

# **Algorytmy ewolucyjne**

Zasada działania, obszary zastosowań, biblioteki programistyczne, zastosowanie do wybranego problemu optymalizacyjnego oraz eksperymenty

Artur Janusz Franasowicz

Piotr Tomasz Kotarba

# 1. Zasada działania i zastosowania

Algorytmy ewolucyjne są wzorowane na mechanizmach biologicznej ewolucji, w celu optymalizacji konkretnych zadań lub modelowania zagadnień. Algorytm ewolucyjny nie jest jakimś pojedynczym sposobem na rozwiązanie określonego problemu, ale określeniem wszystkich algorytmów inspirowanych darwinowską zasadą doboru naturalnego. Dobór naturalny, zwany inaczej selekcją naturalną jest jednym z mechanizmów ewolucji biologicznej, ukierunkowany na jak najlepszą adaptację do warunków środowiskowych. Miarą tej adaptacji jest dostosowanie (ang. *fitness*), który może być rozpatrywany w kontekście konkretnego osobnika lub poszczególnego genu. Organizmy, które mają korzystniejsze cechy mają większe szanse na przeżycie i rozmnażanie, co prowadzi do poprawienia średniego przystosowania w populacji. Dokładnie na tej obserwacji zachowania natury oparto algorytmy ewolucyjne, i starając się naśladować działanie przyrody, te algorytmy składają się z mechanizmów selekcji, reprodukcji i mutacji. Zadany problem, który rozwiązywany jest przez algorytm ewolucyjny możemy sobie wyobrazić jako środowisko, w którym żyje populacja. Każdy osobnik powinien reprezentować jakieś potencjalne(konkretnie) rozwiązanie problemu, lepsze bądź gorsze z punktu widzenia przystosowania (np. lepsze pod względem kosztu tzn. tańsze lub lepiej zoptymalizowane obliczeniowo). Algorytm ewolucyjny dzięki krzyżowaniu i mutacji powinien tworzyć stopniowo coraz to lepsze i bardziej przystosowane do środowiska osobniki, aż do osiągnięcia pewnego akceptowalnego, określonego z góry zadowalającego poziomu przystosowania lub do określonego pokolenia.

Ważnymi pojęciami stosowanymi w algorytmice ewolucyjnej są:

- **chromosom** – składa się z genów będących jednostkami genotypu osobnika
- **populacja** – grupa osobników, z których każdy stanowi jakieś rozwiązanie problemu dla algorytmu ewolucyjnego
- **genotyp** – składa się z chromosomów, a przynajmniej jeden z chromosomów musi zawierać informację kodującą fenotyp
- **fenotyp** – zespół cech, właściwości konkretnego osobnika w środowisku określona przez interakcję genotypu ze środowiskiem, w naszym przypadku będą to parametry rozwiązania
- **funkcja celu** – funkcja zadana w postaci pewnego systemu świata rzeczywistego o dowolnej złożoności
- **funkcja przystosowania** – nie musi być identyczna z funkcją celu, określa poziom przystosowania osobnika do zadanych warunków, funkcja celu zawsze jest składową funkcji przystosowania
- **populacja początkowa** – generowana losowo lub już z jakimiś określonymi cechami (gdy nasza wiedza o problemie jest na tyle duża, że pozwala nakierować bądź przyspieszyć działanie ewolucji)
- **selekcja** – proces, który faworyzuje osobniki o lepszym przystosowaniu, dzięki któremu osobniki lepiej przystosowane powinny mieć większą szansę na rozmnożenie i przekazanie cech następnym pokoleniom niż gorzej przystosowane

- **rekombinacja** – proces wymiany materiału genetycznego poprzez wymieszanie chromosomów rodziców i przekazanie genów osobnikowi potomnemu
- **mutacja** – losowe zaburzenie genotypu potomka, najlepiej jeżeli te zaburzenia będą niewielkie (zbyt duża rozpiętość zakresu mutacji może doprowadzić do tego, że to mutacje, a nie dobór naturalny będą miały decydujący wpływ na rezultat działania algorytmu ewolucyjnego)

Algorytmy ewolucyjne możemy podzielić na:

- **algorytmy genetyczne** szukają najlepszego rozwiązania za pomocą heurystyki, czyli za każdym razem rezultat ich działania może być odmienny, w praktyce stosujemy je głównie gdy algorytm pełny nie jest znany lub gdy ten algorytm jest zbyt kosztowny obliczeniowo i nie jest możliwe uzyskanie rozwiązania za jego pomocą w sensownym czasie
- **programowanie genetyczne** pozwala automatycznie stworzyć program, który rozwiąże jakieś zagadnienie, w oparciu o jakąś definicję
- **programowanie ewolucyjne** różni się od innych podejść do algorytmów ewolucyjnych tym, że nie zachodzi wymiana materiału genetycznego pomiędzy osobnikami, do tworzenia nowych osobników wykorzystujemy tylko mutację
- itd. ...

Budowę algorytmu ewolucyjnego możemy określić za pomocą następującego pseudokodu z komentarzami:

```
t:= 0 //licznik czasowy, iteracji, populacji
ustal P(t); //ustal lub wygeneruj populację początkową
ocień P(t); //ocień populację początkową
dopóki nie warunek zakończenia //pętla główna
{
  P1(t) := reprodukuj P(t); //wybierz osobniki do reprodukcji
  P2(t) := rekombinuj P1(t); //wymień materiał genetyczny
  P3(t) := mutuj P2(t); //wprowadź mutacje dla potomków
  ocień P3(t); //ocień uzyskaną populację potomków
  P(t+1) := sukcesja (P3(t)UP4(t)); //zastąp stare osobniki nowymi
  t := t + 1; //inkrementuj licznik
}
```

Obszarem zastosowań algorytmów ewolucyjnych jest każdy problem, w którym nie istnieje algorytm zupełny zdolny do podania rozwiązania, albo gdy ilość potencjalnych rozwiązań jest tak ogromna, że nie jest możliwe obliczenie takiego rozwiązania bądź rozwiązań w sensownym czasie. Algorytmy ewolucyjne są jedną z technik probabilistycznych, stosowanych do rozwiązywania takich właśnie problemów. Znalazły zastosowanie w ogromnej ilości dziedzin, między innymi w medycynie, planowaniu i harmonogramowaniu, oprogramowaniu typu CAD, nawigacji satelitarnej, planowaniu tras, optymalizowaniu kosztów transportu, narzędziach wspomagania decyzji ekonomicznych, planowaniu tras robotów, sztucznej inteligencji i uczeniu maszynowym oraz w symulacjach z dziedziny sztucznego życia. Algorytmy ewolucyjne mają tak ogromny przedział zastosowań, że nie sposób wymienić chociażby ich ułamka w sprawozdaniu.

## 2. Biblioteki programistyczne

### 2.1 JENETICS

Jenetics jest zaawansowaną biblioteką napisaną w języku Java, skonstruowaną do pracy z algorytmami genetycznymi, algorytmami ewolucyjnymi oraz do programowania genetycznego. Wykorzystuje wszystkie przedstawione pomysły i pojęcia związane z algorytmami ewolucyjnymi, takie jak *gen*, *chromosom*, *genotyp*, *fenotyp* itd. (przedstawione w 1 rozdziale sprawozdania). Została napisana z myślą o wykorzystaniu strumieni wprowadzonych w Javie od wersji Java SE 8. Aktualnie *EvolutionStream* implementuje interfejs *Stream*, więc możemy używać strumieni jak w przypadku każdego innego ich zastosowania w celu wykonania kolejnych kroków ewolucyjnych. Pozwala to wykorzystać zalety strumieni, tj. przejrzystość operacji i uniknięcie „*Anti-Arrow-Pattern*”. Jenetics jest dostępny w repozytorium Mavena.

#### I) Instalacja, konfiguracja i uruchomienie „Hello World”

Aby dodać bibliotekę Jenetics do projektu maven’owego, należy dodać odpowiednie zależności do pliku konfiguracyjnego *pom.xml*. My używaliśmy Jenetics w wersji 7.0.0. Zależności potrzebne do uruchomienia pierwszego programu z [manuala](#) wyglądają następująco:

```
<dependencies>

  <!-- https://mvnrepository.com/artifact/io.jenetics/jenetics -->
  <dependency>
    <groupId>io.jenetics</groupId>
    <artifactId>jenetics</artifactId>
    <version>7.0.0</version>
  </dependency>

</dependencies>
```

Maven szybko pobrał niezbędne biblioteki i moduły. Instalacja okazała się bardzo prosta. Bez problemu uruchomił się również pierwszy program testowy:

```
C:\Users\artfp\.jdk\azul-17.0.3\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2021.3.2\bin" -Dfile.encoding=UTF-8 -classpath C:\Users\artfp\.m2\repository\io\jenetics\jenetics\7.0.0\jenetics-7.0.0.jar HelloWorld
Hello World:
[00000011|11111111]

Process finished with exit code 0
```

#### II) Zalety Jenetics

Do zalet biblioteki Jenetics zaliczyć można brak potrzeby dociągania innych bibliotek do jej działania, możliwość pracy równoległej(wielowątkowej), dobrą dokumentację, możliwość konfigurowania klasy *Engine*, wspomnianą wcześniej obsługę strumieni i lambd i obsługę optymalizacji wielocelowej, bardzo ważnej w przypadku algorytmów ewolucyjnych.

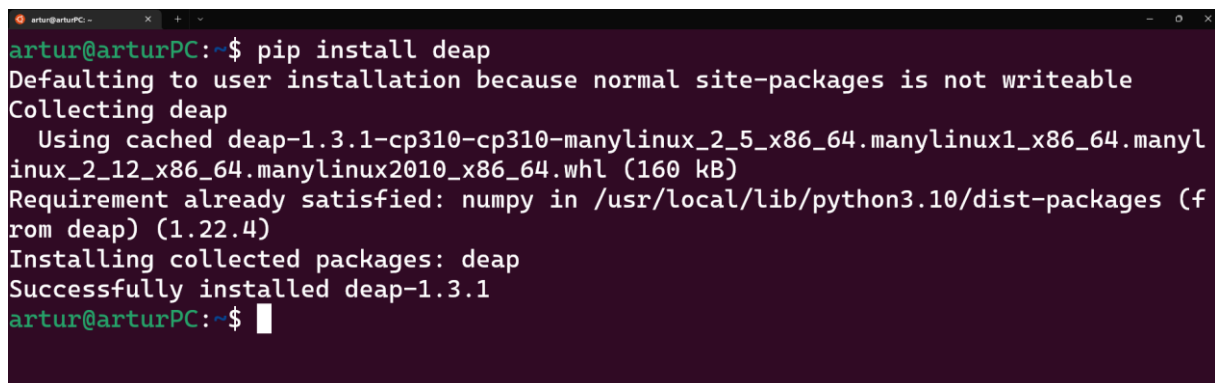
## 2.2 DEAP

DEAP (*Distributed Evolutionary Algorithms in Python*) jest framework’iem obliczeń ewolucyjnych przeznaczonym do szybkiego prototypowania i testowania pomysłów w Pythonie. Jest rozwijany od 2009 roku w Uniwersytecie Laval w Quebecu. Został on napisany głównie za pomocą języka Python, ale posiada również wstawki z C i C++. Zawiera następujące funkcje:

- algorytmy genetyczne wykorzystujące struktury danych używane w Pythonie, tj. listy, krotki, zbiory, słowniki
- programowanie genetyczne z wykorzystaniem drzew przedrostków:
  - silne typowane, słabo typowane
  - funkcje zdefiniowane automatycznie
- strategie ewolucyjne (w tym CMA-ES)
- optymalizacja wielu celów (NSGA-II, NSGA-III, SPEA2, MO-CMA-ES)
- algorytmy koewolucyjne
- zrównoleglenie ocen
- galeria sław „Hall of Fame” zawiera najlepiej przystosowane osobniki
- możliwość utworzenia punktów kontrolnych regularnie wykonywujących migawki systemu
- moduł zawierający najpopularniejsze funkcje testowe

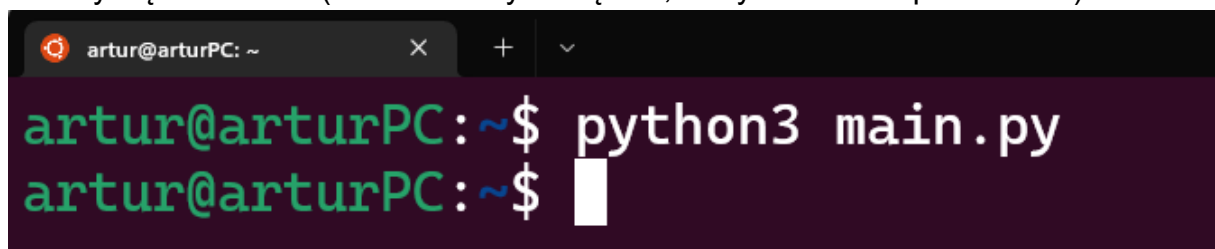
### I) Instalacja i konfiguracja

Instalacja DEAP ogranicza się do wykonania komendy `pip install deap`:



```
artur@arturPC:~$ pip install deap
Defaulting to user installation because normal site-packages is not writeable
Collecting deap
  Using cached deap-1.3.1-cp310-cp310-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux2_12_x86_64.manylinux2010_x86_64.whl (160 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from deap) (1.22.4)
Installing collected packages: deap
Successfully installed deap-1.3.1
artur@arturPC:~$
```

Uruchomienie programu przykładowego importującego moduły DEAP’a ze [strony](#) kończy się sukcesem (nie ma żadnych błędów, wszystko działa prawidłowo):



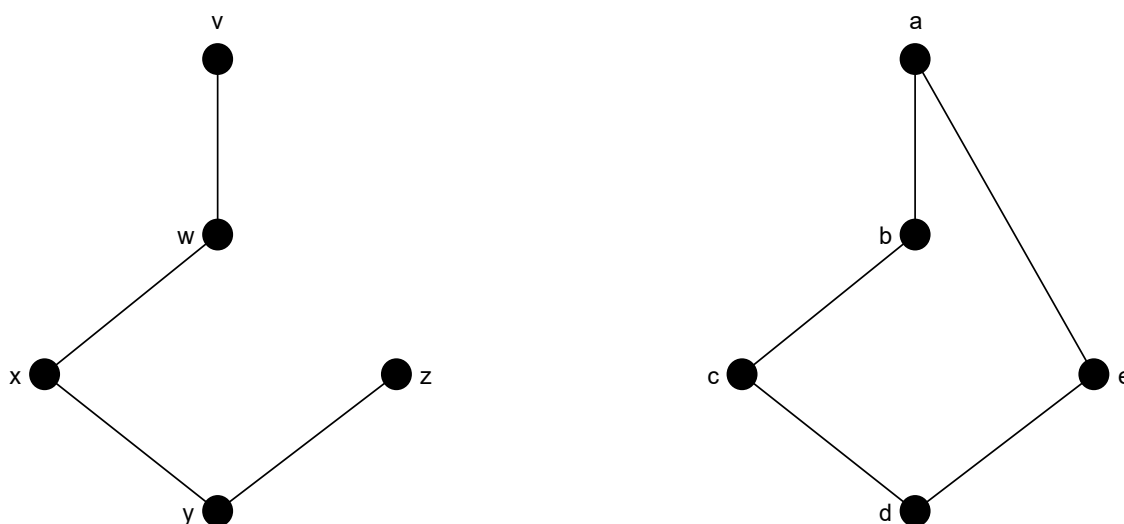
```
artur@arturPC:~$ python3 main.py
artur@arturPC:~$
```

Jak widzimy, instalacja DEAP’a jest bardzo prosta. Wystarczyło wykonać jedną komendę w terminalu, aby móc uruchomić przykładowy program korzystający z tego framework’a. Interpreter nie zwrócił komunikatu błędu i znalazł wszystkie potrzebne moduły. Jak widzimy, obydwa narzędzia bardzo łatwo przygotować do działania.

### 3. Zaimplementowany algorytm rozwiązujący wybrany problem optymalizacyjny

**Problem komiwojażera** (ang. *travelling salesman problem*, *TSP*) polega na odnalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Za pierwszego autora, który rozważał powyższy problem uznaje się austriackiego matematyka Karla Mengera, który zdefiniował go w 1930 roku. Ponieważ problem ten jest bardzo prosty w zrozumieniu a jednocześnie nie istnieje żaden efektywny algorytm znajdowania najmniej kosztownych cykli Hamiltona, problem komiwojażera zyskał bardzo na popularności. W następnym akapicie objaśnimy pojęcia związane z tym problemem.

Graf to zbiór wierzchołków połączonych krawędziami w taki sposób, że każda krawędź kończy i zaczyna się w którymś z wierzchołków, wierzchołki mogą być numerowane lub oznaczane literami. Graf ważony to taki graf, w którym przypisujemy krawędziom bądź wierzchołkom etykiety służące do przechowywania dodatkowych informacji. W przypadku naszego rozważania problemu komiwojażera przypiszemy koszt podróży między dwoma wierzchołkami jako odległość odcinka je łączącego wyliczonego z twierdzenia Pitagorasa (graf będzie opisany na dwuwymiarowym układzie współrzędnych). Oznacza to, że rozważamy **symetryczny problem komiwojażera (STFP)**, czyli taki w którym odległość z punktu A do punktu B zawsze będzie tak samo kosztowna jak odległość z punktu B do punktu A. Warto również zwrócić uwagę na różnicę między **drogą Hamiltona** a **cyklem Hamiltona**. Drogę nazywamy drogą Hamiltona, jeśli przechodzi przez każdy wierzchołek grafu dokładnie jeden raz. Drogę zamkniętą (czyli taką, której pierwszy i ostatni wierzchołek jest równy), która przechodzi przez wszystkie wierzchołki grafu dokładnie jeden raz, z wyjątkiem ostatniego wierzchołka, którym ponownie jest pierwszy wierzchołek, nazywamy cyklem Hamiltona. Graf mający cykl Hamiltona nazywamy grafem hamiltonowskim. Różnice między drogą a cyklem Hamiltona pokazujemy na poniższym rysunku poglądowym:



Graf po lewej, opisany kolejnością przechodzenia przez wierzchołki vwxyz ma drogę Hamiltona, ale nie posiada cyklu Hamiltona, gdyż nie ma powrotu do punktu początkowego v. Graf po prawej ma cykl Hamiltona opisany drogą *abcdea*.

W przypadku naszego rozważanego problemu będą nas interesować wyłącznie minimalne cykle Hamiltona. Głównym problemem w STFP jest liczba kombinacji sposobu poprowadzenia krawędzi tak, aby powstał cykl Hamiltona. Liczba wszystkich możliwych cykli Hamiltona do utworzenia dla  $n$  punktów wynosi  $\frac{(n-1)!}{2}$ . Oczywiście jesteśmy w stanie napisać program, który sprawdzi wszystkie cykle Hamiltona i wskaże najmniej kosztowny, ale może on dawać rezultaty w rozsądnym czasie tylko dla niewielkiej liczby punktów do rozważenia. Dla przykładu dla 10 różnych punktów liczba wszystkich cykli Hamiltona do rozważenia wynosi 181 440, a dla 40 różnych punktów taka liczba wynosi w przybliżeniu  $1.02 \times 10^{46}$ . Jak widzimy, nie jest możliwe sprawdzenie takiej liczby rozwiązań w rozsądnym czasie, stąd problem komiwojażera idealnie nadaje się do zastosowania algorytmów ewolucyjnych.

W celu rozwiązania problemu skorzystaliśmy z programu dostępnego w manualu Jeneticsa. Pseudokod tego rozwiązania wygląda następująco:

```
ustal P(t); //wygeneruj punkty wielokąta foremnego
dopóki nie liczba pokolen <= zadana liczba pokolen
{
    mutuj P(t) z określonym prawdopodobieństwem;
    reprodukuje P(t) z określonym prawdopodobieństwem;
    inkrementuj liczbę pokoleń
}
Oceń i wypisz podsumowanie
```

Możliwe wartości parametrów będą ustalone w następnym podpunkcie dla poszczególnych symulacji. Wartościami są:

- liczba miast (punktów do odwiedzenia)
- parametry matematyczne do opisanie wielokąta foremnego (punkty są losowane jako wielokąt foremny na określonym okręgu), stąd program wie jaka jest najkrótsza możliwa trasa (bo policzenie takiej jakimś algorytmem byłoby zbyt kosztowne w czasie dla większej ilości miast)
- maksymalny wiek fenotypu
- rozmiar badanej populacji
- prawdopodobieństwo mutacji
- szansa na reprodukcję, ale bez możliwości powielenia wielokrotnego genu (PartiallyMatchedCrossover gwarantuje nam, że żaden punkt nie zostanie powielony w trakcie łączenie materiału genetycznego dwóch osobników, w problemie komiwojażera miasto do odwiedzenia nie może być zduplikowane, nie powstanie dzięki temu niepoprawne rozwiązanie)
- liczba generacji (stabilnych)

Przystosowanie (fitness) w przypadku problemu komiwojażera jest powiązany ściśle z długością trasy, jaka tworzy osobnik. Jest odwrotnie proporcjonalny do długości osobnika.

## 4. Wyniki eksperymentów

### 4.1 Pierwsza symulacja z parametrami:

Parametry i symulacji dla 5 uruchomień:

- 40 miast
- promień okręgu opisanego na wielokącie foremnym = 20
- maksymalny wiek fenotypu = 500
- wielkość populacji = 1000
- prawdopodobieństwo mutacji 20%
- szansa na reprodukcję 35%
- 10 000 generacji

Przykładowy wynik symulacji 1:

```
+-----+
| Evolution statistics |
+-----+
|      Generations: 10 000      |
|      Altered: sum=52 228 896; mean=5222,8896000000 |
|      Killed: sum=0; mean=0,0000000000 |
|      Invalids: sum=0; mean=0,0000000000 |
+-----+
| Population statistics |
+-----+
|      Age: max=43; mean=1,877134; var=8,492054 |
|      Fitness: |
|      min  = 234,767789902427 |
|      max  = 1364,660616092629 |
|      mean = 422,584178632771 |
|      var  = 88402,518390044860 |
|      std  = 297,325610047377 |
+-----+
Known min path length: 125.53455316455191
Found min path length: 234.76778990242653
```

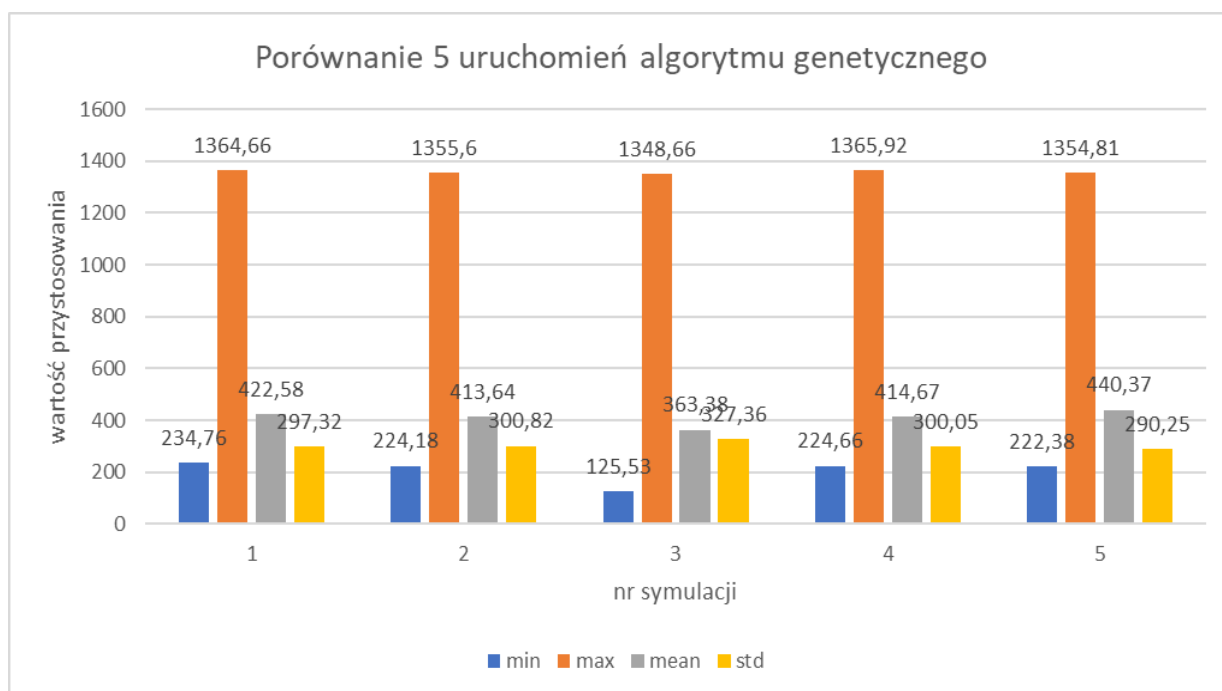
Przykładowy wynik symulacji 2:

```
+-----+
| Population statistics |
+-----+
|      Age: max=40; mean=1,870422; var=8,400962 |
|      Fitness: |
|      min  = 125,534553164552 |
|      max  = 1382,822907509496 |
|      mean = 357,854047113747 |
|      var  = 107944,004069247750 |
|      std  = 328,548328361670 |
+-----+
Known min path length: 125.53455316455191
Found min path length: 125.53455316455191
```

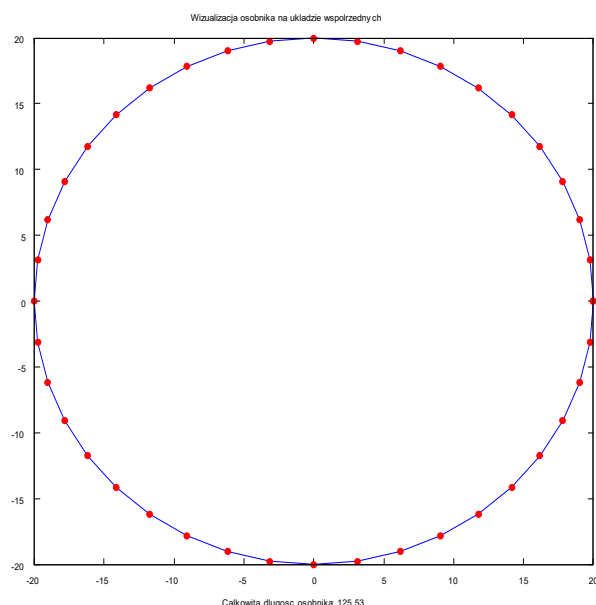


W celu ułatwienia prezentacji wyników zebraliśmy je do poniższej tabeli:

nr symulacji	min	max	mean	std
1	234,76	1364,66	422,58	297,32
2	224,18	1355,6	413,64	300,82
3	125,53	1348,66	363,38	327,36
4	224,66	1365,92	414,67	300,05
5	222,38	1354,81	440,37	290,25



Spoglądając na wykres, możemy zauważyć, że ewolucja za każdym razem doprowadziła do powstania innego osobnika. Najlepiej wypadł osobnik z 3 symulacji działania algorytmu genetycznego. Okazał się on najkrótszym możliwym osobnikiem do uzyskania. Wiemy to, ponieważ najkrótszym cyklem jest cykl opisujący wielokąt foremny:



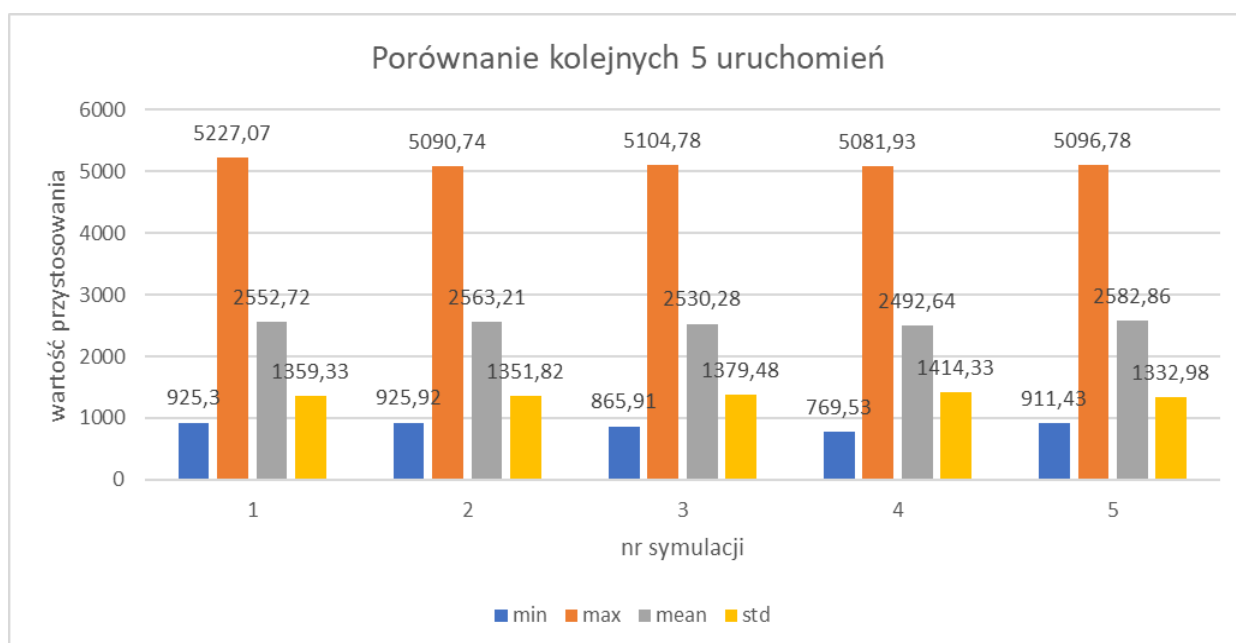
## 4.2 Druga symulacja z parametrami:

Parametry II symulacji dla 5 uruchomień:

- 80 miast
- promień okręgu opisanego na wielokącie foremnym = 40
- maksymalny wiek fenotypu = 300
- wielkość populacji = 2000
- prawdopodobieństwo mutacji 50%
- szansa na reprodukcję 5%
- 5 000 generacji

Wyniki symulacji zostały wpisane do poniższej tabeli:

nr symulacji	min	max	mean	std
1	925,3	5227,07	2552,72	1359,33
2	925,92	5090,74	2563,21	1351,82
3	865,91	5104,78	2530,28	1379,48
4	769,53	5081,93	2492,64	1414,33
5	911,43	5096,78	2582,86	1332,98



Jak widzimy, osobnik o najkrótszym możliwym cyklu Hamiltona wystąpił w 4 wywołaniu algorytmu ewolucyjnego Jeneticsa. Dla powyższych parametrów uzyskiwane najlepsze osobniki miały podobną do siebie długość. Możemy również powiedzieć, że wartości przystosowania minimalnego, uśrednionego i maksymalnego nie są zależne od siebie. Dana symulacja może mieć najgorszy uśredniony fitness, ale jednocześnie może ona stworzyć najlepiej przystosowanego osobnika ze wszystkich symulacji. Spoglądając na wyniki możemy również stwierdzić, że ewolucja dla 5 uruchomień programu nie zdołała utworzyć najkrótszego możliwego osobnika (ponieważ wiemy że najkrótszym osobnikiem jest ten opisany na wielokącie foremnym) o długości równej 251,26.

## 5. Podsumowanie

Podsumowując wyniki przedstawione w podpunkcie 4, widzimy że ewolucja każdorazowo generuje inne wyniki. Może ona skierować się w różne strony, niekoniecznie z pozytywnym rezultatem. Co więcej, to że dana ewolucja np. na początku generuje bardzo słabo przystosowane osobniki wcale nie oznacza, że ostateczny jej wynik będzie mniej korzystny od pozostałych. Dzięki możliwościom biblioteki Jenetics byliśmy w stanie zasymulować rozwiązanie problemu komiwojażera dla różnych zestawów parametrów. Dzięki wykorzystaniu faktu, że naszą najkrótszą trasą była trasa opisana jako wielokąt foremny o znanej liczbie punktów i promieniu koła opisanego na tym wielokącie, mogliśmy każdorazowo porównać wynik ewolucji z tym najlepszym możliwym do uzyskania. Byłoby to niemożliwe, gdyby miasta do odwiedzenia nie stanowiły wielokąta foremnego. W celu ulepszenia (przyspieszenia albo nakierunkowania) ewolucji można by się zastanowić nad tym, czy moglibyśmy wspomóc naszą ewolucję działaniami algorytmów wspomagających odnalezienie optymalnego cyklu Hamiltona, np. algorytmu najbliższego sąsiada albo algorytmu najmniejszej krawędzi. Być może przyspieszyło by to działanie algorytmu ewolucyjnego.

## 6. Bibliografia

1. [https://pl.wikipedia.org/wiki/Algorytm\\_ewolucyjny](https://pl.wikipedia.org/wiki/Algorytm_ewolucyjny)
2. [http://zeszyty-naukowe.wysi.edu.pl/zeszyty/zeszyt1/Algorytmy\\_Ewolucyjne\\_I\\_Ich\\_Zastosowania.pdf](http://zeszyty-naukowe.wysi.edu.pl/zeszyty/zeszyt1/Algorytmy_Ewolucyjne_I_Ich_Zastosowania.pdf)
3. <https://jenetics.io/manual/manual-7.0.0.pdf>
4. [GitHub - DEAP/deap: Distributed Evolutionary Algorithms in Python](https://github.com/DEAP/deap)
5. [https://en.wikipedia.org/wiki/DEAP\\_\(software\)](https://en.wikipedia.org/wiki/DEAP_(software))
6. Książka „Matematyka Dyskretna” – Kenneth A. Ross, Charles R.B. Wright
7. [https://pl.wikipedia.org/wiki/Problem\\_komiwojażera](https://pl.wikipedia.org/wiki/Problem_komiwojażera)
8. <https://stackoverflow.com/questions/7198144/how-to-draw-a-n-sided-regular-polygon-in-cartesian-coordinates>
9. [http://algorytmy.ency.pl/tutorial/problem\\_komiwojazera\\_algorytm\\_genetyczny](http://algorytmy.ency.pl/tutorial/problem_komiwojazera_algorytm_genetyczny)
10. [http://algorytmy.ency.pl/artukul/problem\\_komiwojazera](http://algorytmy.ency.pl/artukul/problem_komiwojazera)
11. + materiały z wykładu