Sztuczna Inteligencja i Inżynieria Wiedzy

Sprawozdanie numer 1

Autor: Patryk Konopka, 237980

Data: 31.03.2019

1. Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z algorytmami genetycznymi. Naszym zadaniem było napisanie algorytmu genetycznego rozwiązującego problem TSP(Problem Komiwojażera).

2. Algorytm genetyczny

Mój algorytm składa się z następujących klas:

- City – obiekty tej klasy są reprezentacją miast
  - Atrybuty:
    - x – położenie x
    - y – położenie y
  - Metody:
    - calculateDistance(self, city) – obliczanie odległości pomiędzy miastem wywołującym metodę, a miastem podanym jako parametr **city**.
- Genotype – obiekty tej klasy są reprezentacją drogi pomiędzy poszczególnymi miastami
  - Atrybuty:
    - route – lista indeksów poszczególnych miast
    - rank – ocena danego genotypu (1/ (odległość pomiędzy wszystkimi miastami)
  - Metody:
    - createRoute(self, cityList) – tworzy losową drogę pomiędzy wszystkimi miastami
    - rankRoute(self, cityList) – oblicza ocenę (rank) danej drogi (genotype)
- Population – obiekty tej klasy są repezentacją populacji
  - Atrybuty:
    - population – tablica obiektów typu Genotype
  - Metody:
    - generatePopulation(popSize, cityList) – tworzy populacje o rozmiarze **popSize** na podstawie **cityList**
- Main – plik wywoławczy
  - Metody:
    - readCities – odczytuje miasta z podanego pliku tekstowego
    - selectGenotypes(population, popSize, tour) – metoda turniejowa. Wybieramy losowo **tour** osobników z **population**. Następnie wybieramy z nich osobnika z najlepszą oceną. Całą procedurę powtarzamy **popSize***2 (potrzeba dwóch osobników do krzyżowania).

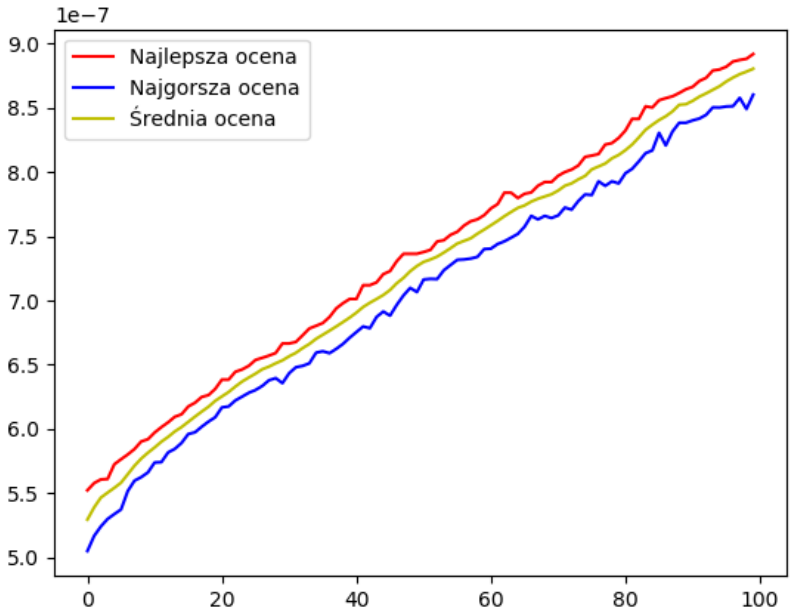
- breed(parent1, parent2) – metoda krzyżująca. Najpierw losujemy dwa geny (miasta) z **parent1**. Następnie wszystkie geny w pomiędzy <genA, genB> zapisujemy do dziecka. Pozostałe geny wybieramy po kolei z **parent2** (o ile już nie są użyte w dziecku).
- mutate(genotype) – metoda mutująca. Losujemy dwa geny(miasta) z **genotype** i zamieniamy je miejscami.
- generatePopulation(population, popSize, cityList, tour, px, pm) – metoda generująca populację o rozmiarze **popSize** z poprzedniej **population**. Krzyżowanie następuję w **px***100% przypadków. Jeżeli krzyżowanie nie następuję, to pierwszy z rodziców zostaje zapisany do następnej populacji. Mutacja następuję w **pm***100% przypadków.

3. Działanie algorytmu

Na początku wywołałem algorytm zgodnie z zalecanymi parametrami. Wyniki można zaobserwować w tabeli poniżej oraz na wykresie. Wyniki (rank) we wszystkich tabelach podane są w postaci 10e-7.

Lp.	popSize	genSize	px	pm	tour	maxRank	minRank	avgRank
1	100	100	0.7	0.01	5	8.633087	5.093437	7.024110
2	100	100	0.7	0.01	5	8.287764	5.078750	6.969033
3	100	100	0.7	0.01	5	8.764999	5.075080	7.223553
4	100	100	0.7	0.01	5	8.550358	5.080104	7.05623
5	100	100	0.7	0.01	5	8.919634	5.048965	7.200335
Odchylenie standardowe						0,225584	0,001058	0,050017

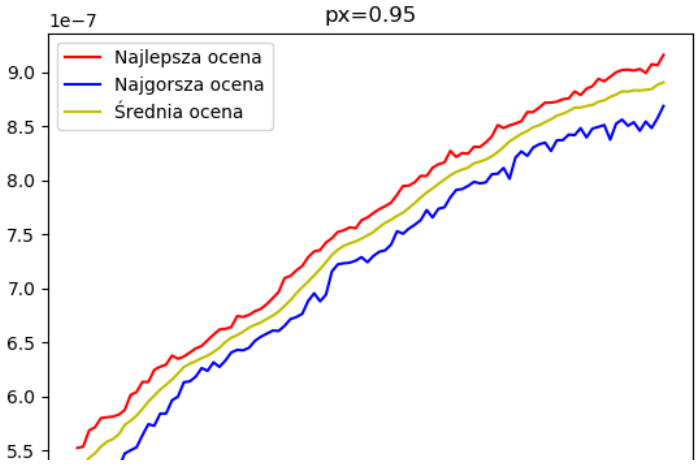
Jak widać w powyższej tabeli najlepsze wyniki oscylują w okolicach 8-9, najgorsze zaś około wartości 5. Z poniższego wykresu możemy zauważyć, że populacja cały czas się rozwija, co sugeruje nam, że należy zwiększyć **genSize.**

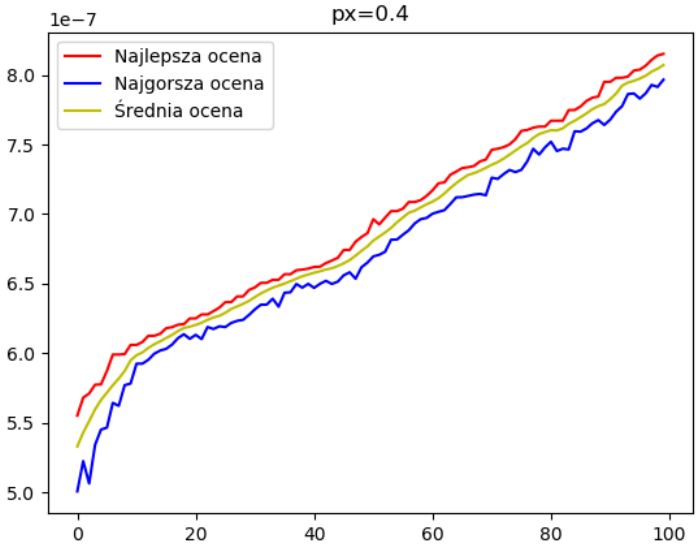


Następnie chciałbym sprawdzić wpływ prawdopodobieństwa krzyżowania **px** na wyniki symulacji. Wykonam 3 symulacje dla wartości większej (0.95) oraz 3 dla wartości mniejszej (0.4).

Lp.	popSize	genSize	px	pm	tour	maxRank	minRank	avgRank
1	100	100	0.95	0.01	5	9.245155	5.011951	7.372010
2	100	100	0.95	0.01	5	8.893194	5.078711	7.205392
3	100	100	0.95	0.01	5	9.161512	5.114485	7.395631
Odchylenie standardowe						0,067622	0,005417	0,021503
4	100	100	0.4	0.01	5	7.892592	5.039352	6.731388
5	100	100	0.4	0.01	5	7.831613	5.119110	6.759666
6	100	100	0.4	0.01	5	8.152229	5.005849	6.848572
Odchylenie standardowe						0,057975	0,006771	0,007479

Jak można zauważyć w powyższej tabeli im wyższe prawdopodobieństwo krzyżowania tym szybsza ewolucja populacji.

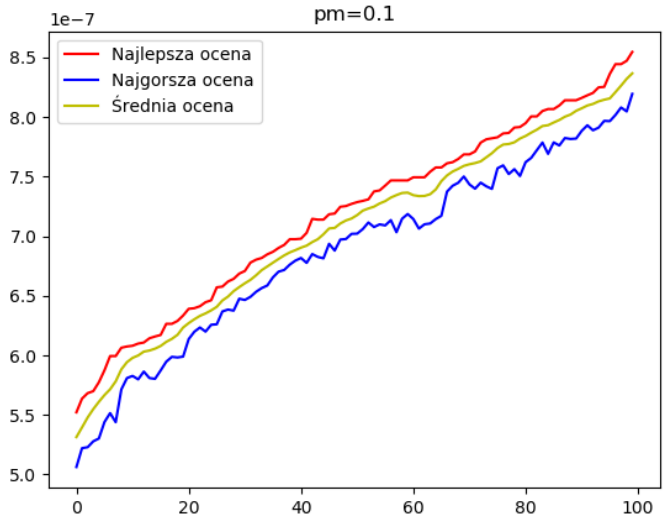


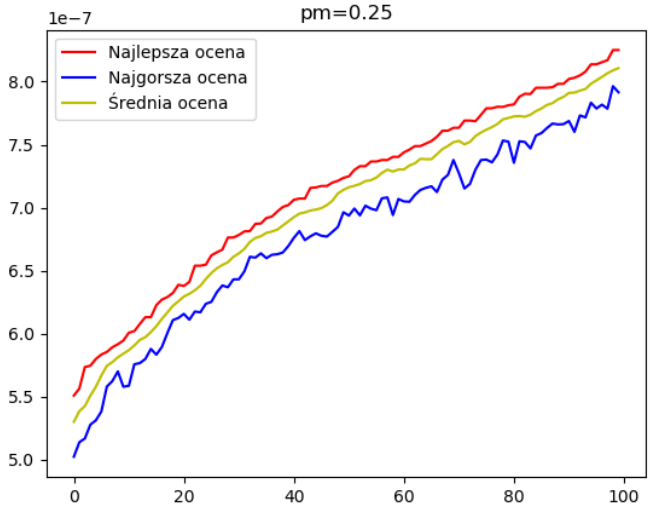


Kolejnym etapem jest badanie wpływu współczynnika mutacji **pm** na wpływ symulacji. Analogicznie symulację przeprowadzę dla współczynnika mutacji o wartości 0.1 oraz 0.25.

Lp.	popSize	genSize	px	pm	tour	maxRank	minRank	avgRank
1	100	100	0.7	0.1	5	8.732356	5.050204	7.169398
2	100	100	0.7	0.1	5	8.787634	5.116286	7.173765
3	100	100	0.7	0.1	5	8.547654	5.062660	7.048699
Odchylenie standardowe						0,031587	0,002466	0,010076
4	100	100	0.7	0.25	5	8.335631	5.074119	7.092838
5	100	100	0.7	0.25	5	8.273173	5.117636	7.065029
6	100	100	0.7	0.25	5	8.250771	5.021652	6.991781
Odchylenie standardowe						0,003868	0,00462	0,00545

Jak można wywnioskować z powyższej tabeli zmiana parametru pm nieznacznie zmieniła wynik działania algorytmu. Warto jednak zauważyć, że im częstsza mutacja tym wynik jest nieznacznie gorszy.

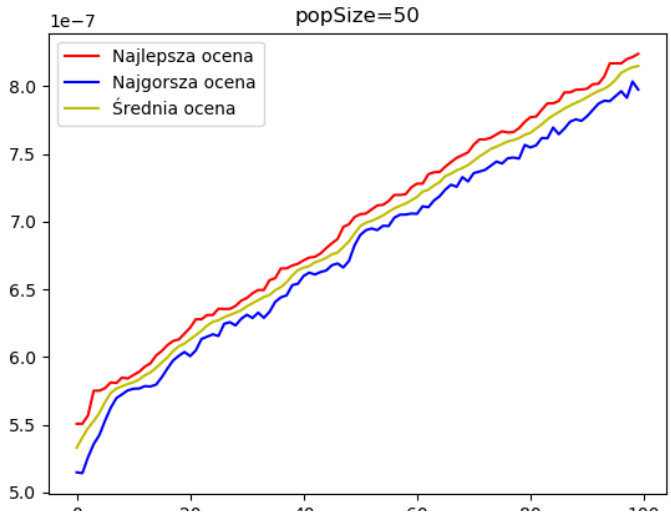


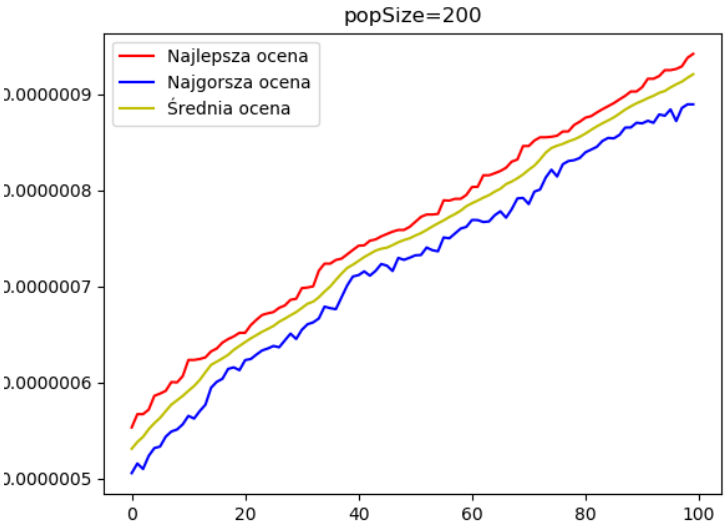


Następnym krokiem jest sprawdzenie wpływu **popSize** na działanie GA. W poniższych przykład przetestuje warianty dla populacji 50 oraz 200.

Lp.	popSize	genSize	px	pm	tour	maxRank	minRank	avgRank
1	50	100	0.7	0.01	5	7.92845	4.968267	6.789451
2	50	100	0.7	0.01	5	8.294366	5.096814	6.998621
3	50	100	0.7	0.01	5	8.240268	5.14322	6.88192
Odchylenie standardowe						0,078017	0,016429	0,021974
4	200	100	0.7	0.01	5	9.492778	5.001047	7.548889
5	200	100	0.7	0.01	5	9.167524	5.071510	7.317510
6	200	100	0.7	0.01	5	9.419721	5.056571	7.463235
Odchylenie standardowe						0,058244	0,002757	0,02737

Łatwo zauważyć, że zmniejszenie rozmiaru znacznie pogorszyło wynik (pierwsza ocena poniżej 5). Zwiększenie zaś rozmiaru populacji znacznie poprawiło wyniki (ocena powyżej 9). Jest to rezultat dwukrotnie większej liczby osobników to skrzyżowania.

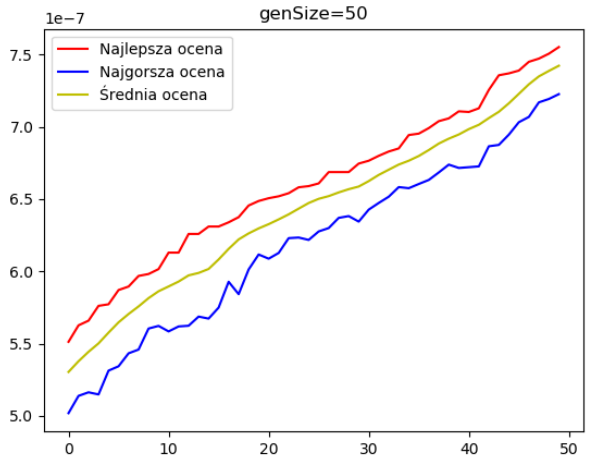


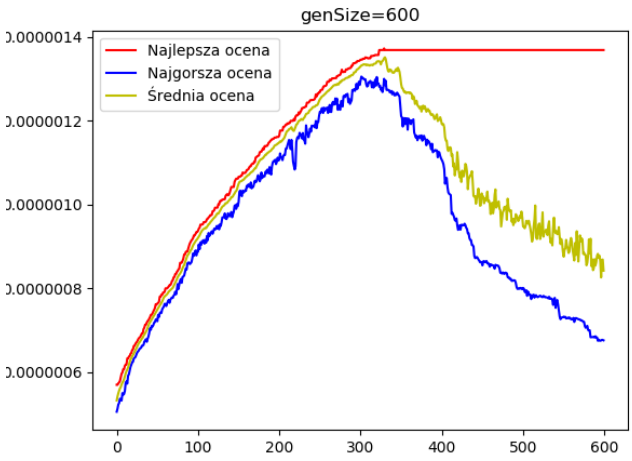


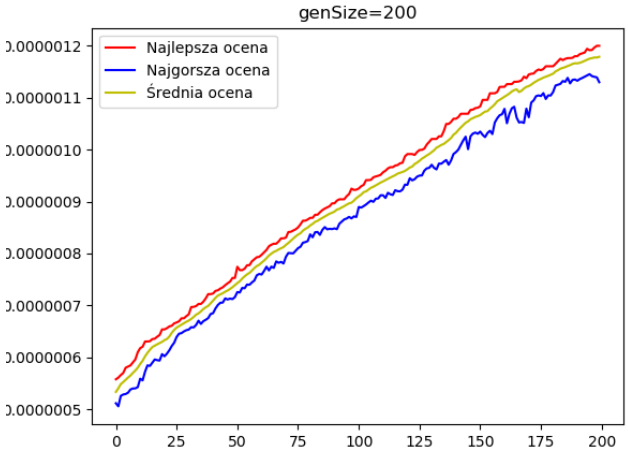
Teraz czas zbadać wpływ zmiany **genSize** na działanie GA. Analogicznie zmierzę wyniki dla parametrów 50 i 200.

Lp.	popSize	genSize	px	pm	tour	maxRank	minRank	avgRank
1	100	50	0.7	0.01	5	7.606618	5.076205	6.498289
2	100	50	0.7	0.01	5	7.614318	5.010699	6.529396
3	100	50	0.7	0.01	5	7.551896	5.018478	6.428163
Odchylenie standardowe						0,002317	0,002561	0,005378
4	100	200	0.7	0.01	5	12.484029	4.931595	9.073907
5	100	200	0.7	0.01	5	12.074425	5.045463	9.140396
6	100	200	0.7	0.01	5	11.995607	5.060490	8.980696
Odchylenie standardowe						0,137515	0,009935	0,012871
7	100	600	0.7	0.01	5	13.72700	5.050897	10.424491

Tutaj widzimy wyraźną zależność pomiędzy parametrem **genSize**, a wynikami GA. Im więcej generacji tym lepsze rezultaty osiągamy. Jak zauważamy na następnych wykresach, wyniki ciągle nie są ustabilizowane. Dopiero dla wartości **genSize** = 600 wyniki się ustabilizowały (od ok. 350 generacji). Dobrze to widać na poniższych wykresach.







Przeanalizowałem swój kod za pomocą narzędzia profiler. Zgodnie z moimi przypuszczeniami najbardziej czasochłonną metodą było obliczanie dystansu pomiędzy miastami. Poniżej tabela z metodami, których wywołanie zajęło najwięcej czasu.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
6286280	18.010	0.000	19.097	0.000	City.py:11(calculateDistance)
14087	17.777	0.001	17.777	0.001	Main.py:65(<listcomp>)
14287	6.863	0.000	25.963	0.002	Genotype.py:9(rankRoute)
12572560	1.087	0.000	1.087	0.000	{built-in method builtins.abs}

4. Wnioski

Realizacja tego ćwiczenia znacznie poszerzyła moją wiedzę. Dowiedziałem się o istnieniu algorytmu genetycznego, jak i ze pozytywnym skutkiem udało mi się go zaimplementować. Zdaję sobie sprawę, że mój algorytm nie jest idealny. Na pewno jest w nim dużo rzeczy, które można by zoptymalizować. Jednakże z braku czasu i natłoku zadań wymaganych na innych kursach aktualnie nie jest to możliwe.

W mojej realizacji największy wpływ miały parametry popSize i genSize, czyli po prostu długość trwania algorytmu. Im dłużej on trwał, tym lepsze wyniki zwracał. Odbywało się to jednak wspomnianym już kosztem czasu (czas wykonania jednej 600 generacji po 100 osobników każda wynosił już niecałe 2 minuty). Inne parametry oczywiście też miały wpływ na przebieg algorytmu, jednak nie był on tak duży jak wpływ wcześniej wymienionych wartości.

Zmiana sposobu selekcji osobników (wyznaczenie elit, zmiana zasad krzyżowania itp.) zapewne również by wpłynęły na działanie GA, jednak czas nie pozwolił mi tego zbadać.

Podsumowując całe zadanie było ciekawym zagadnieniem. Znacząco pomogło mi w rozwoju moje wiedzy z zakresu programowania w Pythonie oraz problemu komiwojażera.