Metody inteligencji obliczeniowej w analizie danych

Sprawozdanie z algorytmów ewolucyjnych.

Damian Skowroński — 313506

7 czerwca 2023

1 Wprowadzenie

Celem projektu była implementacja algorytmu genetycznego i sprawdzenie jego działania w trzech problemach:

- Znajdowanie minimum funkcji.
- Rozwiązanie problemu *cutting stock*.
- Optymalizacja wag w sieci MLP.

Dla każdego z tych problemów potrzebowałem zaimplementować nowy algorytm ewolucyjny, wobec tego w tym sprawozdaniu krótko opowiem jak działają poszczególne implementacje. Przedstawię również wnioski wyciągnięte podczas pracy z nimi.

2 Znajdowanie minimum funkcji

Celem było zaimplementować algorytm genetyczny z mutacją gaussowską i krzyżowaniem jednopunktowym w taki sposób, aby radził sobie ze znajdowaniem minimum dla funkcji. Testowałem działanie na:

- funkcji kwadratowej - $f(x, y, z) = x^2 + y^2 + 2z^2$
- pięciowymiarowej funkcji Rastrigina - $f(\mathbf{x}) = 50 + \sum_{i=1}^{5} \left(x_i^2 - 10\cos(2\pi x_i)\right)$

2.1 Implementacja

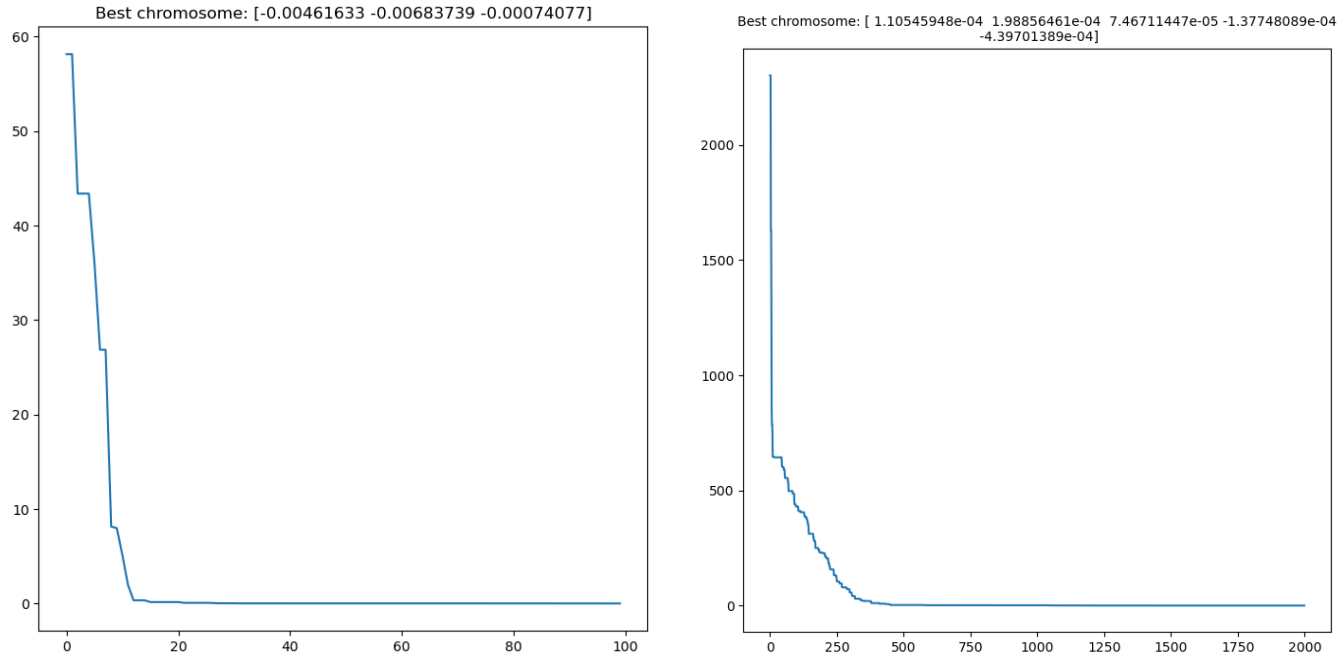
Algorytm zaimplementowałem jako klasę. Przy inicjalizacji należy podać argumenty:

- `chromosome_size` - w tym przypadku ile wymiarowa jest funkcja
- `population_size` - z ilu członków będzie składała się populacja
- `fitness_function` - funkcja, którą algorytm ma maksymalizować, żeby minimalizować wystarczy podać $\frac{1}{f(\mathbf{x})}$
- `set_chromosomes` - (opcjonalne) początkowe wartości dla populacji, domyślnie są losowane z rozkładu $U(-1, 1)$

Po inicjalizacji instancji algorytmu wystarczy użyć metody `evolve`, która wykorzystuje:

- mutację gaussowską - wybiera kilka członków z populacji (w zależności od `mutation_coef`) i dla każdego dodaje próbkę z rozkładu $N(0, \sigma)$, gdzie σ zależy od `mutation_coef` i numeru generacji - im późniejsza generacja tym mniejsza mutacja
- krzyżowanie jednopunktowe - wybiera wielokrotnie dwóch członków i zamienia ze sobą część ich wartości, liczba zmian jest zależna od `crossover_coef`
- elityzm - do nowej generacji wybierani są członkowie z prawdopodobieństwem w zależności od ich wyniku *fitness*, `elite_coef` reguluje ile członków jest wybierane w ten sposób, reszta populacji jest dopełniana poprzez krzyżowanie elity

Metodzie `evolove` należy podać wcześniej wspomniane: `mutation_coef`, `crossover_coef`, `elite_coef`, a także docelową liczbę generacji `generations_number`.



(a) funkcja kwadratowa

(b) pięciowymiarowa funkcja Rastrigina

Rysunek 1: Wykresy wartości funkcji w najlepszym znalezionym punkcie w zależności od numeru generacji.

2.2 Test działania

Wyniki testu działania zostały przestawione na rysunku 1. Obie funkcje mają minimum globalne w punkcie o wszystkich współrzędnych równych 0. Wówczas dla obu funkcji w takim punkcie jest wartość 0. W tytułach wykresów przedstawione są najlepsze znalezione współrzędne i widać, że są one bliskie 0. Podobnie wartości funkcji w znalezionych punktach szybko zbliżyły się do 0. Przy funkcji Rastrigina wymagało to więcej generecji ze wsględu na jej liczne minima lokalne. W każdym razie widać, że algorytm działa dobrze.

3 Cutting stock

Problem polegał na wypełnianiu kół o różnej średnicy, różnej wielkości prostokątami. Każdy prostokąt ma swoją wartość. Celem jest aby w kole była łącznie jak największa wartość. Zasady wypełniania koła prostokątami są następujące:

- boki wszystkich prostokątów muszą być równoległe do osi układu
- prostokąty nie mogą na siebie nachodzić, ale mogą się stykać bokami
- prostokąty w całości muszą mieścić się w obszarze koła
- nie ma ograniczeń co do liczby prostokątów

3.1 Implementacja

W skrócie moja implementacja polega na skupieniu się na *gęstości* koła zamiast łącznej sumy wartości prostokątów w kole jako wartość do maksymalizowania. Wobec tego każdy rodzaj prostokąta ma swoją *gęstość* równą ilorazowi wartości i pola. Algorytm ma populację chromosomów. Każdy chromosom ma swój zbiór obecnych prostokątów.

Nowe prostokąty są generowane w następujących krokach:

1. dla obecnego w zbiorze prostokąta zostaje wylosowana kolejność boków, na których potencjalnie pojawi się nowy prostokąt, oraz przemieszczenie względem środka obecnego prostokąta
2. losowany jest nowy prostokąt - prawdopodobieństwo wylosowania konkretnego rodzaju prostokąta jest zależne od jego *gęstości* - im większa tym bardziej on się opłaca, więc jest większe prawdopodobieństwo, że zostanie wybrany
3. w zależności od wcześniej wylosowanej kolejności boków algorytm próbuje na nich wygenerować nowy prostokąt
4. nowy prostokąt przy wygenerowaniu zawsze będzie stykał się z obecnym prostokątem, jednak będzie przesunięty od jego środka wzdłuż stycznego boku

5. nowy prostokąt zostaje przyjęty do zbioru prostokątów, jeśli nie nachodzi na żaden prostokąt obecny w zbiorze oraz w całości mieści się w kole

W każdej generacji algorytm próbuje wygenerować jeden nowy prostokąt dla każdego obecnego w zbiorze prostokąta. Mutacja polega na próbie przesunięcia każdego z prostokątów oddzielnie w stronę środka. W trakcie przesunięcia na początku prostokąt próbuje ruszyć się pionowo, a następnie poziomo. Może wydarzyć się żadna, jedna, lub obie z tych operacji w zależności, czy przesunięty prostokąt spełnia założenia zadania. Na podstawie takich mutacji środek koła powinien być coraz bardziej *zagęszczany*. Algorytm nie wykorzystuje krzyżowania, ponieważ nie wymyśliłem żadnej krzyżówki, która nie działałaby negatywnie na *gęstość* koła.

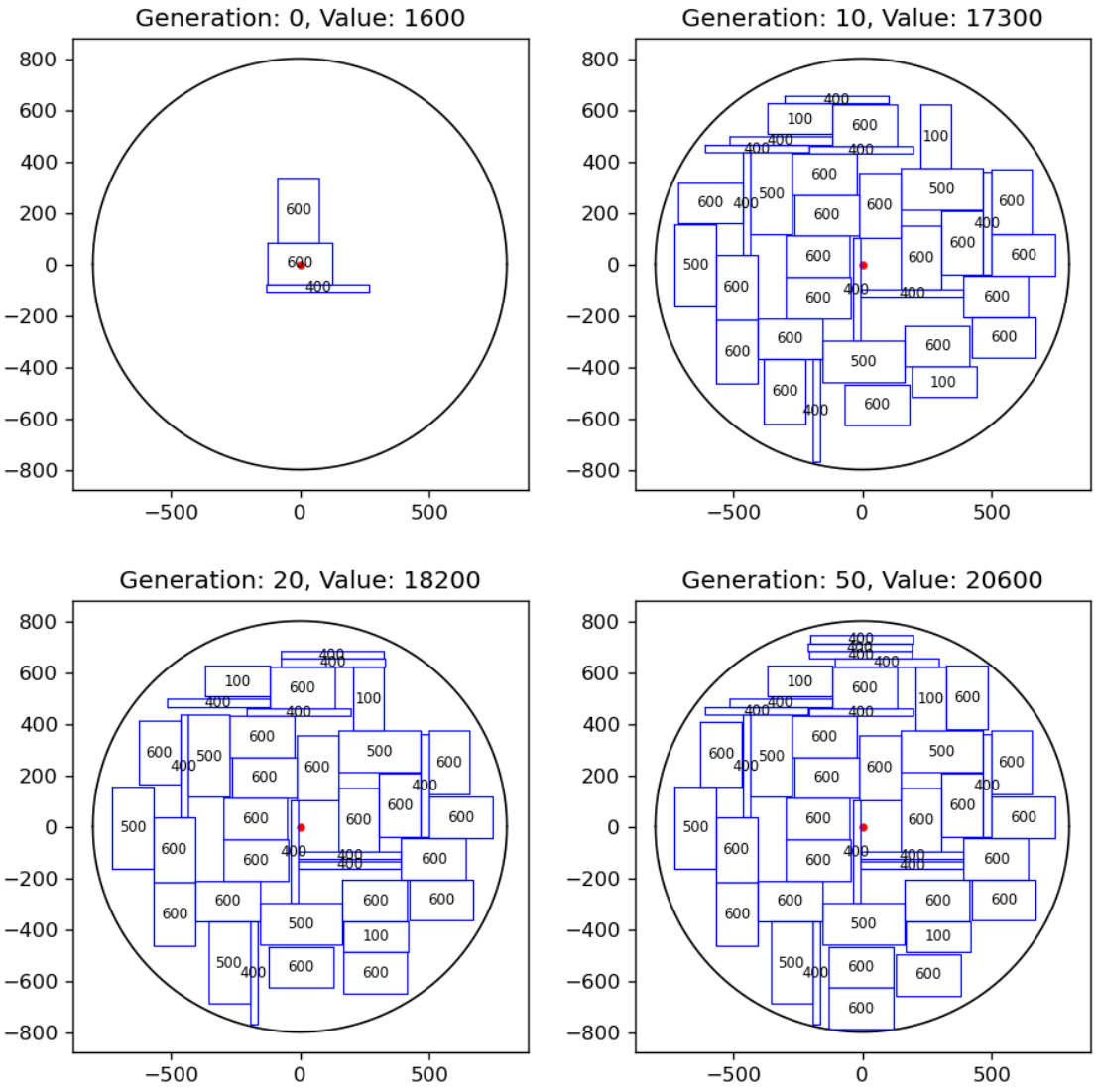
Minimalizowana *gęstość* koła jest liczona w następujący sposób:

1. w zbiorze prostokątów znajdowany jest jeden róg, który ogólnie jest najdalszym punktem od środka
2. odległość tego punktu od środka jest promieniem *aktywnego* koła
3. wszystkie prostokąty ze zbioru znajdują się w całości w obszarze *aktywnego* koła
4. *gęstość* danego chromosomu to iloraz sumy wartości wszystkich obecnych prostokątów i pola *aktywnego* koła

Z tak zdefiniowaną *gęstością* najlepsze chromosomy to takie, które najbardziej optymalnie mieszczą prostokąty wokół środka. Na podstawie *gęstości* wybierani są też członkowie na zasadzie elityzmu. Nowa generacja ma wiele powtarzających się najlepszych członków z poprzedniej generacji, ale zmutują się oni w inny sposób.

3.2 Przykład działania algorytmu

Na rysunku 2 widać działanie algorytmu dla zbioru *r800* - koło o promieniu 800 i 5 rodzajów prostokątów (zbiór będzie lepiej opisany w następnej sekcji).



Rysunek 2: Wykresy wypełniania koła prostokątami przez algorytm. W tytułach wykresów generacja i suma wartości.

Generacja 0 pokazuje jak algorytm zaczyna swoje działanie - generuje prostokąt w pobliżu środka koła, a następnie degenerowuje 2 prostokąty na swoich bokach.
Po 10 generacjach widać, że najbardziej *gęste* ustawienie zostało zastąpione. W tym momencie nadal jest dużo miejsca, w którym mogą zostać wygenerowane nowe prostokąty. Niektóre obecne prostokąty mają też miejsce do przesuwania się w stronę środka.
Po 20 generacjach ustawienie jest podobne do poprzedniego, ale na pewno nie jest jego potomkiem. Można zobaczyć między innymi to po tym, że z lewej strony koła prostokąt przy krawędzi z wartością 600 ma inną orientację.
Po 50 generacjach widać, że prawie całe koło zostało zapełnione. Nie jest to optymalne rozwiązanie - wciąż pozostaje wiele pustego miejsca.

3.3 Test działania

Testy działania algorytmu zostały przeprowadzone na 5 różnych zbiorach: <
(reprezentacja prostokątów <*szerokość*>x<*wysokość*>w<*wartość*>)

- *r800* - koło o promieniu 800, prostokąty: 250x120w100, 320x160w500, 250x160w600, 150x120w40, 400x30w400
- *r1200* - koło o promieniu 1200, prostokąty: 200x120w200, 200x160w300, 250x160w500, 100x120w40
- *r1000* - koło o promieniu 1000, prostokąty: 200x120w200, 200x160w300, 250x160w500, 100x120w40
- *r1100* - koło o promieniu 1100, prostokąty: 250x120w100, 120x360w300, 250x160w600, 150x120w40
- *r850* - koło o promieniu 850, prostokąty: 10x120w120, 120x10w150, 400x20w1200, 300x30w1200, 120x120w1200, 100x100w900, 450x80w11000

3.3.1 r800

Dla tego zbioru próg sumy wartości oznaczający dobre działanie algorytmu wynosił 30000. Na tym zbiorze najtrudniej było osiągnąć oczekiwany wynik. Wymagało to spróbowania kilku różnych podejść:

1. wykorzystanie wszystkich rodzai prostokątów, pozwolenie na rotacje o 90 stopni
2. wykorzystanie wszystkich rodzai prostokątów, nie używanie rotacji
3. wykorzystanie tylko najbardziej opłacalnego prostokąta, pozwolenie na rotacje o 90 stopni
4. wykorzystanie tylko najbardziej opłacalnego prostokąta, nie używanie rotacji

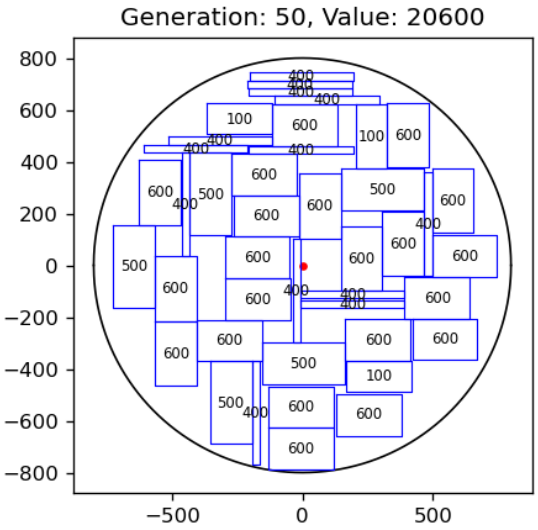
Wyniki kolejnych podejść są przedstawione na rysunku 3. Dla każdego podejścia użyłem tylko 50 generacji. Dla podejścia pierwszego wynik 20600 jest zdecydowanie niezadowalający. Widać, że rotacje prostokątów długich i wąskich tworzą duże wolne przestrzenie. Dla podejścia drugiego nieco lepszy wynik 22180, choć nadal niezadowalający. Widać trochę lepsze zagęszczenie. Dla podejścia trzeciego wynik 40800 znacząco lepszy i z łatwością pokonuje próg 30000. Widać dużo wolnego miejsca, nadal można by dodać kilka prostokątów innego rodzaju do zwiększenia wyniku. W podejściu czwartym najlepszy wynik 43200. Tutaj widać zdecydowanie najlepsze zagęszczenie środka - prawie nie ma wolnego miejsca. Dużo wolnego miejsca zostało jeszcze na brzegach koła.

3.3.2 r1200

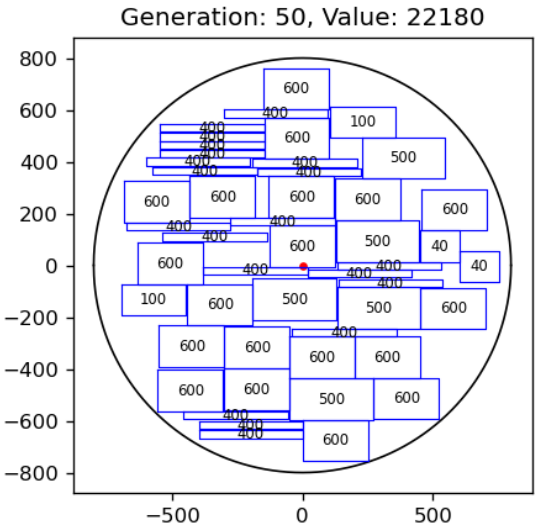
Dla tego zbioru próg znowu sumy wartości znowu wynosił 30000. Dla tego zbioru przekroczenie progu nie było problematyczne. Wypróbowałem dwa podejścia:

1. wykorzystanie wszystkich rodzai prostokątów, pozwolenie na rotacje o 90 stopni
2. wykorzystanie wszystkich rodzai prostokątów, nie używanie rotacji

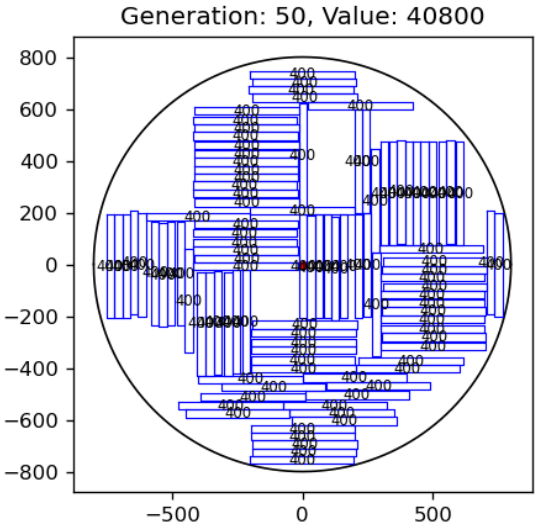
Wyniki dla obu podejść są bardzo podobne (rysunek 4). Ze względu na charakterystykę prostokątów, rotacja tu nie ma dużego znaczenia.



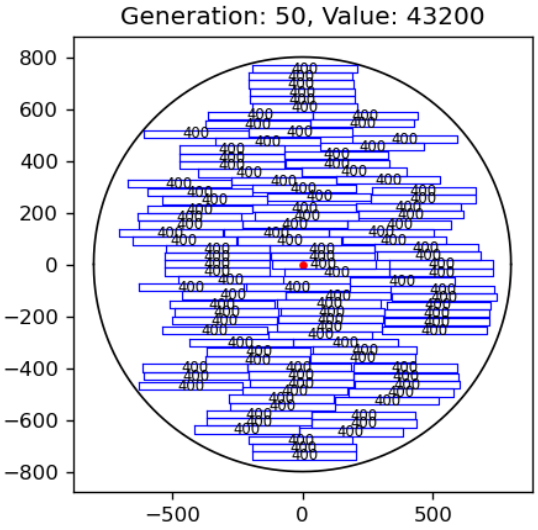
(a) wszystkie prostokąty + rotacja



(b) wszystkie prostokąty + brak rotacji

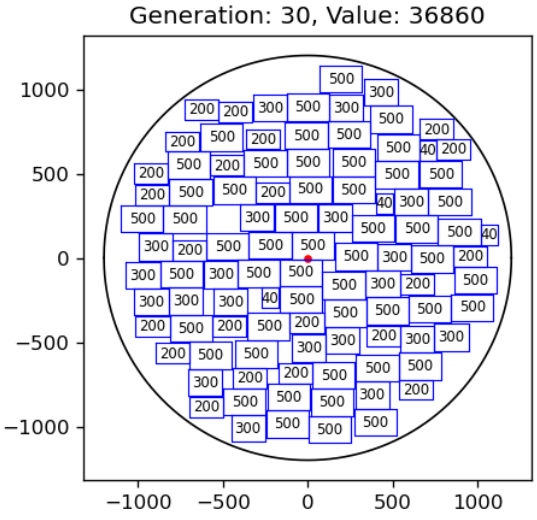


(c) najlepszy prostokąt + rotacja

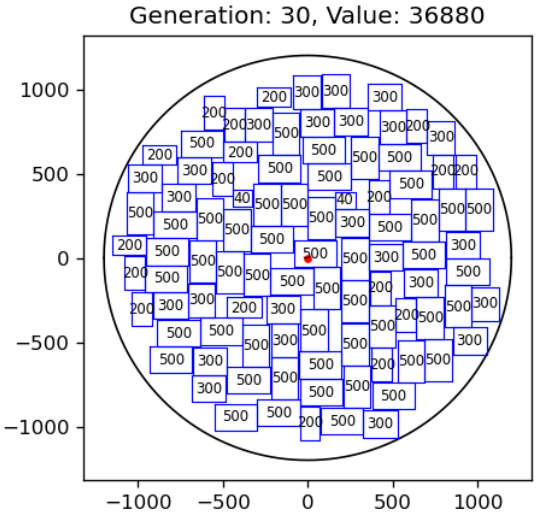


(d) najlepszy prostokąt + brak rotacji

Rysunek 3: Wykresy wypełniania koła prostokątami przez algorytm w zależności od podejścia dla zbioru *r800*.



(a) wszystkie prostokąty + rotacja

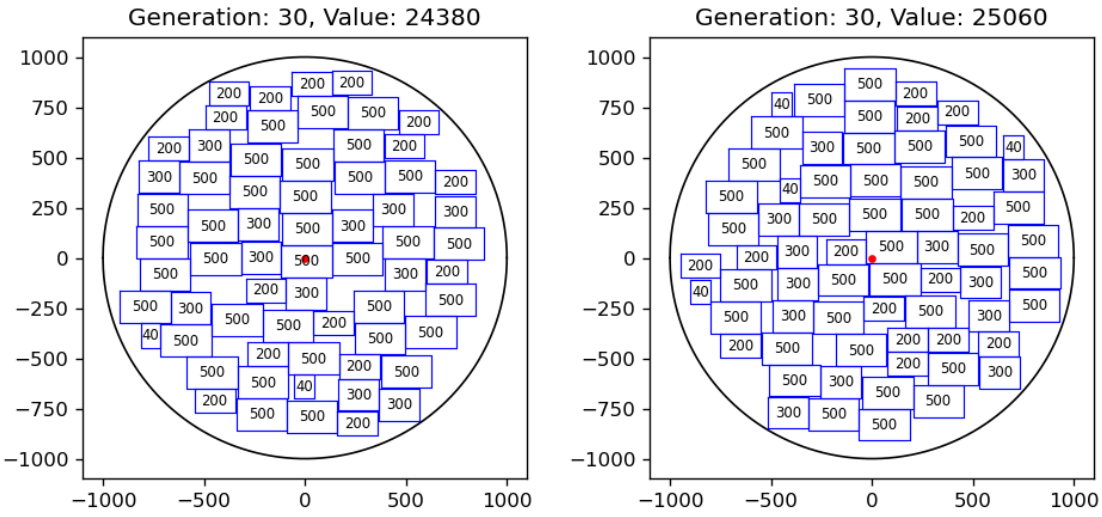


(b) wszystkie prostokąty + brak rotacji

Rysunek 4: Wykresy wypełniania koła prostokątami przez algorytm w zależności od podejścia dla zbioru *r1200*.

3.3.3 r1000

Dla tego zbioru próg sumy wartości wynosił 17500. Możliwe rodzaje prostokątów są takie same jak w *r1200*, więc znowu rotacje nie miały dużego znaczenia. Dla obu podejść wyniki (rysunek 5) wyszły podobne, oba z łatwością przekraczają próg.



(a) wszystkie prostokąty + rotacja

(b) wszystkie prostokąty + brak rotacji

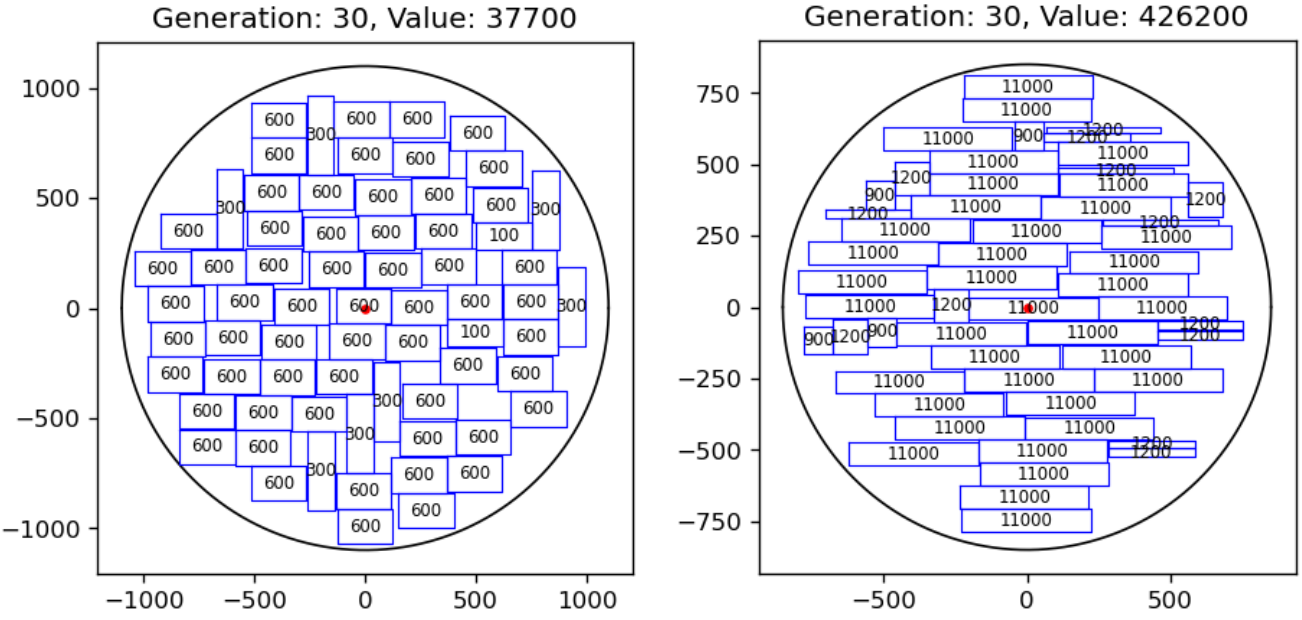
Rysunek 5: Wykresy wypełniania koła prostokątami przez algorytm w zależności od podejścia dla zbioru *r1000*.

3.3.4 r1100

Tutaj próg wynosił 25000. Dla wstawienia dopuszczającego wszystkie prostokąty i nie pozwalającego na rotacje nie było trudne przekroczyć próg (rysunek 6 (a))

3.3.5 r850

Dla tego zbioru nie było podanego progu. Wyjątkowym faktem dla tego zbioru jest to, że jeden prostokąt jest znacząco bardziej opłacalny niż pozostałe. Jest to sprzyjające dla mojej implementacji algorytmu, która faworyzuje prostokąty w zależności od ich opłacalności. Wynik przedstawiony na rysunku 6 (b).



(a) wszystkie prostokąty + brak rotacja dla zbioru *r1100*

(b) wszystkie prostokąty + brak rotacji dla zbioru *r850*

Rysunek 6: Wykresy wypełniania koła prostokątami przez algorytm w zależności od zbioru.

4 Optymalizacja wag w sieci MLP

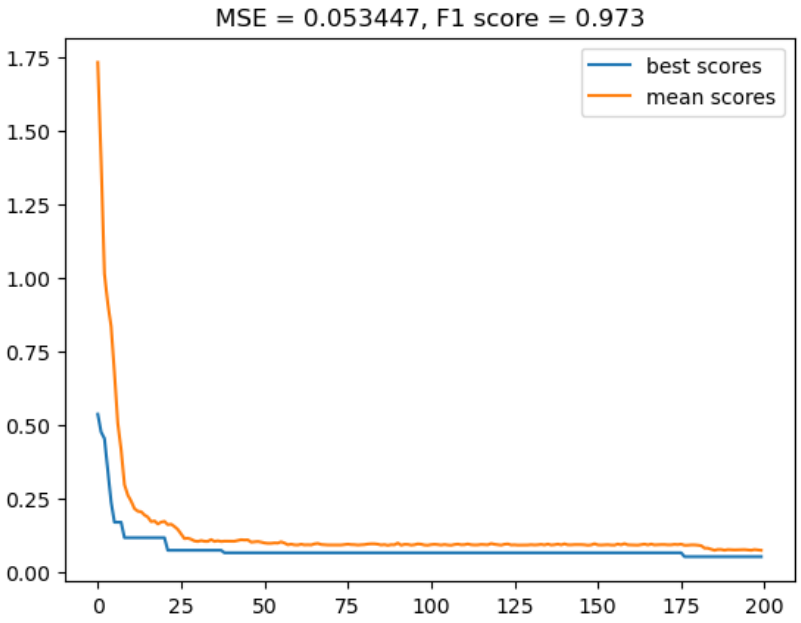
Celem było zaimplementowanie sieci MLP (*Multi-Layer Perceptron*), która wykorzystuje algorytm ewolucyjny do aktualizacji wag. Implementacja wykorzystuje:

- mutację do zmiany wag i biasów
- krzyżowania do wymiany warstw pomiędzy dwoma instancjami MLP
- elityzm - zawsze 10% populacji w następnej generacji to najlepsze dotychczasowe rozwiązanie

Test działania przeprowadziłem na zbiorach *iris*, *multimodal-large-training*, *auto-mpg*. Patrzyłem tylko na to czy sieć się potrafi dopasować, nie zwracałem uwagi na przterenowanie i podział na zbiór testowy i treningowy. Funkcją minimalizowaną było zawsze MSE, niezależnie od tego czy problem jest regresji czy klasyfikacji.

4.1 iris

Na rysunku 7 widać, że zaimplementowana sieć potrafi się nauczyć do prostego zbioru.



Rysunek 7: Wyniki uczenia sieci do zbioru *iris*

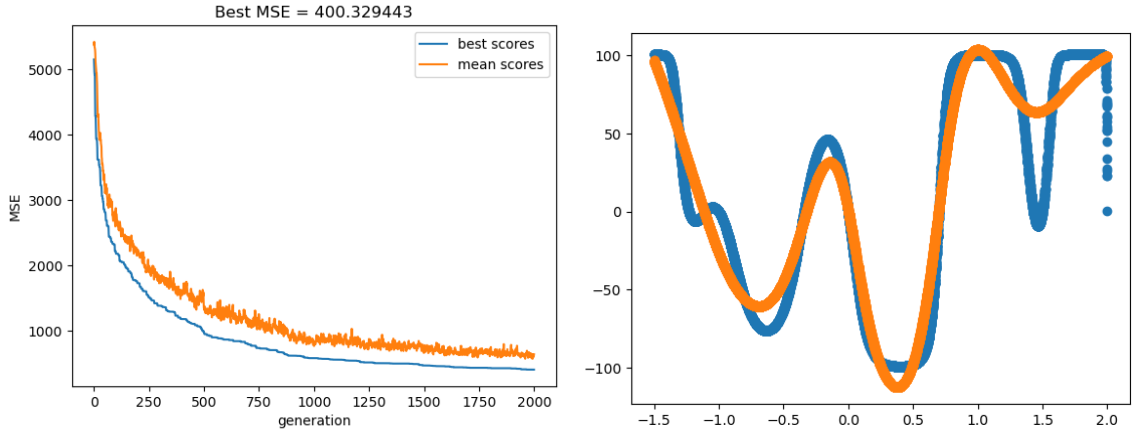
4.2 multimodal-large

Ten zbiór ma 10000 obserwacji więc uczenie sieci trwa dużo dłużej. Przetestowałem cztery ustawienia:

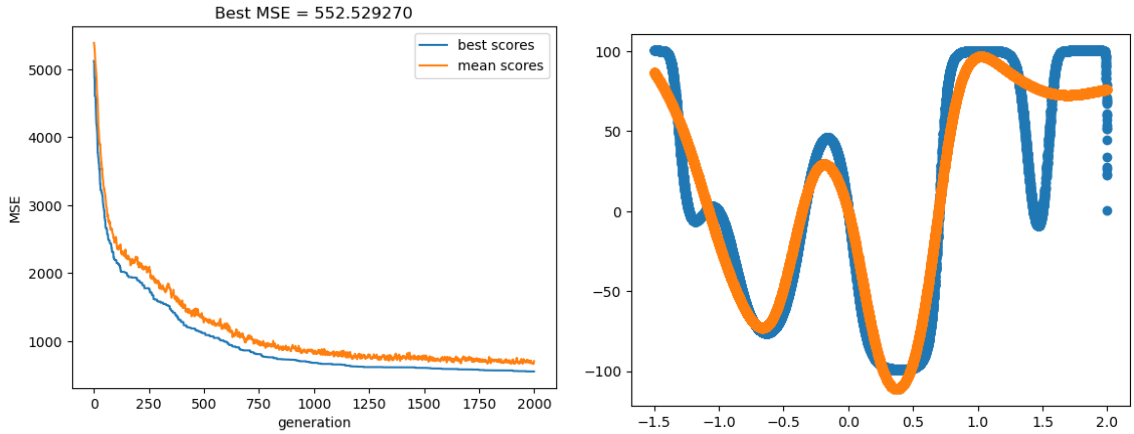
1. architektura sieci - 1 warstwa ukryta o 100 neuronach, `mutation_coef`=0.7, `crossover_coef`=0.2
2. architektura sieci - 1 warstwa ukryta o 100 neuronach, `mutation_coef`=0.3, `crossover_coef`=0.1
3. architektura sieci - 3 warstwy ukryte po 20 neuronów, `mutation_coef`=0.7, `crossover_coef`=0.2
4. architektura sieci - 3 warstwy ukryte po 20 neuronów, `mutation_coef`=0.3, `crossover_coef`=0.1

Dla każdego z ustawień rozmiar populacji wynosił 100, a algorytm przeszedł przez 2000 generacji. Na poniższych rysunkach widać wyniki dla kolejnych ustawień. Ogólnie można zobaczyć, że dla każdego ustawienia sieć potrafiła się dopasować, ale nie było to tak dobre dopasowanie jak przy użyciu propagacji wstecznej. Wybranie odpowiedniej architektury i współczynników może mieć duże znaczenie na wydajność sieci. Najlepszy wynik otrzymałem dla ustawienia pierwszego (rysunek 8). Można zobaczyć też pewne różnice w działaniu pomiędzy różnymi ustawieniami. Im większe wartości współczynników tym bardziej średnie wyniki są "oddalone" od najlepszych i mają większą wariancje. Architektura też ma wpływ na różnicę najlepszych i średnich wyników. Podejrzewam, że dla rysunku 10 jest większa różnica niż dla rysunku 8 pomimo tych samych współczynników, ponieważ ta architektura ma więcej wartości w warstwach ukrytych, które można zmieniać.

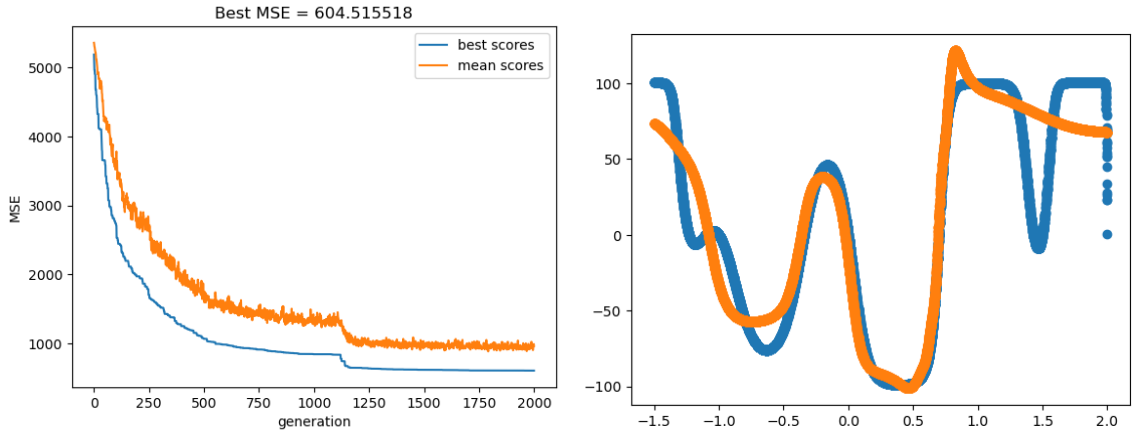
Wniosek z korzystania algorytmu ewolucyjnego w sieci MLP jest taki, że może to działać, ale mniej efektywnie i dużo wolniej, niż klasyczne podejście propagacji wstecznej.



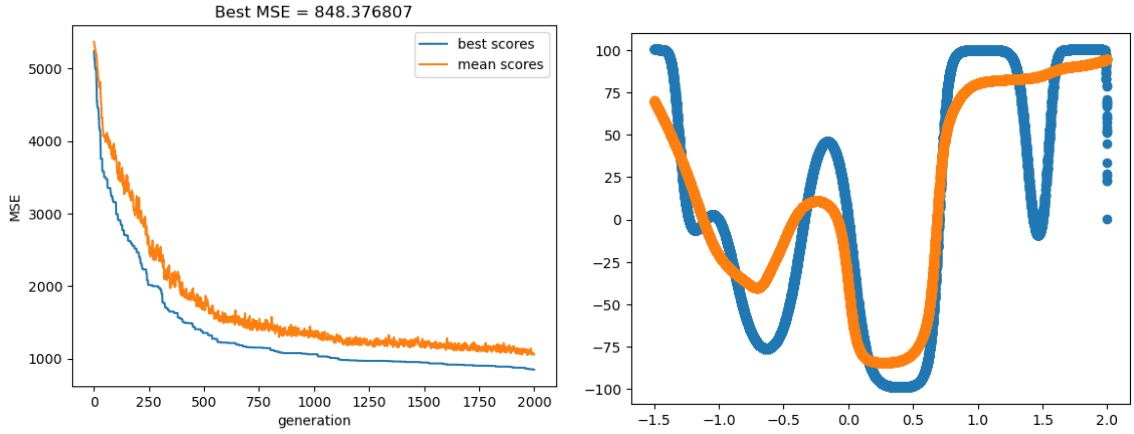
Rysunek 8: Ustawienie: 1 warstwa ukryta o 100 neuronach, mutation_coef=0.7, crossover_coef=0.2



Rysunek 9: Ustawienie: 1 warstwa ukryta o 100 neuronach, mutation_coef=0.3, crossover_coef=0.1



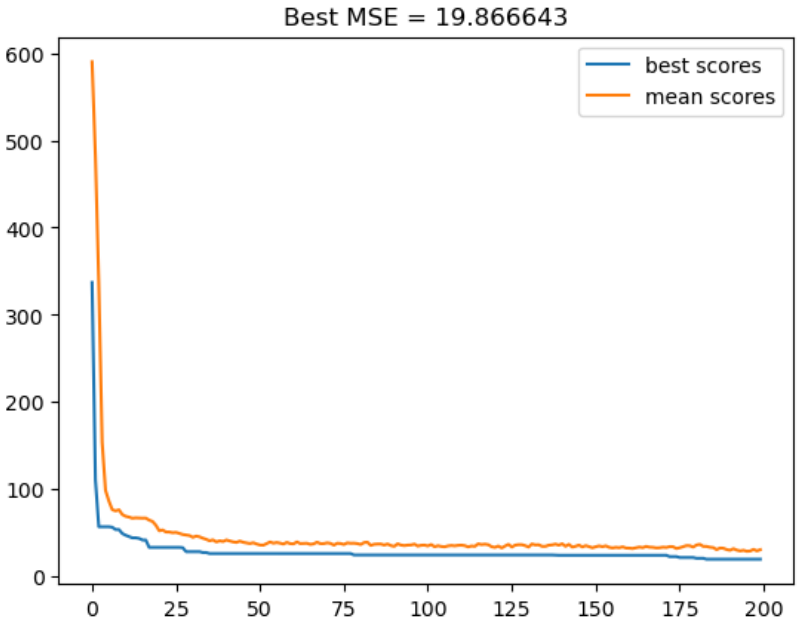
Rysunek 10: Ustawienie: 3 warstwy ukryta po 20 neuronów, mutation_coef=0.7, crossover_coef=0.2



Rysunek 11: Ustawienie: 3 warstwy ukryta po 20 neuronów, mutation_coef=0.3, crossover_coef=0.1

4.3 auto-mpg

W tym zbiorze należało wykonać zadanie regresji. Wynik z użycia sieci można zobaczyć na rysunku 12.



Rysunek 12: Wyniki uczenia sieci do zbioru *auto-mpg*

5 Podsumowanie

W ramach tego projektu zaimplementowałem algorytm ewolucyjny w celu rozwiązywaniu 3 różnych problemów. Algorytm świetnie się spisał w zadaniu szukania minimum globalnego funkcji, potrafił dać satysfakcjonujący wynik w problemie *cutting stock* i dawał radę optymalizować wagi w sieci MLP. Generalnie pokazuje to, że można z niego korzystać w wielu problemach wymagających optymalizacji, jednak zwykle nie jest on najlepszą opcją. Dodatkowo nieprzyjemnym jest fakt, że tak naprawdę dla każdego nowego problemu trzeba wykonać oddzielną indywidualną implementację.