Sztuczna inteligencja i inżynieria wiedzy laboratorium

Sprawozdanie 1

Algorytmy genetyczne

Arkadiusz Marcinowski

228160

W8, Informatyka

1. Wstęp

Celem zadanie było zapoznanie się z metaheurystyką algorytmów generycznych przez samodzielną implementację. Algorytm generyczny jest metaheurystyką, która naśladuje ewolucję naturalną metodą ciśnienia selekcyjnego i doboru naturalnego. Aby ją zastosować, zdefiniowano potencjalne rozwiązanie (osobnika), sposób jego zmiany (mutacja), łączenia (krzyżowania) oraz oceny jakości rozwiązania (funkcja oceny).

Na wygenerowanej populacji osobników posiadających własne genotypy, należało użyć algorytmu przeprowadzającego populację przez wiele generacji. Pierwszym sposobem był algorytm generyczny, następnie generacja losowa i algorytm zachłanny. W pracy przedstawiono porównanie skuteczności algorytmu generycznego z dwoma innymi metodami nieewolucyjnymi oraz wpływ parametrów na algorytmy.

Do badań wykorzystano 5 instancji ze strony
http://anjos.mgi.polymtl.ca/qaplib/inst.html#HRW

2. Podejście do problemu

By zastosować metaheurystykę, zdefiniowano następujące elementy:

- **Osobnik** – obiekt przechowujący kolekcję swojego genotypu
- **Genotyp** – zestaw genów zawierający liczby od 1 do długości genotypu, niepowtarzające się
- **Selekcja**:
    - **Turniejowa** – sposób selekcji, w którym z całej populacji wybieranych jest wstępnie **T** losowych osobników, a następnie, spośród wstępnie wybranych, wybrany zostaje osobnik o najlepszej wartości funkcji oceny (najmniejszej)
    - **Ruletka** – sposób selekcji, gdzie każdy osobnik ma szansę wyboru zależną od odwrotności wartości jego funkcji oceny (f). Szanse osobnika to: 10000/f, gdyż im mniejsza wartość funkcji, tym osobnik jest lepiej przystosowany.
    - **Losowa** – wybór osobników odbywa się w sposób losowy, z całej populacji
- **Krzyżowanie** – zostało zaimplementowane w taki sposób, by punkt podziału genotypu był losowy, przy czym zawsze oddziela się co najmniej jeden gen. Zwrócono również uwagę na możliwość uszkodzenia genotypu przy procesie krzyżowania
- **Naprawa genotypu** – uszkodzony genotyp naprawiany jest przez podmianę powtarzających się genów wartościami które w genotypie nie wystąpiły, a które powinny wystąpić
- **Mutacja** – jeśli dla danego genu występuje mutacja, zostaje on zamieniony z innym losowym genem w obrębie tego genotypu
- **Funkcja oceny (funkcja jakości)** – funkcja zwracająca wartość przystosowania osobnika. Im wartość jest mniejsza, tym osobnik jest lepiej przystosowany.

Parametry programu:

POP_SIZE – ilość osobników w populacji
GEN – liczba generacji przez które przechodzi populacja
PM – prawdopodobieństwo mutacji określające czy dla danego genu zachodzi mutacja
PX - prawdopodobieństwo krzyżowania określające czy dla danego osobnika zachodzi krzyżowanie
TOUR – liczba osobników wybieranych wstępnie w selekcji turniejowej
filePath – ścieżka do pliku na którym prowadzone jest badanie
macierzOdleglosci – macierz odległości z pliku określonego w filePath
macierzPrzeplywu– macierz przepływu z pliku określonego w filePath

3. Zbudowanie algorytmu genetycznego

3.1. Implementacja

3.1.1. Osobnik

```
public class Person {
    private List <Integer> genotype;

    public Person(List genotype) {
        this.genotype = genotype;
    }
}
```

3.1.2. Funkcja oceny

```
public int checkValue(int [][] matrixDist, int [][] matrixFlow){
    int result = 0;
    for (int i = 0; i < genotype.size(); i++){
        for (int j = 0; j < genotype.size(); j++){
            result += ((matrixDist[i][j]) *
matrixFlow[genotype.get(i)-1][genotype.get(j)-1]);
        }
    }
    return result;
}
```

3.1.3. Krzyżowanie

```
public Population crossoverPopulation(Population population, double PX, int
POP_SIZE, int PERSON_SIZE){
    ArrayList <Person> pop = population.getPopulation();
    Population newPopulation = new Population(POP_SIZE, PERSON_SIZE);
    Random rn = new Random();
    while ( pop.size() > 1){
        int currentIndex = 0;
        int nextIndex = 1;
        if (Math.random() < PX){
            int cutPoint = rn.nextInt(PERSON_SIZE - 1);
            List<Integer> genotype1 = new ArrayList<Integer>();
            List<Integer> genotype2 = new ArrayList<Integer>();
            for (int gen = 0; gen <= cutPoint; gen++) {
                genotype1.add((Integer)
pop.get(currentIndex).getGenotype().get(gen));
                genotype2.add((Integer)
pop.get(nextIndex).getGenotype().get(gen));
            }
            int goCut = cutPoint+1;
            for (int gen = goCut; gen < PERSON_SIZE; gen++) {
```

```
                genotype1.add((Integer)
pop.get(nextIndex).getGenotype().get(gen));
                genotype2.add((Integer)
pop.get(currentIndex).getGenotype().get(gen));
            }
            Person person1 = new Person(genotype1);
            Person person2 = new Person(genotype2);
            person1.repairGenotypeIfWrong();
            person2.repairGenotypeIfWrong();
            newPopulation.addPerson(person1);
            newPopulation.addPerson(person2);
            pop.remove(0)
            pop.remove(0);
        }
        else{
            newPopulation.addPerson(pop.get(currentIndex));
            pop.remove(currentIndex);
        }
    }
    if(pop.size() == 1){
        newPopulation.addPerson(pop.get(0));
    }
    return newPopulation;
}
```

3.1.4. Naprawa genotypu

```
public void repairGenotypeIfWrong(){
    List<Integer> duplicates = new
ArrayList<Integer>(findDuplicates(genotype));
    List<Integer> potential = new
ArrayList<Integer>(findPotential(genotype));
    if(!duplicates.isEmpty()) {
        for (int i = 0; i < duplicates.size(); i++) {
            int indexOfDup = genotype.indexOf(duplicates.get(i));
            genotype.set(indexOfDup, potential.get(i));
        }
    }
}
```

3.1.5. Mutacja

```
public Population mutatatePopulation(Population population, double PM) {
    Random rn = new Random();
    int genotypeSize = population.getPersonSize();
    int populationSize = population.getPopulation().size();
    Population newPopulation = new Population(populationSize,
genotypeSize);
    ArrayList<Person> pop = population.getPopulation();

    for (int i = 0; i < populationSize; i++) {
        Person person = pop.get(i);
        for (int gen = 0; gen < genotypeSize; gen++) {
            if (Math.random() < PM) {
                int toSwapWith = rn.nextInt(genotypeSize);
                while (gen == toSwapWith) {
                    toSwapWith = rn.nextInt(genotypeSize);
                }
                person = mutatePerson(person, gen, toSwapWith);
            }
        }
        newPopulation.addPerson(person);
    }
```

```
    return newPopulation;
}

private Person mutatePerson(Person person, int genIndex, int toSwapWith){
    int tpm = person.getGenotype().get(genIndex);
    person.getGenotype().set(genIndex,
person.getGenotype().get(toSwapWith));
    person.getGenotype().set(toSwapWith, tpm);
    return person;
}
```

3.1.6. Selekcja

3.1.6.1. Turniejowa

```
public Person tourSelect(ArrayList <Person> pop, int TOUR, int[][]
distanceMatrix, int[][] flowsMatrix) {
    Random rn = new Random();
    ArrayList <Person> selectedTour = new ArrayList <Person>();
    for (int i = 0; i <TOUR; i++) {
        int randomRow = rn.nextInt(pop.size());
        selectedTour.add(pop.get(randomRow));
    }
    int bestVal = selectedTour.get(0).checkValue(distanceMatrix,
flowsMatrix);
    int indexOfBest = 0;
    for(int i = 1; i < selectedTour.size(); i++){
        int currentVal = selectedTour.get(i).checkValue(distanceMatrix,
flowsMatrix);
        if (currentVal < bestVal){
            bestVal = currentVal;
            indexOfBest = i;
        }
    }
    return selectedTour.get(indexOfBest);
}
```

3.1.6.2. Ruletka

```
public Person rouletteSelect(ArrayList <Person> pop, int[][]
distanceMatrix, int[][] flowsMatrix) {
    int popSize = pop.size();
    double totalSum = 0;
    for(int i = 0; i < popSize; i++){
        totalSum += (1000)/(pop.get(i).checkValue(distanceMatrix,
flowsMatrix));
    }

    Random rn = new Random();
    double randomChoice = rn.nextDouble() * totalSum;

    int actualId = 0;
    double sum = 0;
    while (actualId < popSize && totalSum < randomChoice) {
        sum += (1000)/(pop.get(actualId).checkValue(distanceMatrix,
flowsMatrix));
        actualId++;
    }
    return pop.get(actualId);
}
```

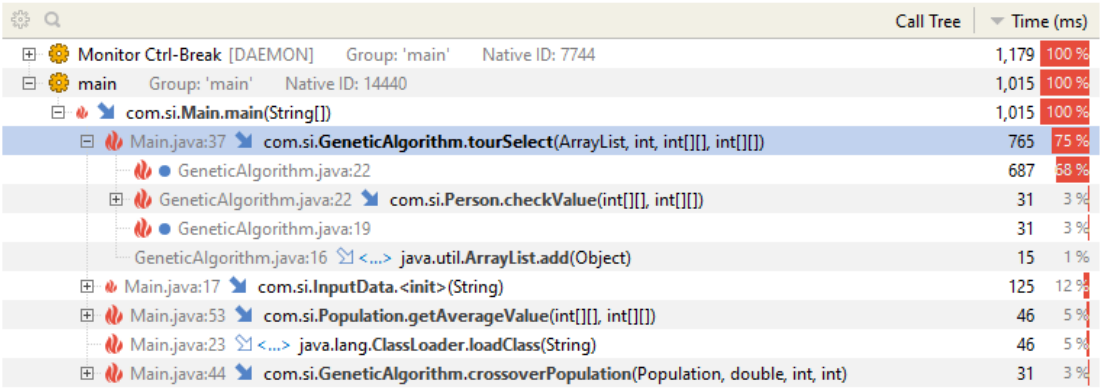
3.1.6.3. Losowa

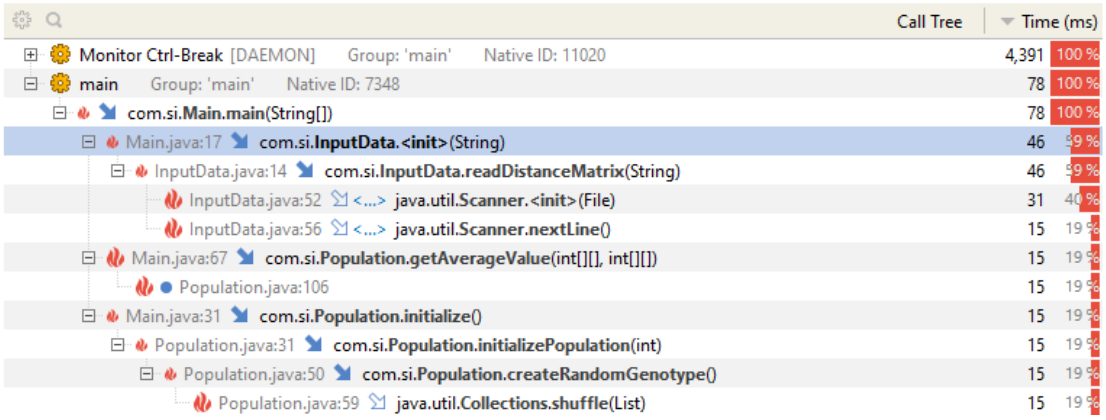
```
public Person randomSelect(ArrayList <Person> pop, int[][] distanceMatrix,
int[][] flowsMatrix) {
    int popSize = pop.size();
    Random rn = new Random();
    int randomIndex = rn.nextInt(popSize);
    return pop.get(randomIndex);
}
```

3.2 Test działania

Przeprowadzono test działania programu z użyciem oprogramowania *YourKit Java Profiler.*

Wyniki pokazały, że najbardziej kosztowną metodą jest odczyt wartości macierzy przepływu oraz macierzy odległości z pliku. Bardzo kosztowna jest również metoda selekcji turniejowej.





4. Badania

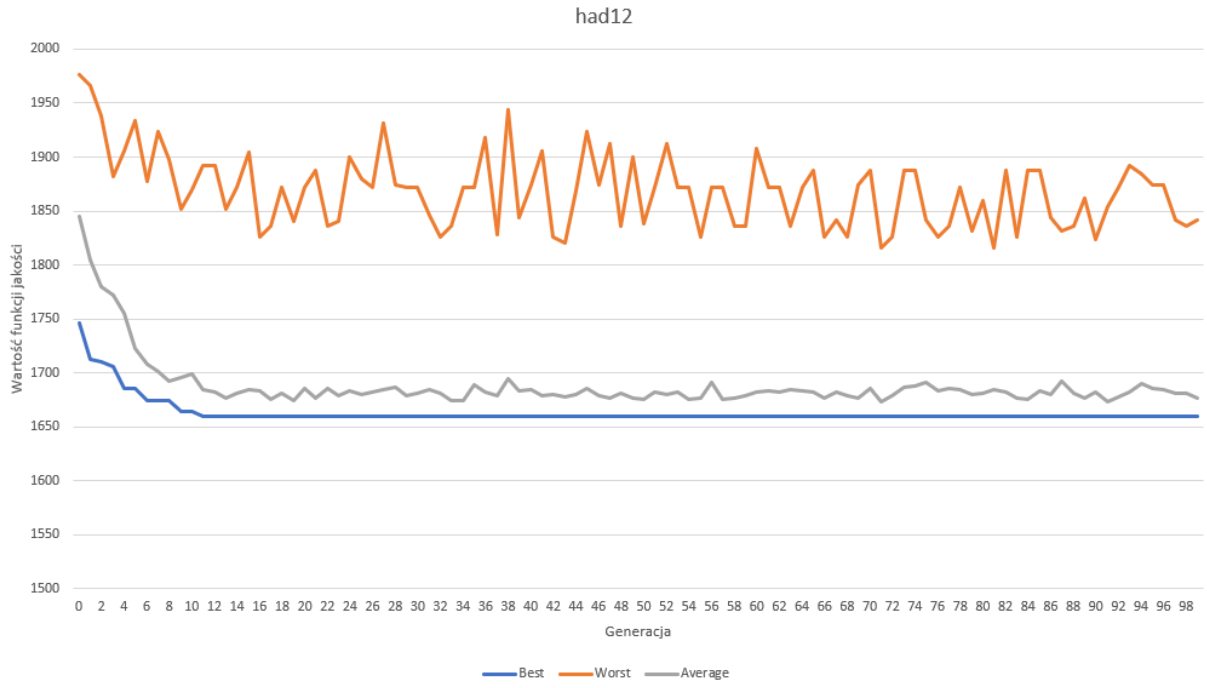
4.1 Zbadanie działania na 5 plikach testowych (pliki hadXX)

Algorytm generyczny został przetestowany dla wszystkich 5 plików tekstowych. Pobierając wartości funkcji oceny najlepszego osobnika, najgorszego osobnika oraz średnią wartość funkcji oceny dla populacji, wygenerowano wykresy pozwalające ocenić czy algorytm działa poprawnie. W tym badaniu ustalono także najlepsze wartości parametrów działania algorytmu. Na Rys.4.1.2-Rys.4.1.6 przedstawiono wykresy dla wszystkich plików. Rys.4.1.1 zawiera optymalne parametry na których operowano w tym punkcie.

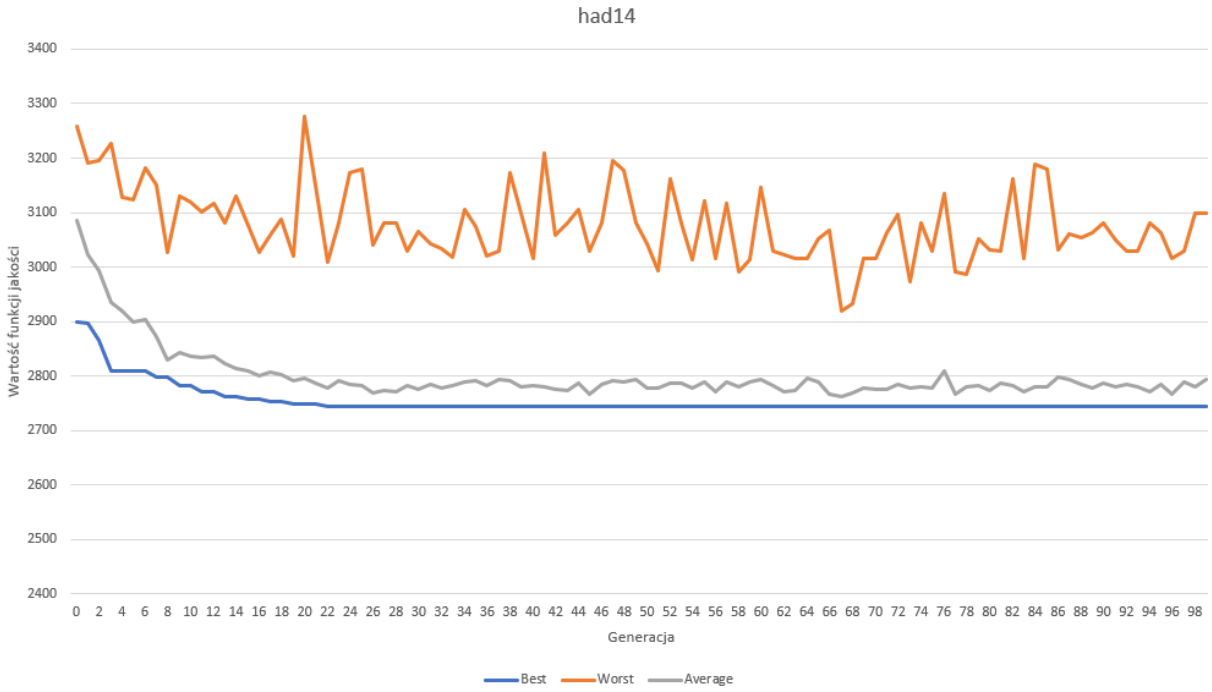
Na podstawie informacji z zajęć laboratoryjnych stwierdzono, że wykresy są zgodne z oczekiwaniami, a algorytm generyczny działa poprawnie dla wszystkich danych dostępnych do zadania.

```
final int POP_SIZE = 100;
final int GEN = 100;
final double PM = 0.03;
final double PX = 0.7;
final int TOUR = 8;
```

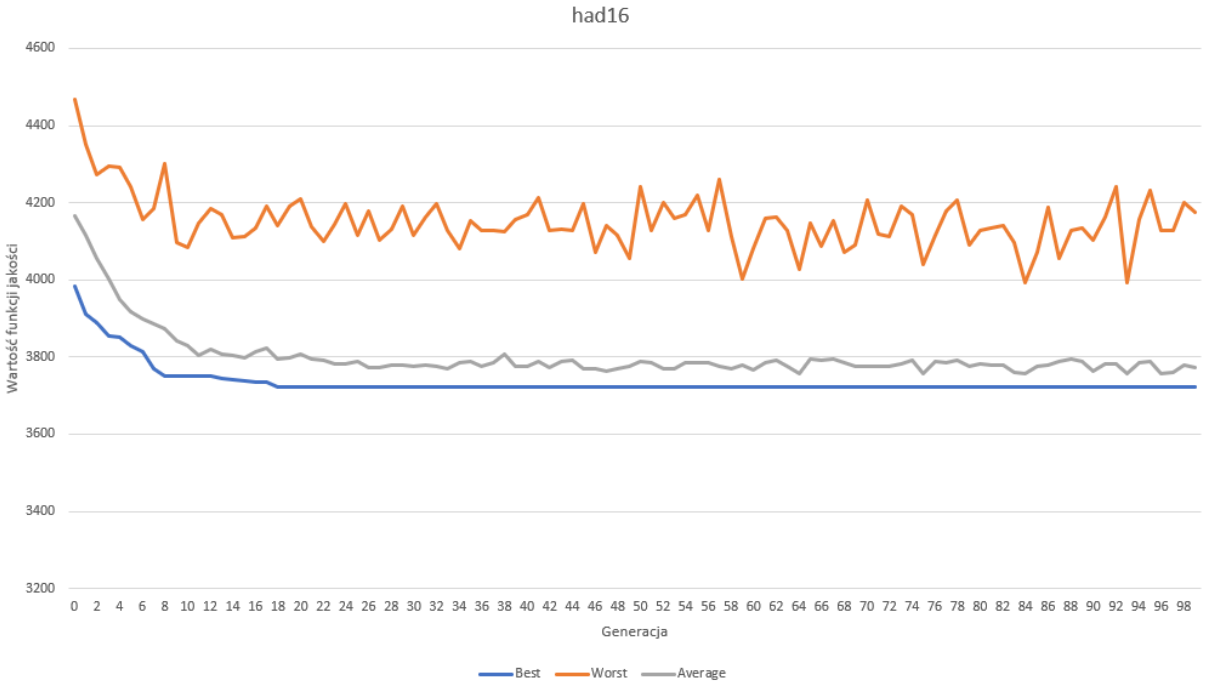
Rys.4.1.1 Parametry optymalne



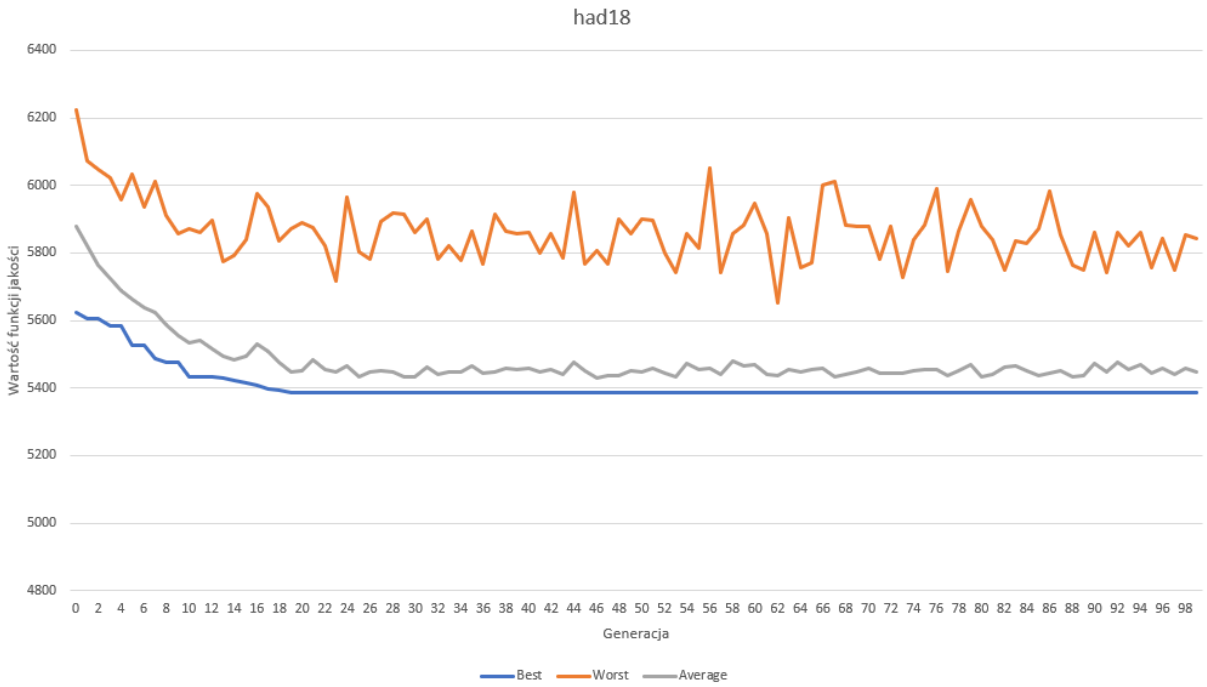
Rys.4.1.2 had12



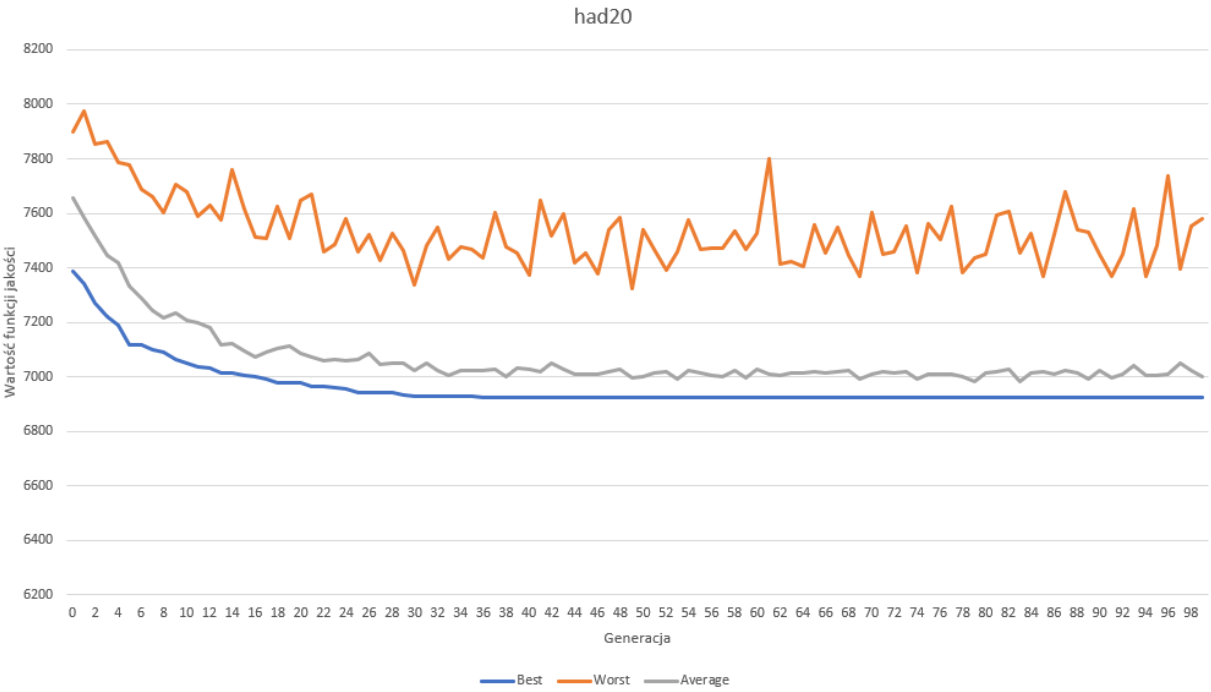
Rys.4.1.3 had14



Rys.4.1.4 had16



Rys.4.1.5 had18



Rys.4.1.6 had20

4.2 Zbadanie wpływu prawdopodobieństwa krzyżowania *PX* i mutacji *PM* na wyniki działania GA

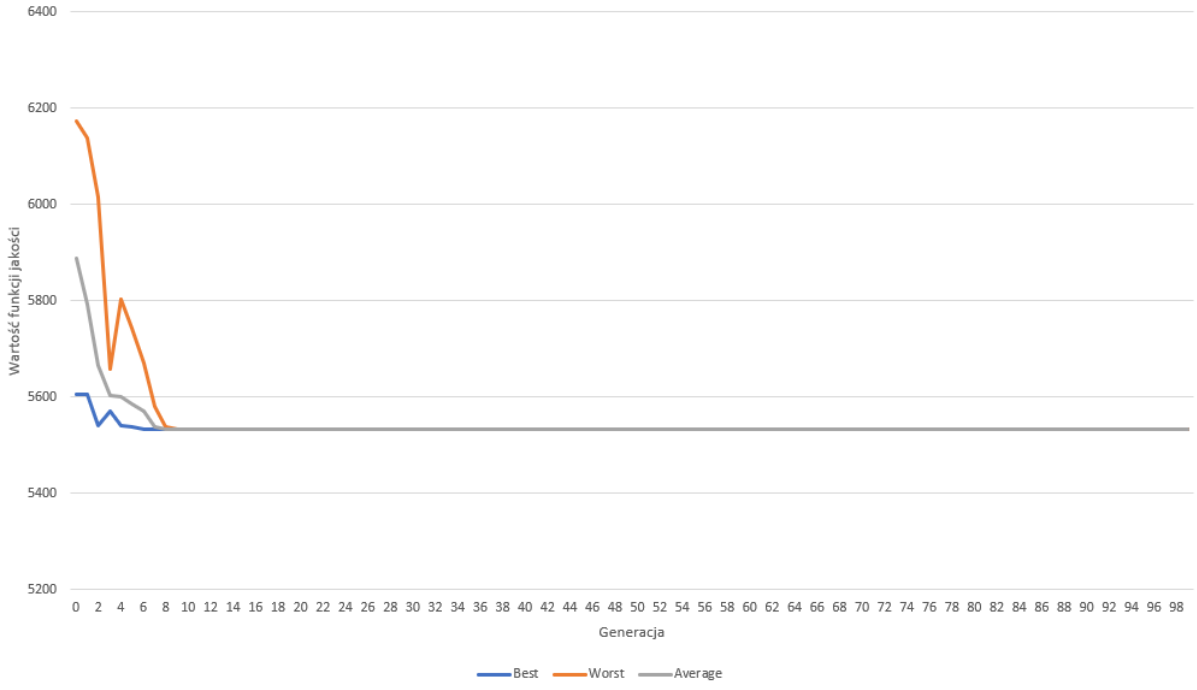
Badanie wpływy parametrów PX oraz PM na działanie algorytmu generycznego przeprowadzano na pliku *had18,* ponieważ stwierdzono brak zauważalnych różnic wpływu badanych parametrów na różne pliki.

Średni czas działania algorytmu z parametrami wyznaczonymi w *punkcie 4.1* to 271ms i do tego czasu należy odnosić dalsze wyniki. Na podstawie wyników czasowych przy każdym z przedstawionych badań zauważono, że prawdopodobieństwo krzyżowania ma większy wpływ na długość działania algorytmu niż prawdopodobieństwo mutacji. Wynika to z faktu złożoności operacji oraz tego że prawdopodobieństwo mutacji nie może przekraczać pewnego progu, opisanego w *punkcie 4.2.4.*

4.2.1 Brak mutacji

Średni czas wykonania programu dla 10 testów: 263ms

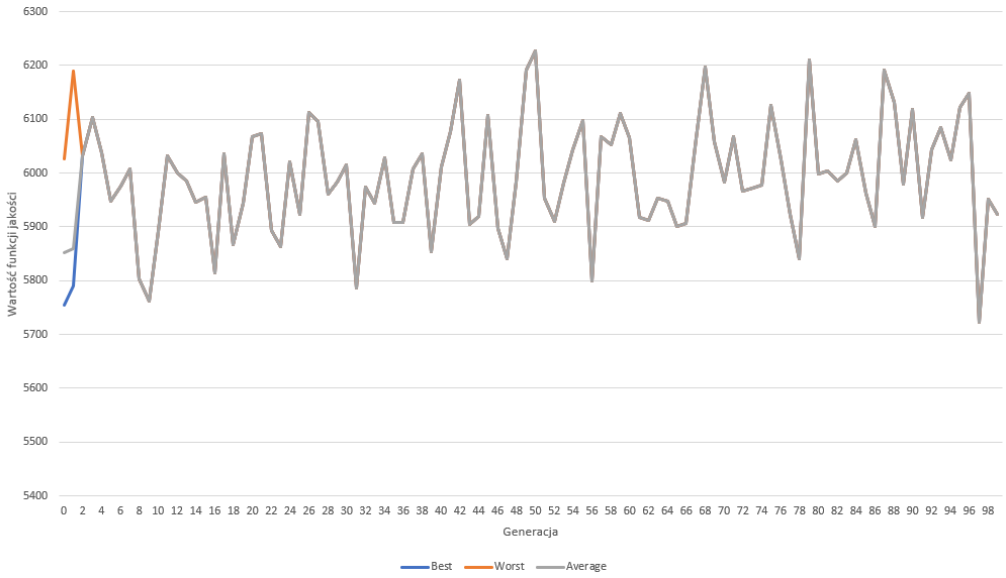
```
final int POP_SIZE = 100;
final int GEN = 100;
final double PM = 0;
final double PX = 0.7;
final int TOUR = 8;
String filePath = "input/had18.dat.txt";
```



4.2.2 Brak krzyżowania

Średni czas wykonania programu dla 10 testów: 193ms

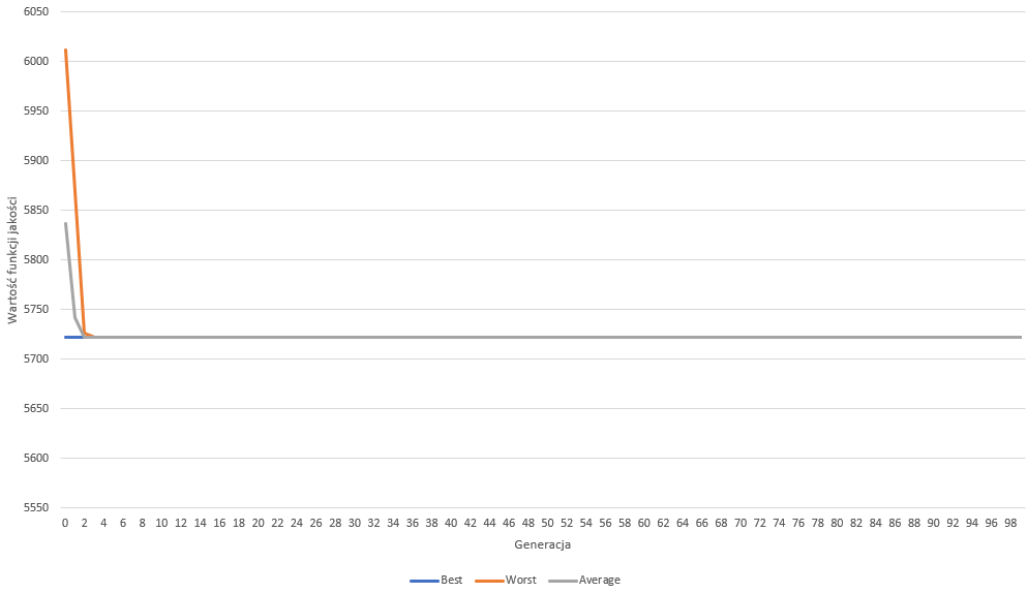
```
final int POP_SIZE = 100;
final int GEN = 100;
final double PM = 0.03;
final double PX = 0;
final int TOUR = 8;
String filePath = "input/had18.dat.txt";
```



4.2.3 Brak mutacji i krzyżowania

Średni czas wykonania programu dla 10 testów: 194ms

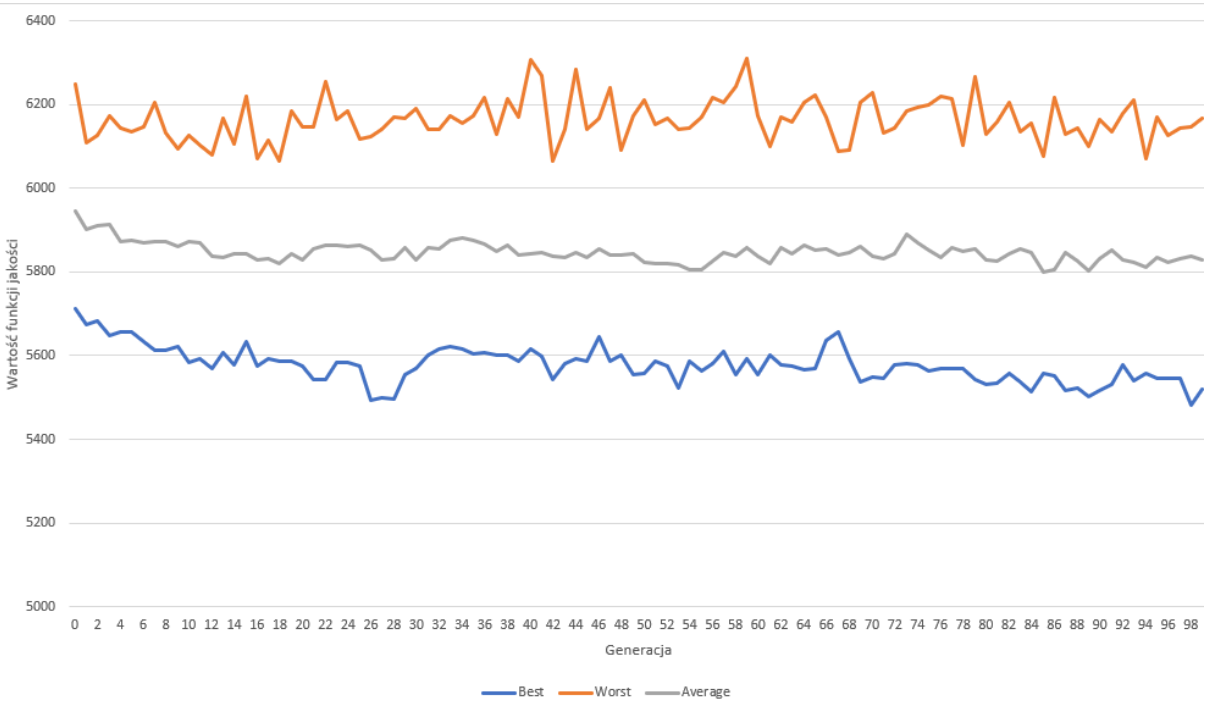
```
final int POP_SIZE = 100;
final int GEN = 100;
final double PM = 0;
final double PX = 0;
final int TOUR = 8;
String filePath = "input/had18.dat.txt";
```



4.2.4 Czy mutacji może być za dużo / za mało

Mutacji może być za dużo. Przy prawdopodobieństwie mutacji większym niż 10%. Na wykresie wartości funkcji jakości na przestrzeni pokoleń, można zauważyć ze algorytm generyczny traci swoją skuteczność. Dzieje się tak aż do wartości 100%, lecz to właśnie 10% jest w tym przypadku granicą parametru, której nie należy przekraczać.

```
final int POP_SIZE = 100;
final int GEN = 100;
final double PM = 0.15;
final double PX = 0.7;
final int TOUR = 8;
String filePath = "input/had18.dat.txt";
```



Jak widać w *punkcie 4.2.1,* zbyt mała liczba mutacji powoduje małą różnorodność populacji, przez co już po około 10 generacjach, wszystkie osobniki mają taki sam zestaw genów. Algorytm jest skuteczny, gdyż typuje najlepszego osobnika, lecz wytypowanie go tak szybo jest błędem parametru mutacji.

4.2.5 Czy krzyżowania może być za dużo / za mało

W zadaniu nie występuje problem ze zbyt dużą ilością krzyżowań osobników. Wydłuża to czas działania algorytmu, lecz nie zmienia wyników i skuteczności algorytmu.

W *punkcie 4.2.2* przedstawiono przypadek w którym nie występuje krzyżowanie. Zbyt mała liczba krzyżowań osobników zakłóca działanie algorytmu i niweluje jego skuteczność.

4.3 Zbadanie wpływu rozmiaru populacji *POP_SIZE* i liczby pokoleń *GEN* na wyniki działania GA

Badanie wpływy parametrów POP_SIZE oraz GEN na działanie algorytmu generycznego przeprowadzano na pliku *had18,* ponieważ stwierdzono brak zauważalnych różnic wpływu badanych parametrów na różne pliki.

Oczywistym jest, że dla parametrów wielkości 0, 1 algorytm generyczny nie jest skuteczny. Sprawdzono wartości mniejsze niż przyjęte wcześniej parametry testowe, ale też większe wartości, by zbadać ich wpływ na wynik i działanie algorytmu. Przy tym badaniu wykorzystano 'liczbę urodzeń', czyli iloczyn POP_SIZE oraz GEN.

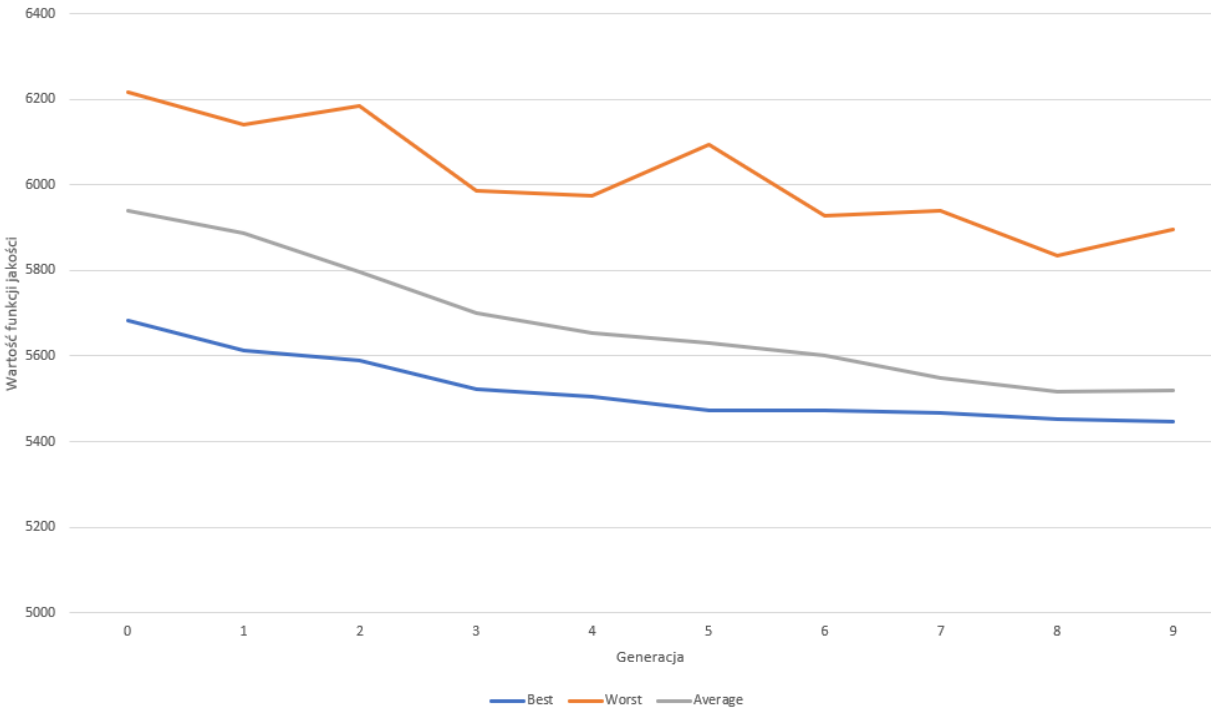
4.3.1 Mała liczba generacji

Przy zbyt małej liczbie generacji, algorytm ma zbyt mało czasu na znalezienie wyniku. Algorytm nie jest skuteczny. Czas działania programu jest krótki.

Średni czas wykonania programu dla 10 testów: 95ms

Liczba urodzeń: 1000

```
final int POP_SIZE = 100;
final int GEN = 10;
final double PM = 0.03;
final double PX = 0.7;
final int TOUR = 8;
String filePath = "input/had18.dat.txt";
```



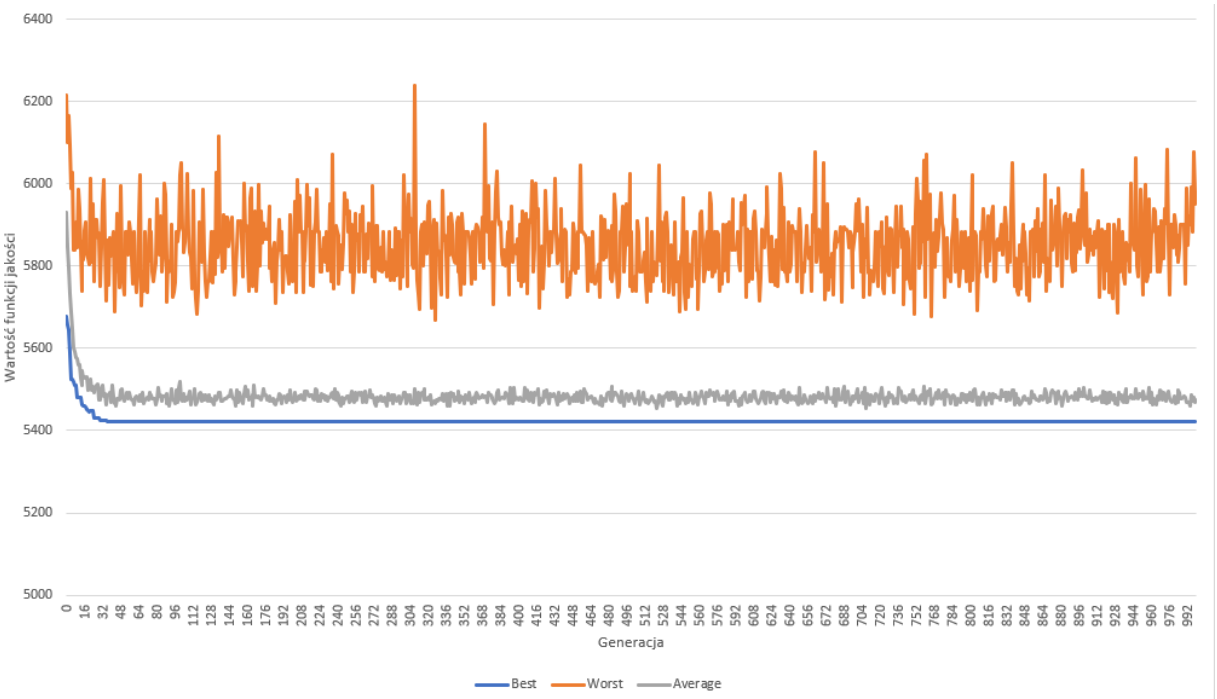
4.3.2 Duża liczba generacji

Przy zbyt dużej liczbie generacji i zachowanej wielkości populacji, algorytm wyznacza najlepszego osobnika, a jego tak długie działanie nie jest potrzebne. Za duża ilość generacji negatywnie wpływa na efektywność metody.

Średni czas wykonania programu dla 10 testów: 1062ms

Liczba urodzeń: 100000

```
final int POP_SIZE = 100;
final int GEN = 1000;
final double PM = 0.03;
final double PX = 0.7;
final int TOUR = 8;
String filePath = "input/had18.dat.txt";
```



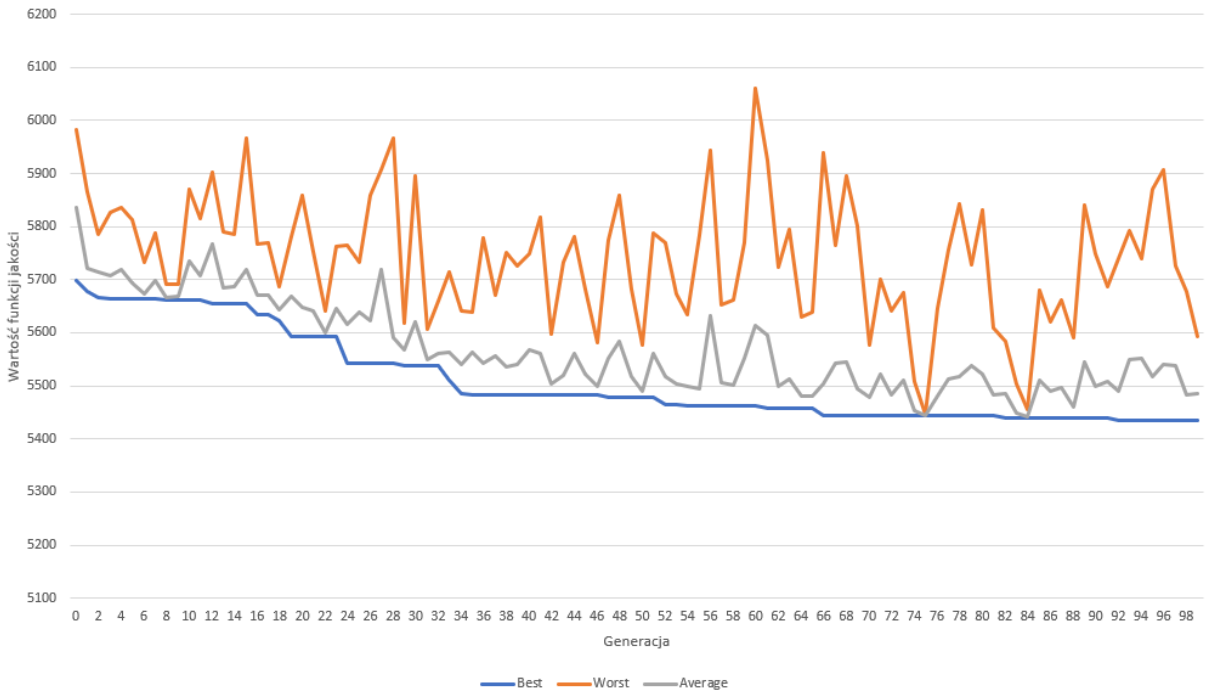
4.3.3 Mała populacja

Dla małej populacji algorytm zachowuje swoją skuteczność, jego efektywność jest wysoka z powodu małej ilości porównań i szybkiej selekcji.

Średni czas wykonania programu dla 10 testów: 93ms

Liczba urodzeń: 1000

```
final int POP_SIZE = 10;
final int GEN = 100;
final double PM = 0.03;
final double PX = 0.7;
final int TOUR = 8;
String filePath = "input/had18.dat.txt";
```



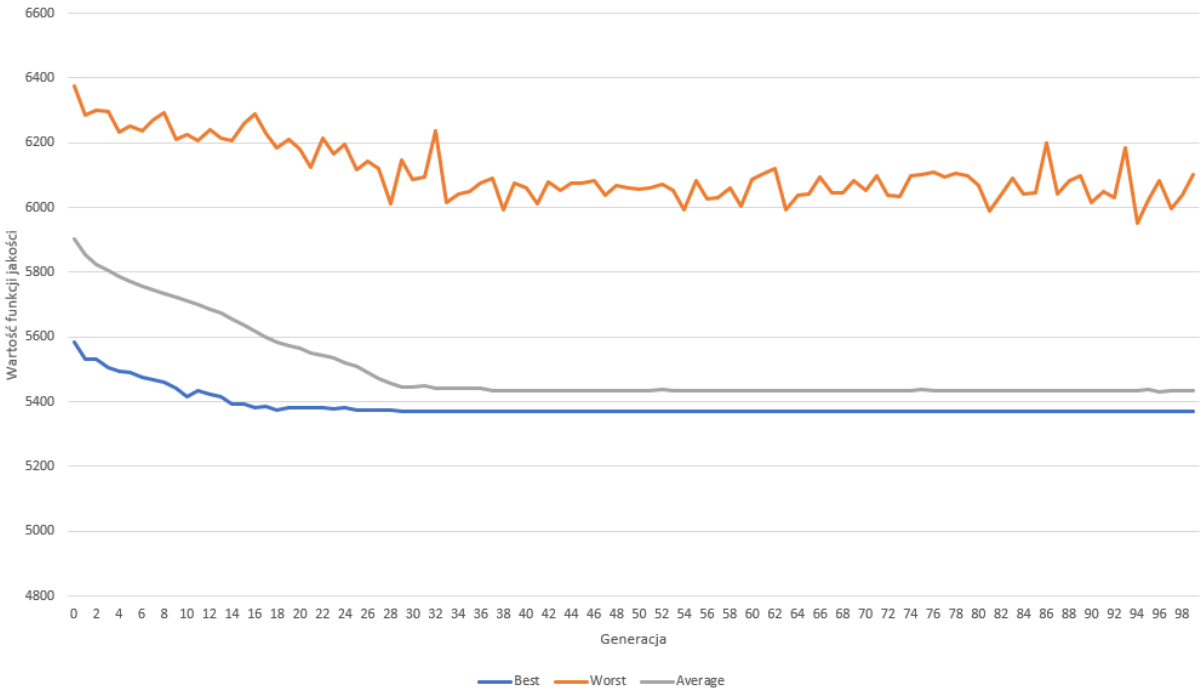
4.3.4 Duża populacja

Dla dużych populacji algorytm zachowuje swoją skuteczność, lecz parametr wpływa negatywnie na efektywność metody

Średni czas wykonania programu dla 10 testów: 9697ms

Liczba urodzeń: 1000000

```
final int POP_SIZE = 10000;
final int GEN = 100;
final double PM = 0.03;
final double PX = 0.7;
final int TOUR = 8;
String filePath = "input/had18.dat.txt";
```



4.4 Zbadanie wpływu selekcji na skuteczność GA – turniej, ruletka, losowa

4.4.1 Wpływ parametru T na selekcję turniejową

Dla parametru T = 1, jest to selekcja losowa. Algorytm nie jest skuteczny.

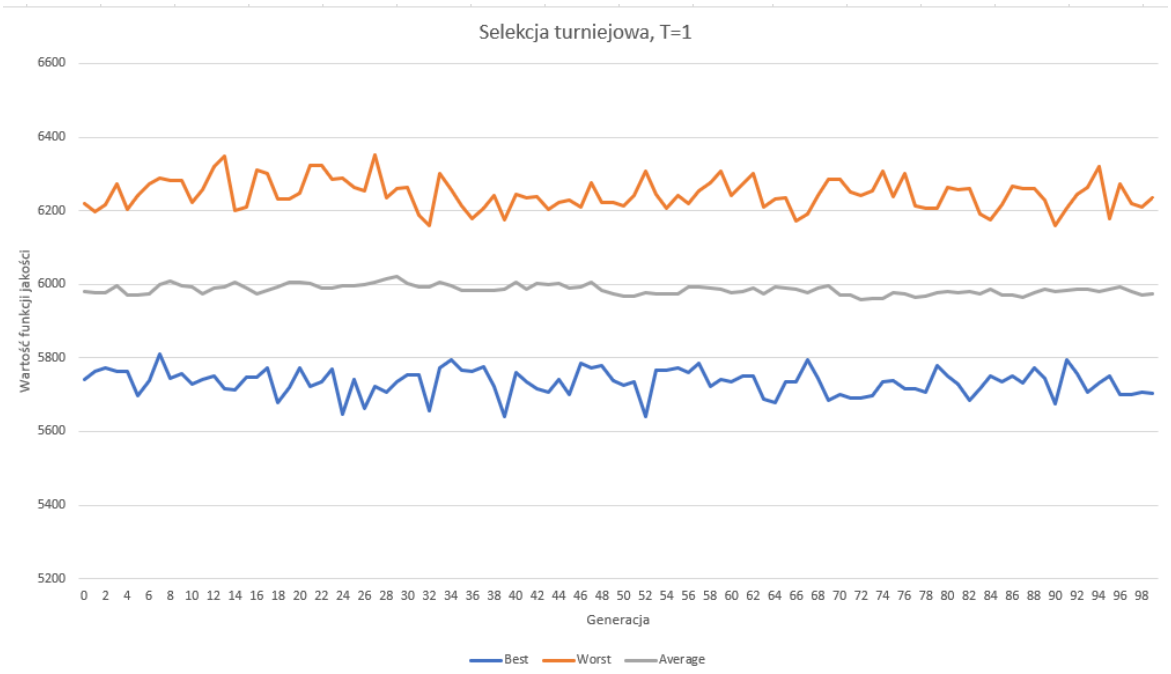
Dla parametru T = 100 (T = POP_SIZE), do kolejnego kroku po selekcji przechodzi zawsze najbardziej przystosowany osobnik. Nie jest to dobre podejście i ma dużą złożoność obliczeniową.

Zwiększając parametr T zauważono, że już przy wartości 3 i więcej, algorytm działa poprawnie. Wraz ze wzrostem wielkości turnieju, rośnie złożoność i czas działania algorytmu. Wartość funkcji oceny musi być liczona dla wszystkich T osobników. Optymalny parametr należało dobrać tak, by nie była to selekcja losowa i jednocześnie, by efektywność selekcji była na wysokim poziomie.

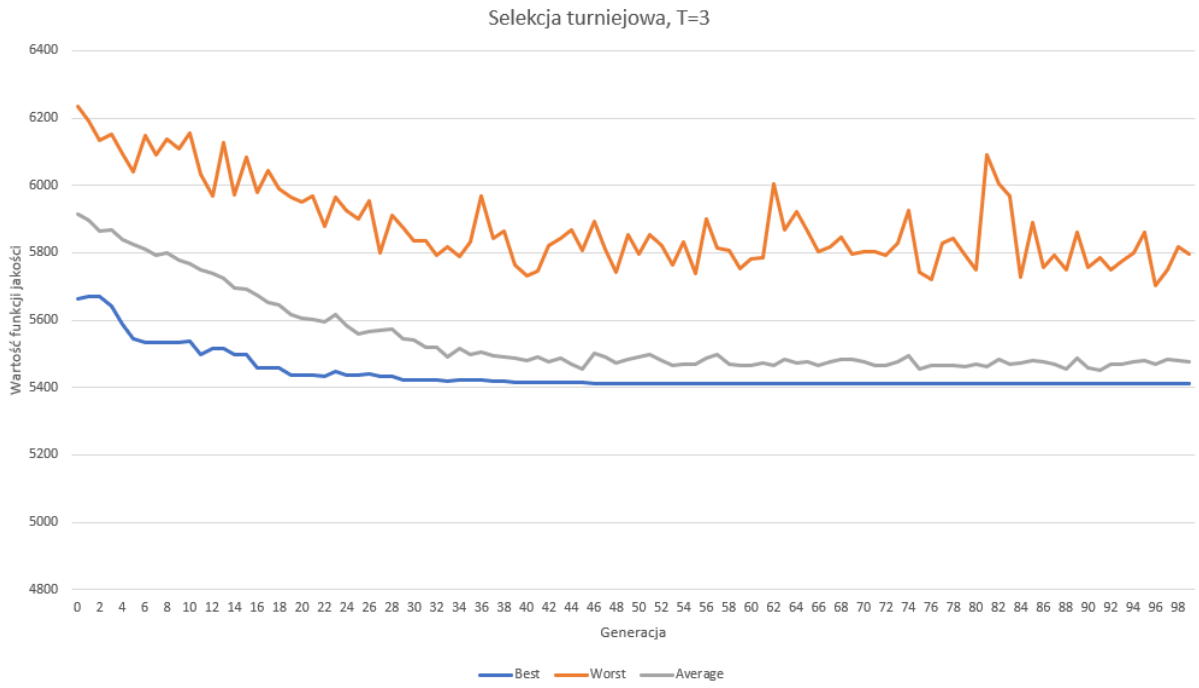
Parametry zastosowane w badaniu, T zmienne:

```
final int POP_SIZE = 100;
final int GEN = 100;
final double PM = 0.03;
final double PX = 0.7;
final int TOUR = 8;
String filePath = "input/had18.dat.txt";
```

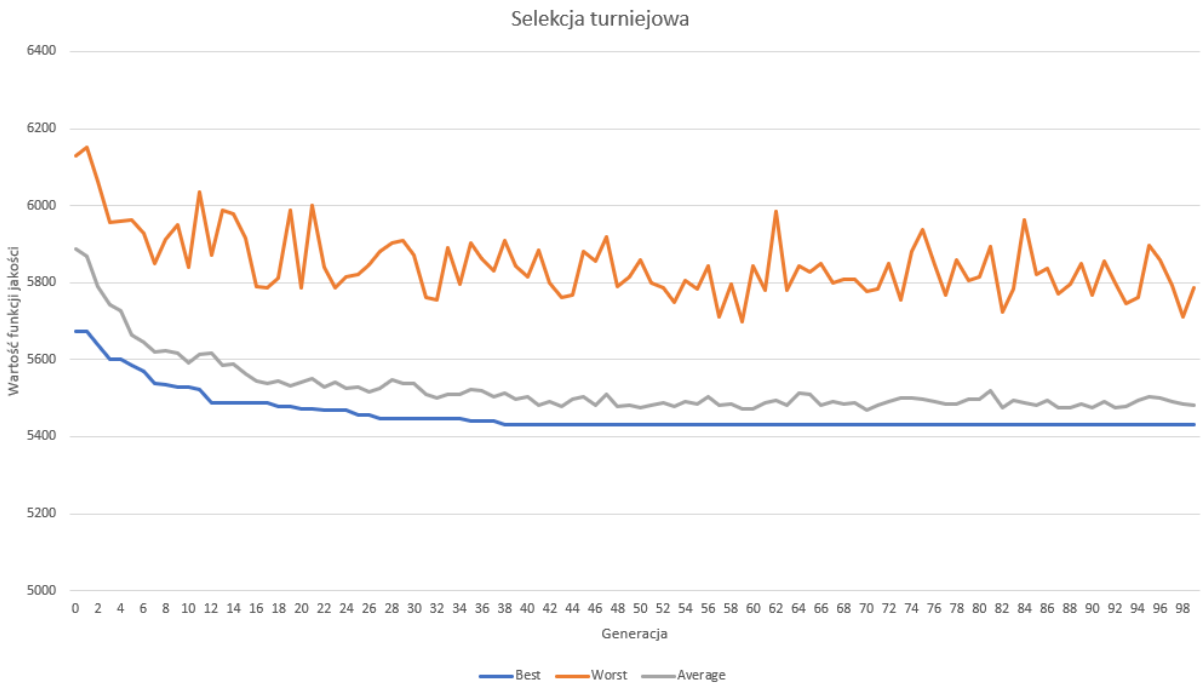
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 1): 198ms



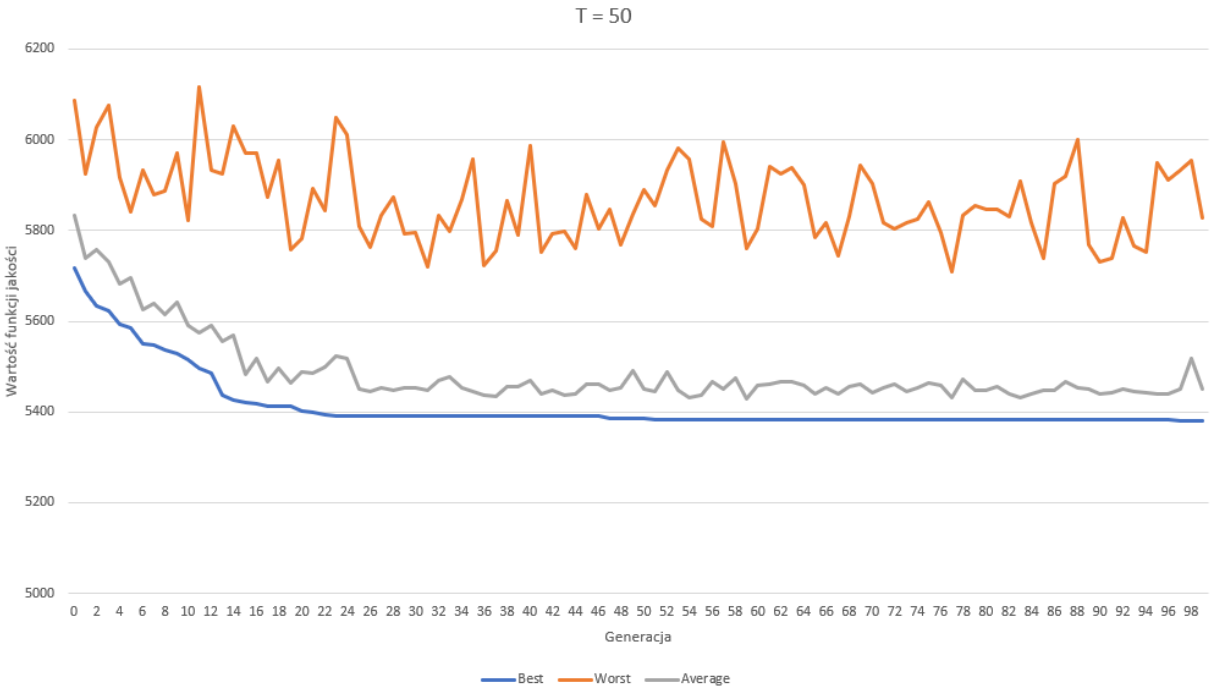
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 3): 201ms



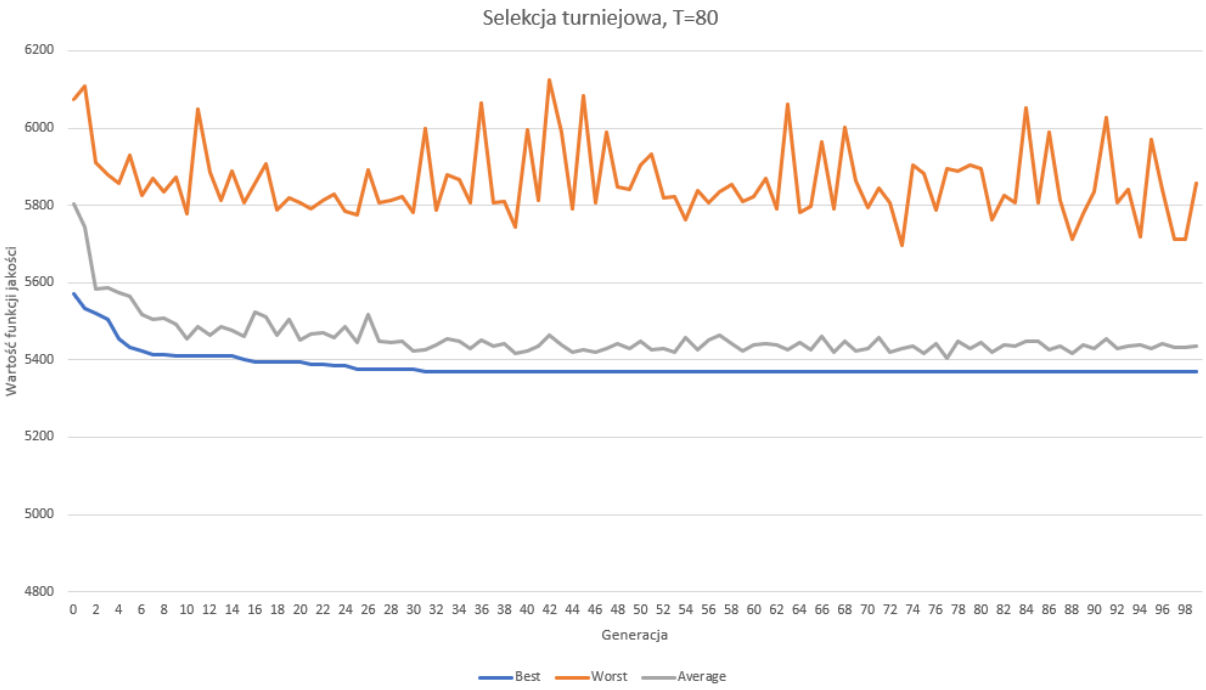
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 8): 225ms



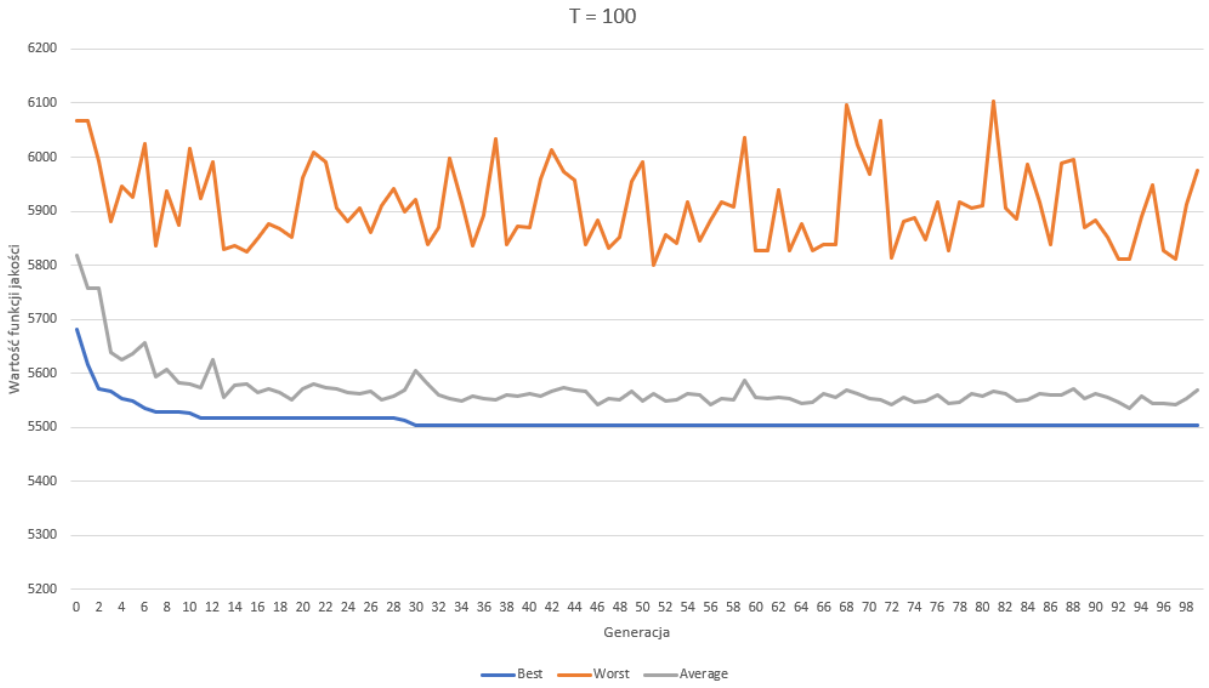
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 50): 480ms



Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 80): 680ms

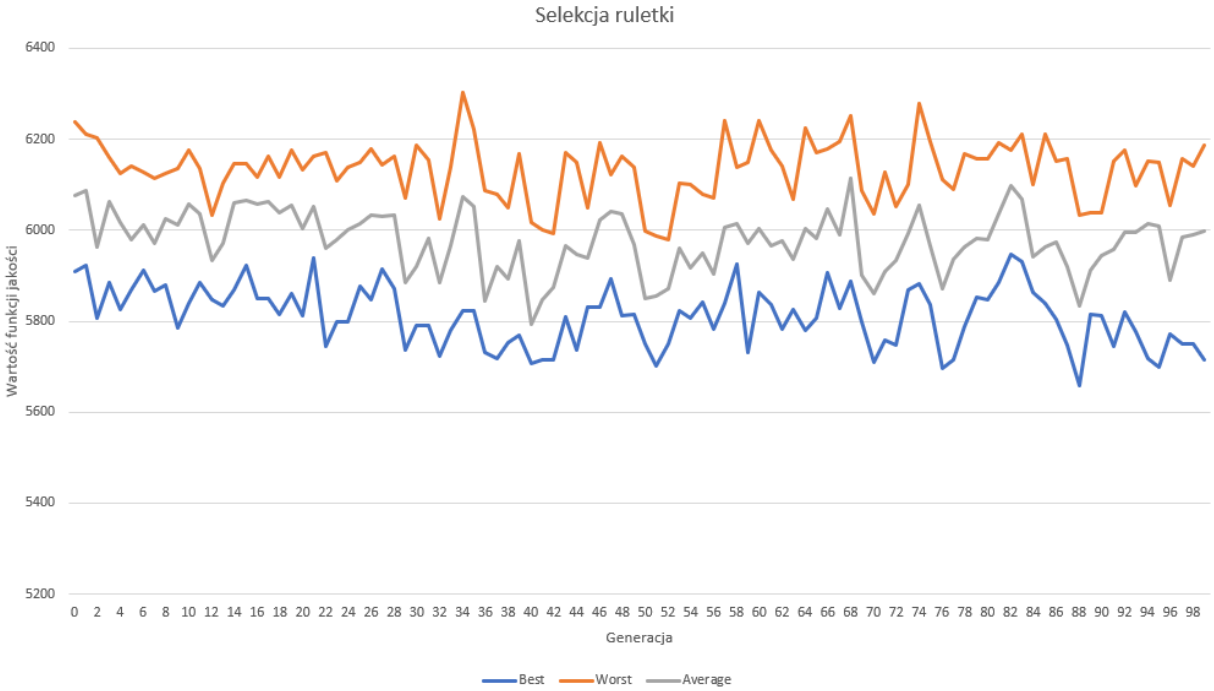


Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 100): 787ms

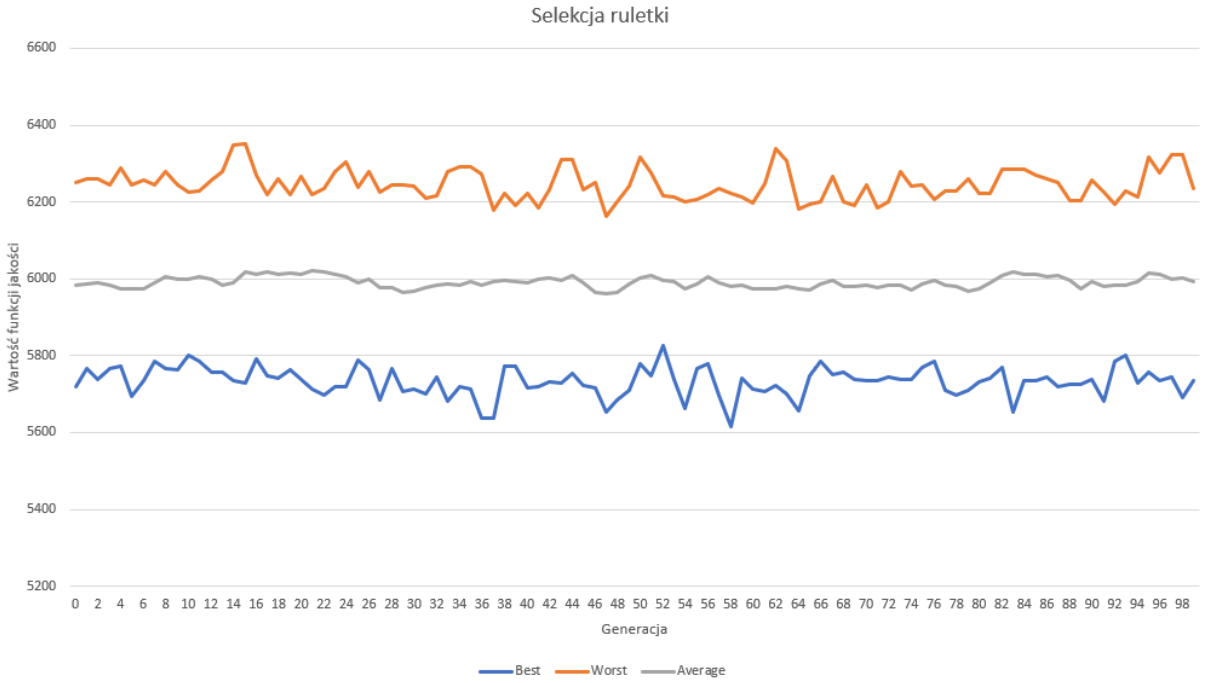


4.4.2 Porównanie turnieju z ruletką oraz selekcją losową

Średni czas wykonania programu dla selekcji z wykorzystaniem ruletki: 789ms



Średni czas wykonania programu dla selekcji losowej: 176ms



Selekcja metodą ruletki jest mało skuteczna, ponieważ różnice miedzy wartościami funkcji jakości są małe. Dla lepszego działania należałoby przeskalować te wartości, by różnice (a także prawdopodobieństwo ich wyboru) były większe. Algorytm bez takiego przeskalowania nie jest skuteczny. Czas działania programu wspieranego selekcją za pomocą ruletki jest zbliżony do czasu działania selekcji turniejowej z parametrem T=100. Selekcja ruletki bez przeskalowania jest nieskuteczna oraz nieefektywna.

Selekcja losowa przedstawiona została jako punkt odniesienia do selekcji turniejowej. Na tym etapie można zauważyć, że ruletka działająca z parametrem T=1 działa tak jak selekcja losowa. Brak skuteczności algorytmu.

4.5 Porównanie skuteczności GA z wynikami dwóch innych metod nieewolucyjnych, badanie dla 5 plików tekstowych

	EA		inicjalizacja losowa		Alg. zachłanny	
	AVG	**OS**	**AVG**	**OS**	**AVG**	**OS**
had12	1699,64	23,02	1879	56	1827,9	25,0
had14	2788,37	53,83	3156	79	3049,3	35,1
had16	3813,08	73,47	4230	82	4061,8	89,7
had18	5492,12	82,62	5978	100	5789	85
had20	7139,60	119,42	7760	122	7484,1	110,9

Dla każdej populacji policzono średnią wartość funkcji jakości. Następnie policzono średnią wartość dla wszystkich generacji. I tą ostatnią wartość wyznaczono dla 10 testów programu. Takie badanie powtórzono dla wszystkich plików tekstowych oraz wszystkich badanych algorytmów.

Dla wszystkich wartości policzono również odchylenia standardowe (OS). Dla EA użyto selekcji turniejowej z wytypowanymi parametrami w *punkcie 4.1.*

Zbadano również średnie czasy działania metod dla pliku *had18:*
GA: 825ms
Inicjalizacja losowa: 55ms
Alg. Zachłanny: 56ms

Inicjalizacja losowa jest tutaj punktem odniesienia dla pozostałych dwóch algorytmów. Wyniki inicjalizacji losowej są największe spośród trzech metod. Nie jest to skuteczna metoda dla tego zadania.

Zastosowanym algorytmem zachłannym pokazano, że stosunkowo prostym w implementacji algorytmem można rozwiązać zadanie i otrzymać wyniki spełniające pewne kryteria. Nie jest to najlepszy algorytm pod względem skuteczności, lecz przy stosunkowo wysokiej skuteczności, czas jego działania zbliżony jest do czasu działania inicjalizacji losowej.

W celu otrzymania jeszcze lepszych wyników należy użyć algorytmu generycznego, którego wyniki są najmniejsze, więc spełniają bardziej wymagające kryteria. Zmierzony czas działania tego algorytmu jest największy spośród badanych.

Zarówno algorytm zachłanny jak i algorytm generyczny są skutecznymi algorytmami przy tym zadaniu.

5. Podsumowanie

Algorytm generyczny jest bardzo ciekawym sposobem rozwiązania przedstawionego zadania. Jest to metoda bardzo skuteczna, lecz trzeba pamiętać że jest metaheurystyką, a jej rozwiązanie jest tylko przybliżeniem. Heurystyka może zwrócić rozwiązanie w postaci najlepszego wyniku, lecz nie mamy pewności, czy właśnie takie otrzymaliśmy. Jest to algorytm który można zastosować w przypadku, gdzie inne algorytmy wykazują zbyt dużą złożoność obliczeniową i nie ma możliwości otrzymania wyniku spełniającego oczekiwania użytkownika w odpowiednim czasie.

Na działanie algorytmu wpływa to, jak zdefiniujemy jego składowe takie jak *selekcja, krzyżowanie, mutacja oraz funkcja oceny.* Budując oraz implementując omawiany algorytm należy zastanowić się czego oczekujemy i wytypować odpowiednie parametry działania.

Parametry prawdopodobieństwa mutacji oraz krzyżowania muszą mieć wartości które pozwolą na posiadanie zróżnicowanej populacji. Ilość generacji musi być odpowiednia, by algorytm był w stanie na jej przestrzeni wyznaczyć rozwiązanie. W przypadku wyznaczania najlepszych osobników z populacji, najlepszymi znalezionymi parametrami działania algorytmu były: PM = 3%, PX = 70%, TOUR = 8, POP_SIZE = 100, GEN = 100.

Najbardziej kosztowne metody programu to odczyt macierzy przepływu i macierzy odległości z pliku tekstowego, oraz metoda selekcji turniejowej.