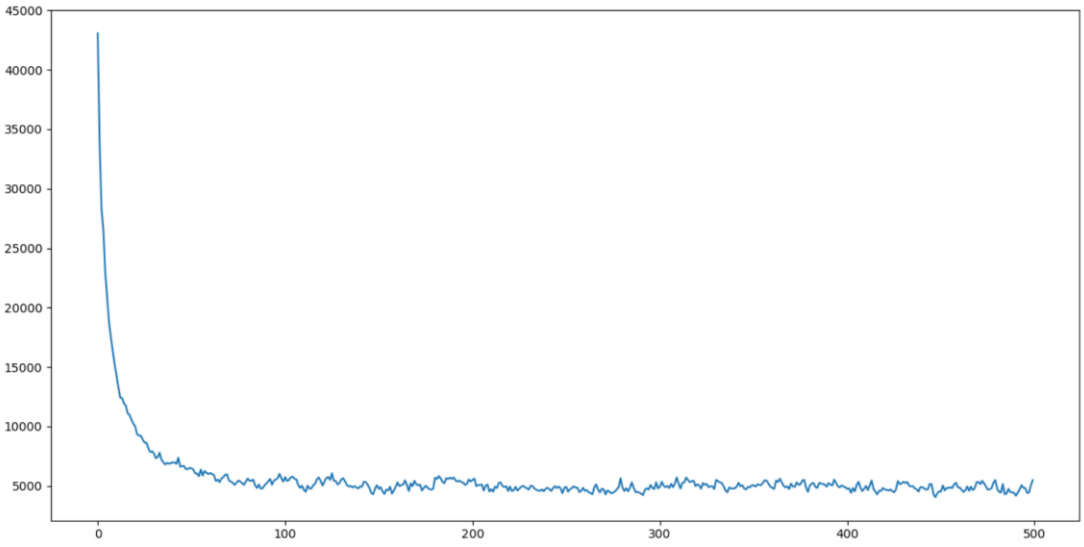
Dane startowe i znalezione minimum



Wytworzone osobniki i ich oceny na wykresie



Wykres średniej oceny populacji od numeru generacji

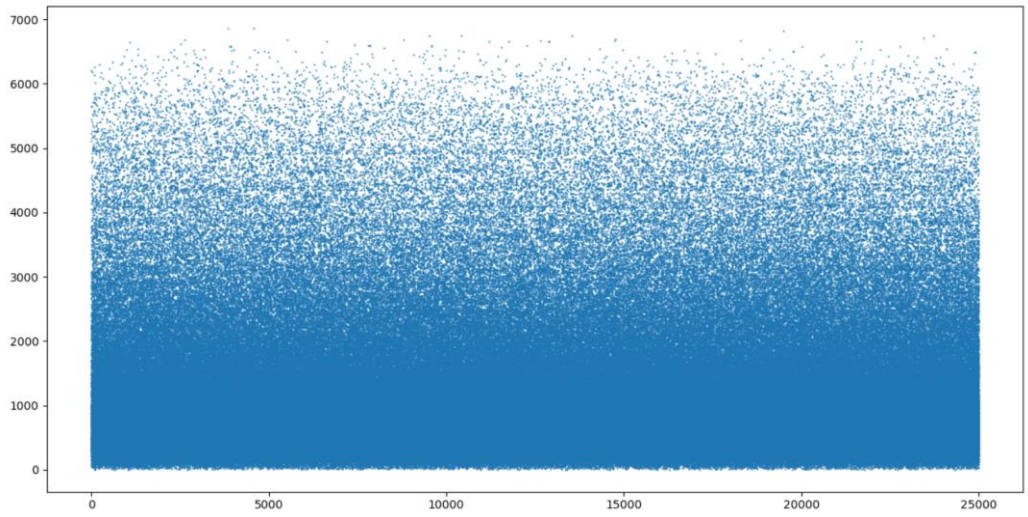


Wykres odchylenia standardowego oceny od numeru generacji

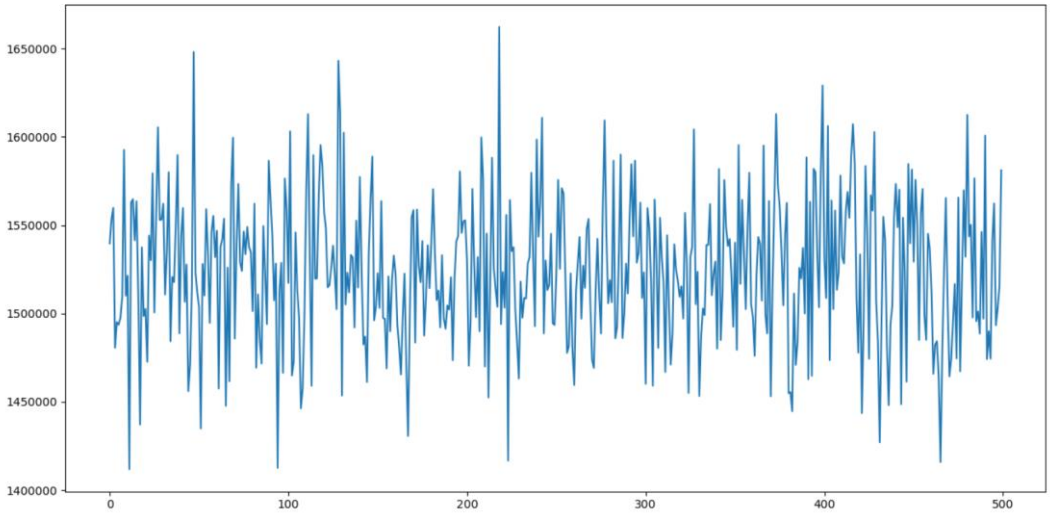
3. Duże prawdopodobieństwo mutacji

```
Enter population size: 50
Enter mutation probability: 0.5
Enter crossing probability: 0.7
Enter number of iterations: 500
Best found x =  [1, 3, 2, 0] Best evaluation =  2.0028103847344614
```

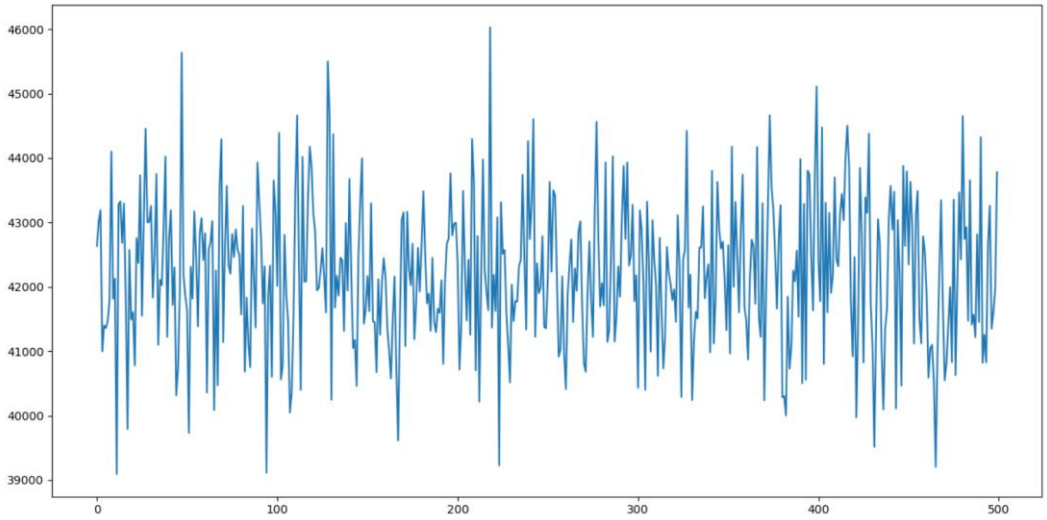
Dane startowe i znalezione minimum



Wytworzone osobniki i ich oceny na wykresie



Wykres średniej oceny populacji od numeru generacji



Wykres odchylenia standardowego oceny od numeru generacji

WNIOSKI

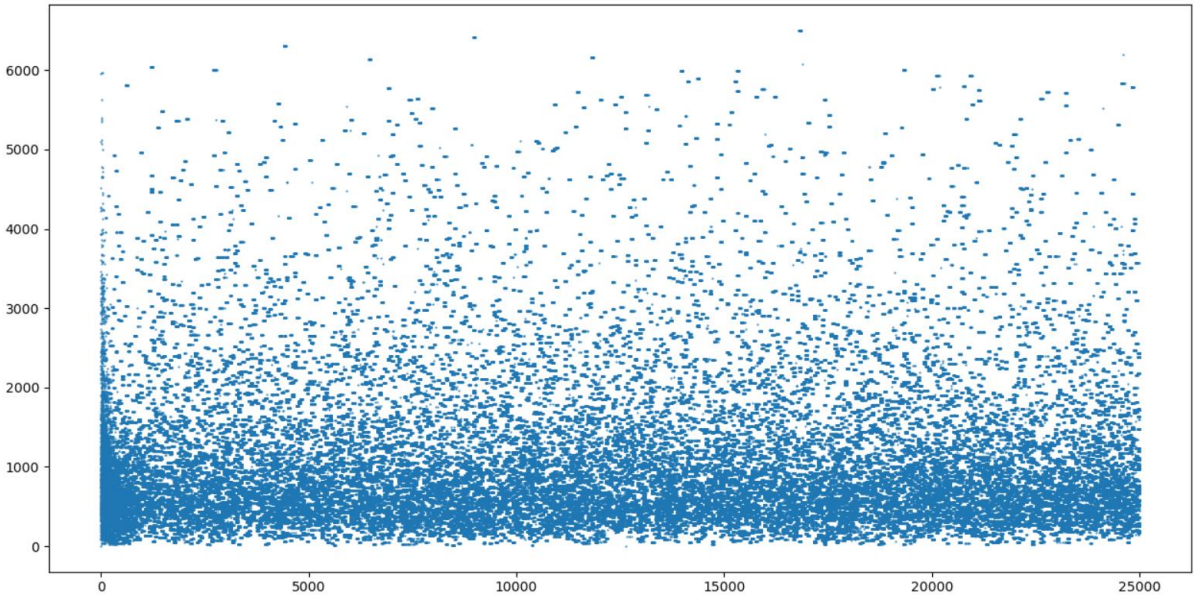
Jak można zauważyć algorytm zachowuje się najlepiej przy małym, ale nie bardzo małym prawdopodobieństwie mutacji. Przy dużym prawdopodobieństwie algorytm zyskuje ogromne zdolności eksploracyjne (co jest pożądane), ale dzieje się to kosztem zdolności eksploatacyjnych. Koszt jest tak duży, że trudno nazwać algorytm ewolucyjnym, gdyż bardziej przypomina on generowanie i sprawdzanie przypadkowych punktów. Przy bardzo małym prawdopodobieństwie natomiast, algorytm praktycznie nie ma zdolności eksploracyjnych.

Wpływ prawdopodobieństwa krzyżowania

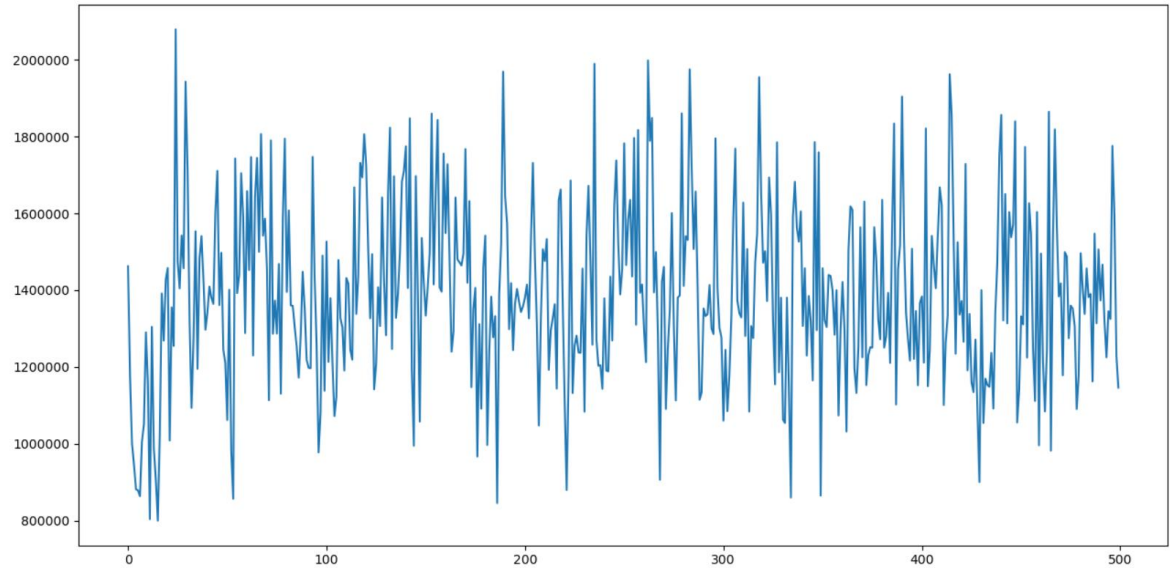
1. **Małe prawdopodobieństwo krzyżowania**

```
Enter population size: 50
Enter mutation probability: 0.01
Enter crossing probability: 0.01
Enter number of iterations: 500
Best found x =  [1, 3, 0, 1] Best evaluation =  2.989992496600445
```

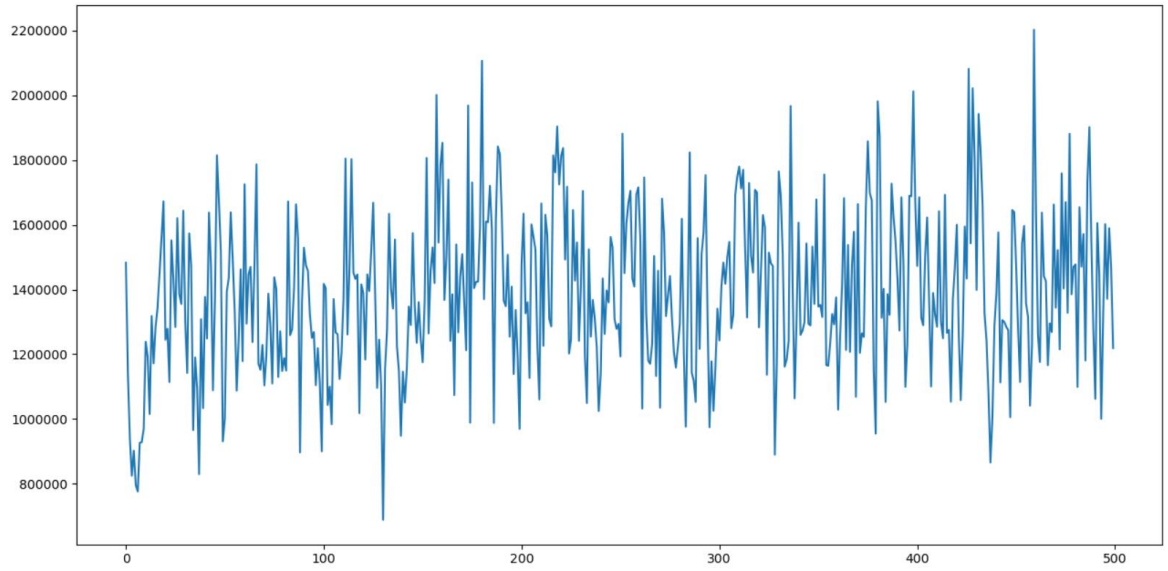
Dane startowe i znalezione minimum



Wytworzone osobniki i ich oceny na wykresie



Wykres średniej oceny populacji od numeru generacji

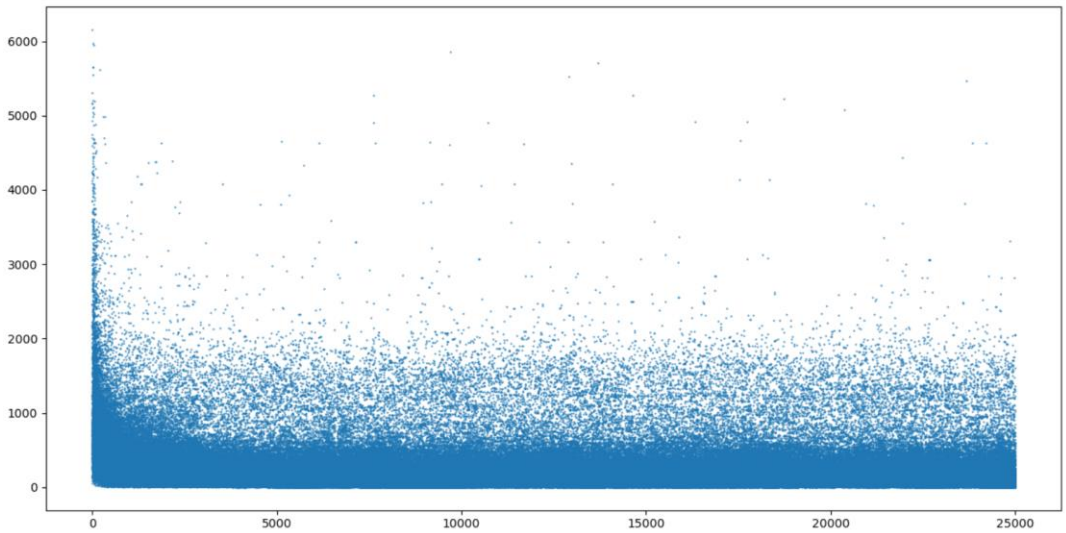


Wykres odchylenia standardowego oceny od numeru generacji

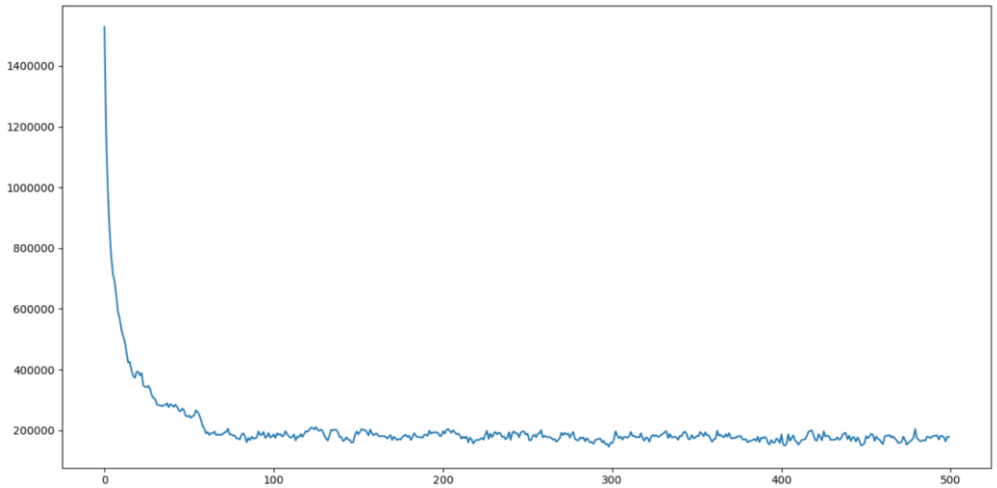
2. Duże prawdopodobieństwo krzyżowania

```
Enter population size: 50
Enter mutation probability: 0.01
Enter crossing probability: 0.7
Enter number of iterations: 500
Best found x =  [1, 3, 1, 1] Best evaluation =  0.9925037693693151
```

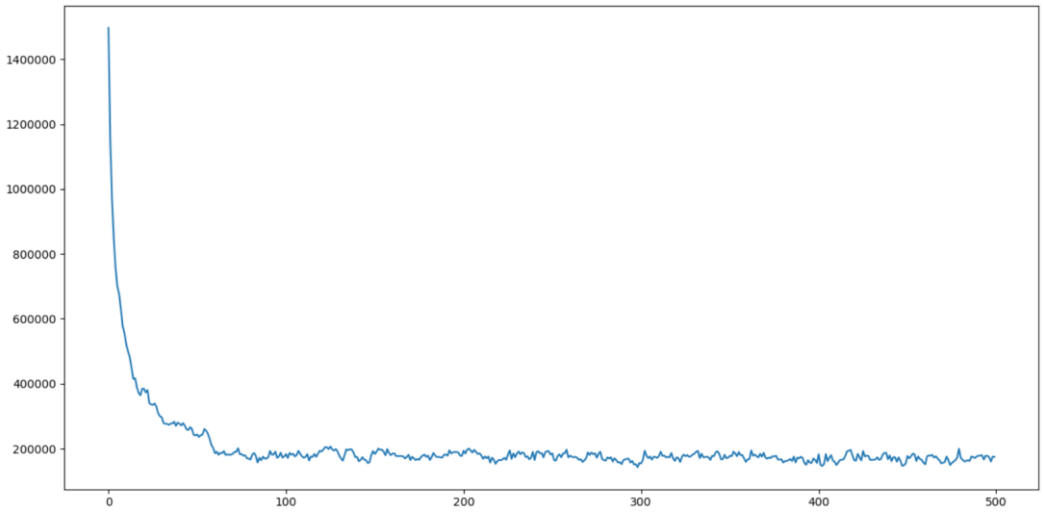
Dane startowe i znalezione minimum



Wytworzone osobniki i ich oceny na wykresie



Wykres średniej oceny populacji od numeru generacji



Wykres odchylenia standardowego oceny od numeru generacji

WNIOSKI

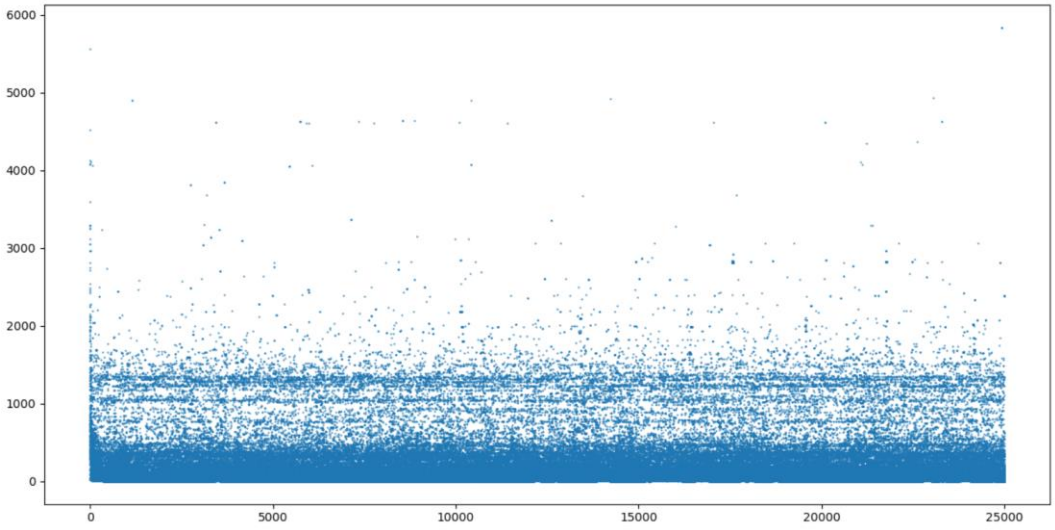
Najlepsze efekty dają duże wartości prawdopodobieństwa krzyżowania. Przy małym prawdopodobieństwie krzyżowania algorytm traci zdolności eksploatacyjne. Dzieje się tak dlatego, że nie tworzone są wtedy punkty średnie dobrych, znalezionych już punktów. Trudne jest więc znalezienie minimów lokalnych.

Wpływ wielkości populacji i ilości i teracji

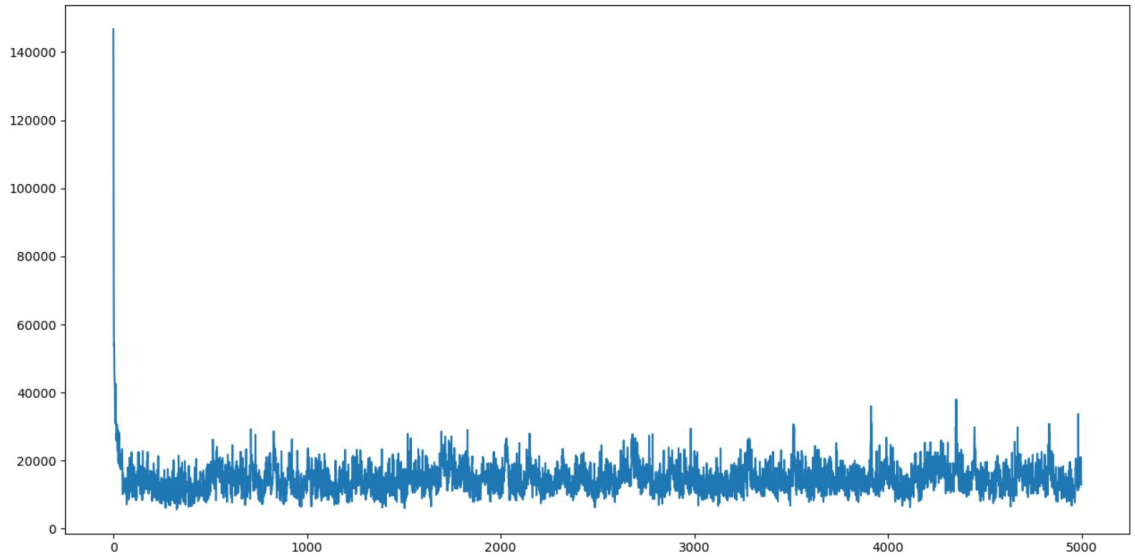
1. **Bardzo mała populacja**

```
Enter population size: 5
Enter mutation probability: 0.01
Enter crossing probability: 0.7
Enter number of iterations: 5000
Best found x =  [-15, 3, -15, 1] Best evaluation =  0.9925037693693152
```

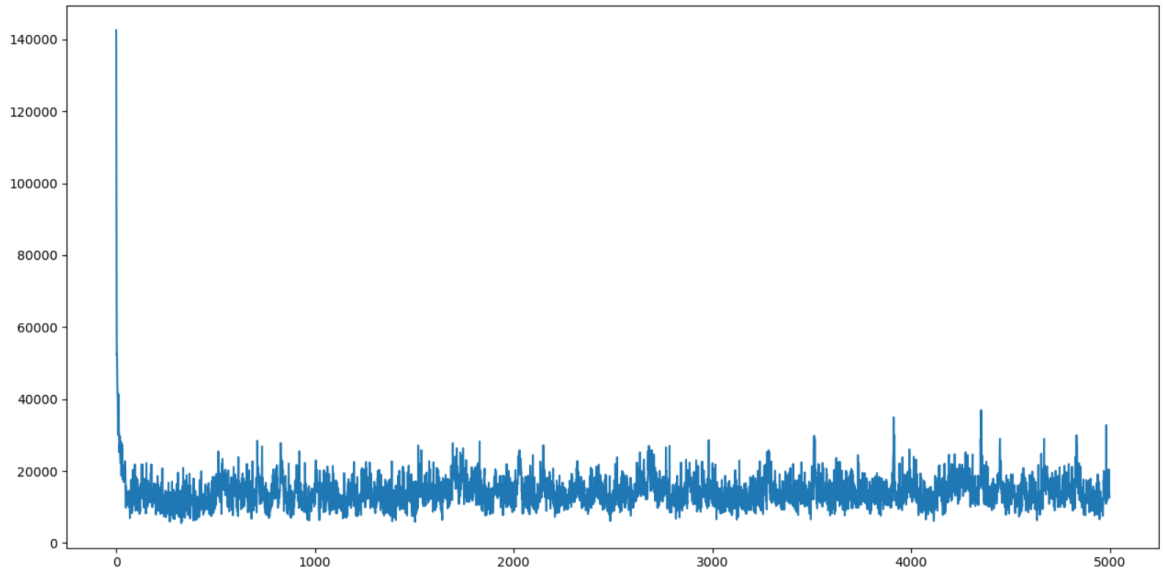
Dane startowe i znalezione minimum



Wytworzone osobniki i ich oceny na wykresie



Wykres średniej oceny populacji od numeru generacji

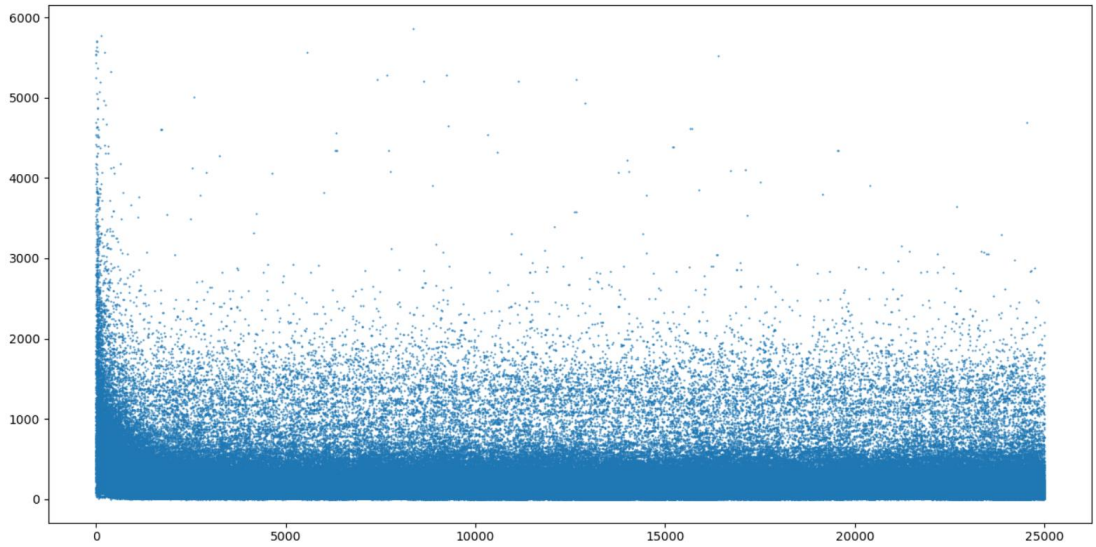


Wykres odchylenia standardowego oceny od numeru generacji

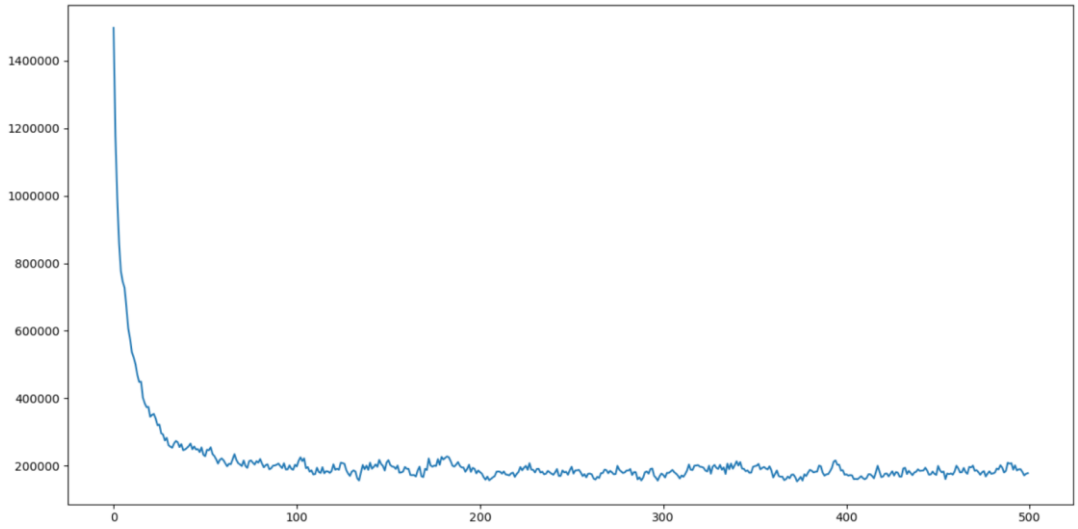
2. Mała populacja

```
Enter population size: 50
Enter mutation probability: 0.01
Enter crossing probability: 0.7
Enter number of iterations: 500
Best found x =  [1, 3, 1, 1] Best evaluation =  0.9925037693693152
```

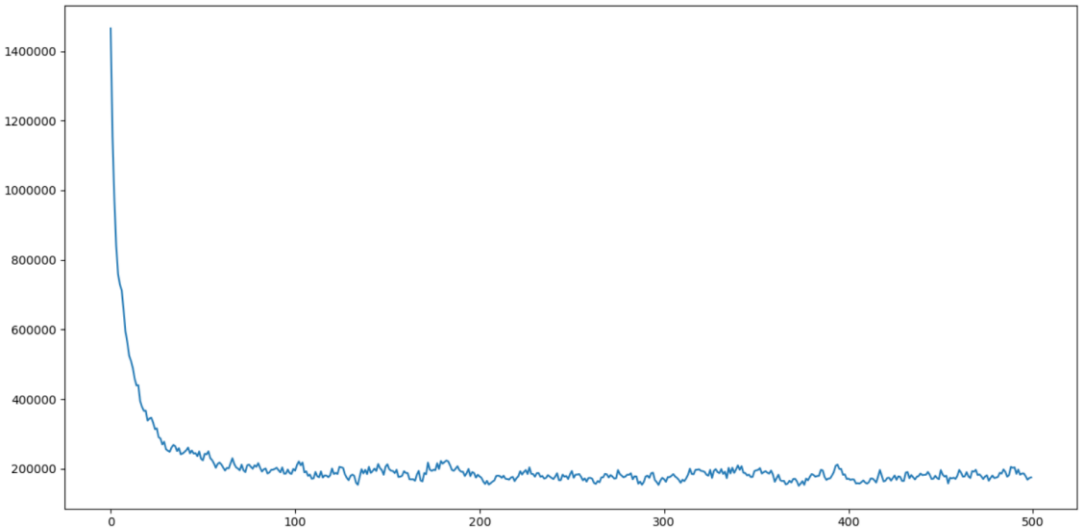
Dane startowe i znalezione minimum



Wytworzone osobniki i ich oceny na wykresie



Wykres średniej oceny populacji od numeru generacji

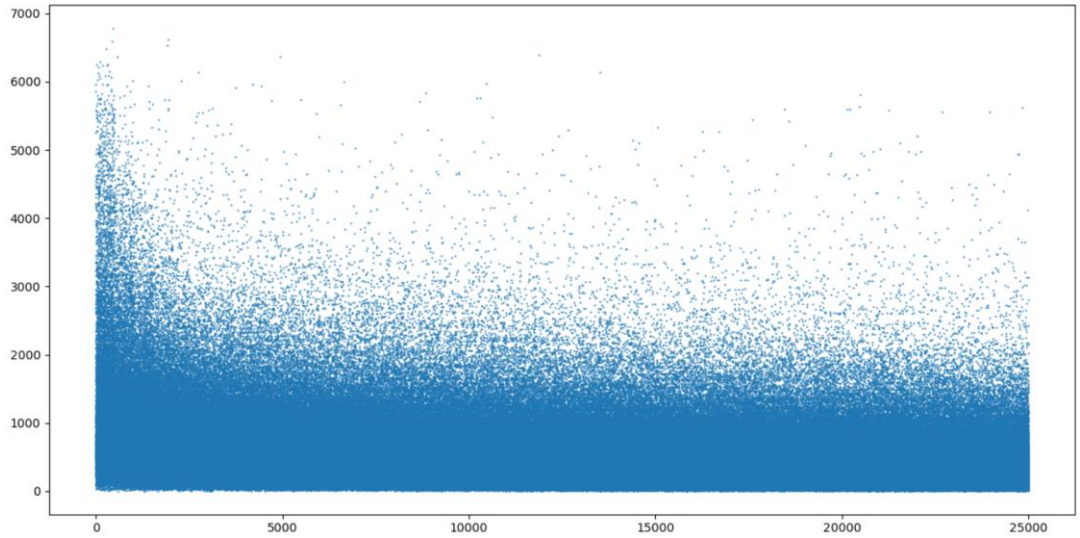


Wykres odchylenia standardowego oceny od numeru generacji

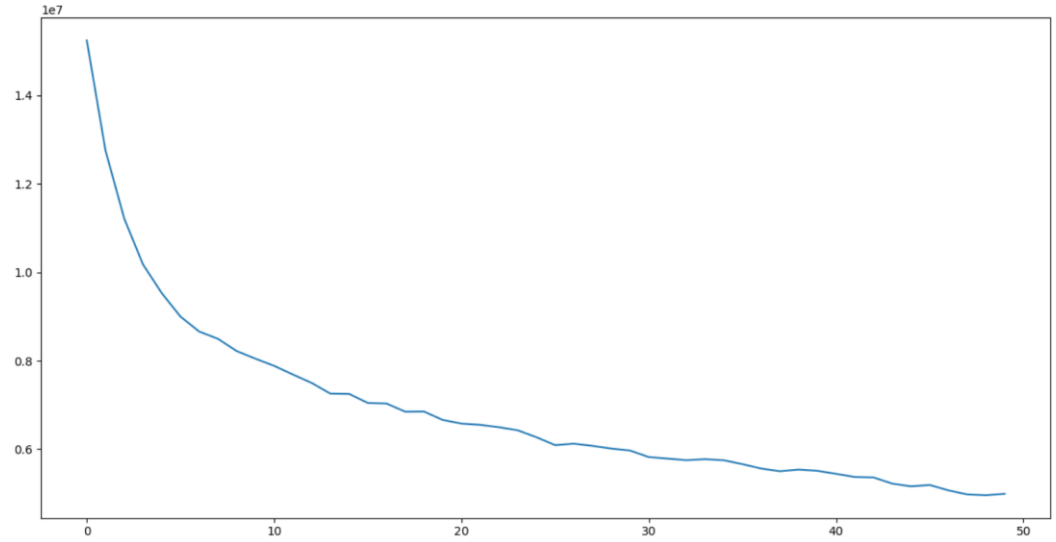
3. Duża populacja

```
Enter population size: 500
Enter mutation probability: 0.01
Enter crossing probability: 0.7
Enter number of iterations: 50
Best found x =  [1, 3, 1, 1] Best evaluation =  0.9925037693693152
```

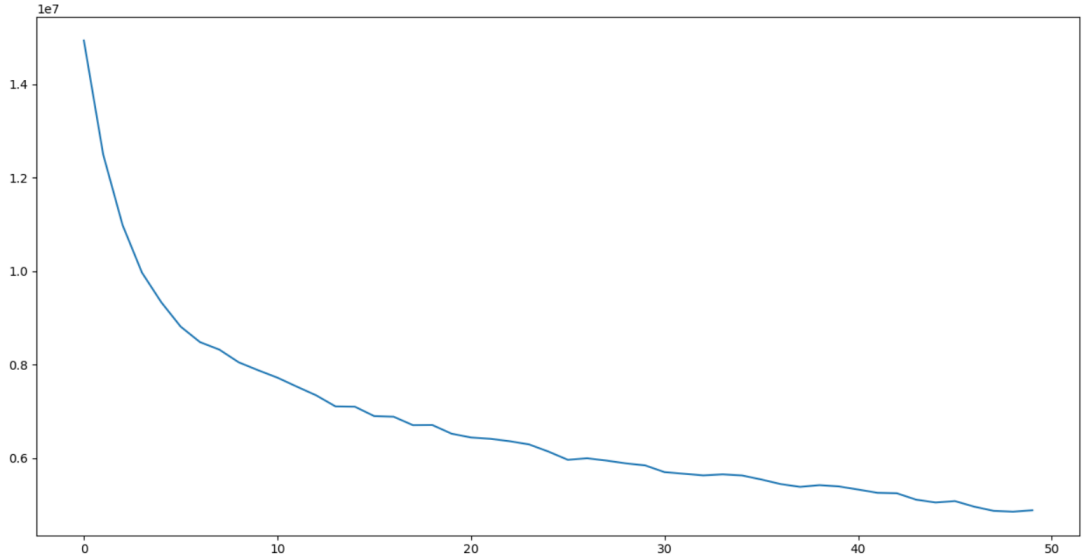
Dane startowe i znalezione minimum



Wytworzone osobniki i ich oceny na wykresie



Wykres średniej oceny populacji od numeru generacji



Wykres odchylenia standardowego oceny od numeru generacji

WNIOSKI

Eksperymenty przeprowadzono tak, by przy każdej z wybranych par wielkości populacji i ilości generacji, ilość wytworzonych osobników była taka sama. Możemy zauważyć, że najlepiej działa algorytm z populacją 50 osobników i ilością generacji równą 500. Dzieje się tak dlatego, że aby algorytm ewolucyjny działał poprawnie potrzebna jest populacja znacznie większa od wymiaru przestrzeni przeszukiwań i duża liczba generacji (na ogół co najmniej kilka set). Algorytmy z populacją porównywalną z wymiarem przestrzeni przeszukiwań (liczebność populacji, 5 liczba generacji 5000) mają za małą różnorodność genów więc za bardzo zależą od tego od jakiej populacji zaczynamy (znajdziemy najpewniej tylko minima lokalne bliskie początkowym osobnikom). Natomiast algorytmy z dużą populacją i małą liczbą generacji (liczebność populacji, 500 liczba generacji 50) mają za mało czasu (za mało generacji) by poprawnie przetworzyć początkową pulę genów i znaleźć minima lokalne.