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).

W dalszej części przedstawiono również 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>

1. **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
  + Ruletka
  + Losowa
* **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** -

**Parametry programu**:

**POP\_SIZE** = 100  
**GEN** = 100  
**PM** = 0.03  
**PX** = 0.7  
**TOUR** = 8  
**filePath**

**macierzOdleglosci**

**macierzPrzeplywu**

1. **Zbudowanie algorytmu genetycznego**
   1. **Osobnik**

public class Person {  
 private List <Integer> genotype;  
  
 public Person(List genotype) {  
 this.genotype = genotype;  
 }

}

* 1. **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;  
}

* 1. **Krzyżowanie**

1. 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;  
   }
2. **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));  
 }  
 }  
}

1. **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;  
}

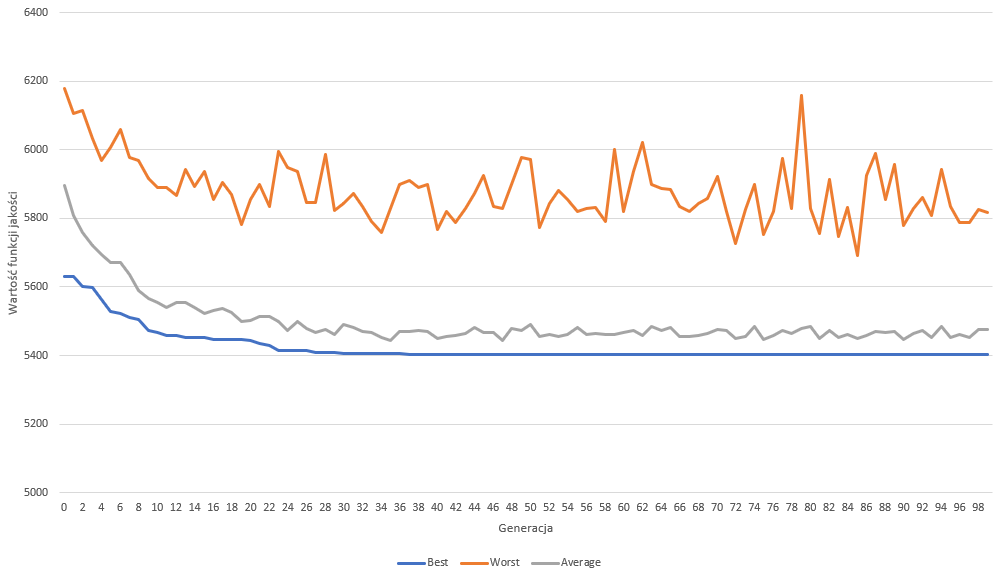
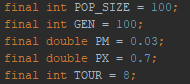
1. **Selekcja**

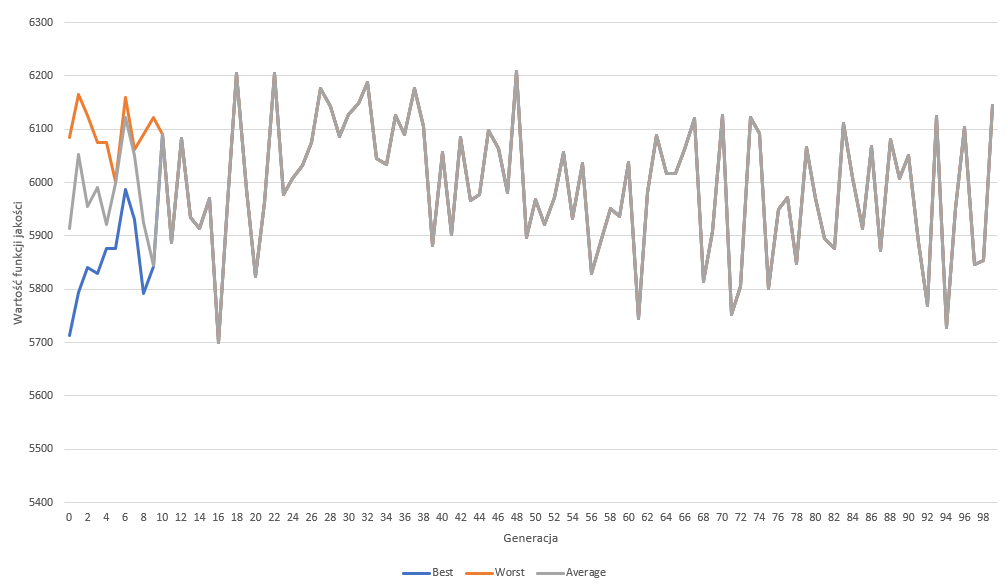
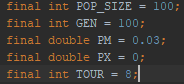
public Person tourSelect(ArrayList <Person> pop, int TOUR, int[][] distanceMatrix, int[][] flowsMatrix) {  
 Random rn = new Random();  
 ArrayList <Person> selectedTour = new ArrayList <Person>();  
 for (int i = 0; i <TOUR; i++) {  
 int randomRow = rn.nextInt(pop.size());  
 selectedTour.add(pop.get(randomRow));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

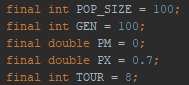
public Person rouletteSelect(ArrayList <Person> pop, int[][] distanceMatrix, int[][] flowsMatrix) {  
 int popSize = pop.size();  
 double totalSum = 0;  
 for(int i = 0; i < popSize; i++){  
 totalSum += (1000)/(pop.get(i).checkValue(distanceMatrix, flowsMatrix));  
 }  
  
 Random rn = new Random();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

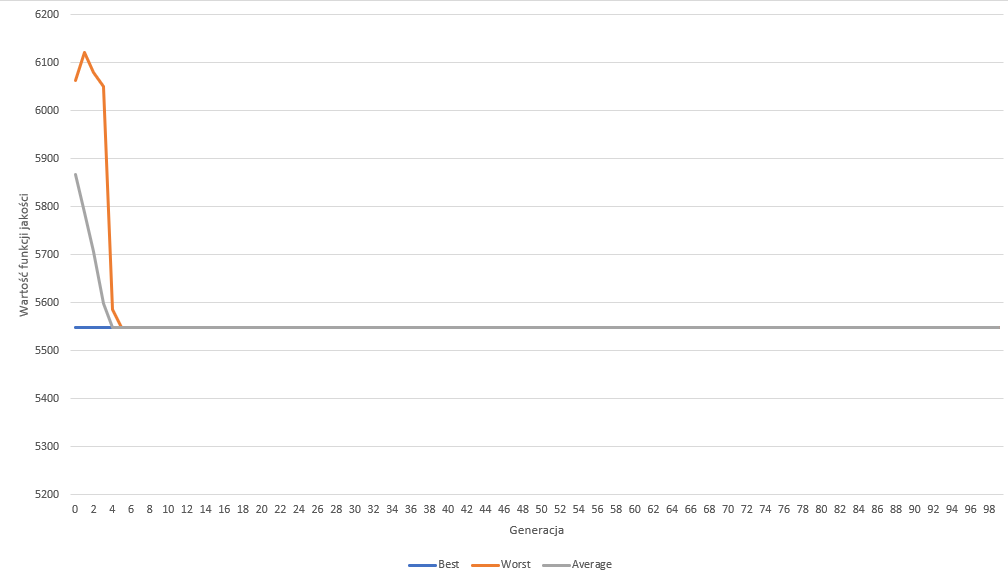
public Person randomSelect(ArrayList <Person> pop, int[][] distanceMatrix, int[][] flowsMatrix) {  
 int popSize = pop.size();  
 Random rn = new Random();  
 int randomIndex = rn.nextInt(popSize);  
 return pop.get(randomIndex);  
}

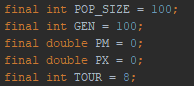
1. **Badania**
   1. Zbadanie wpływu prawdopodobieństwa krzyżowania *Px* i mutacji *Pm* na wyniki działania GA

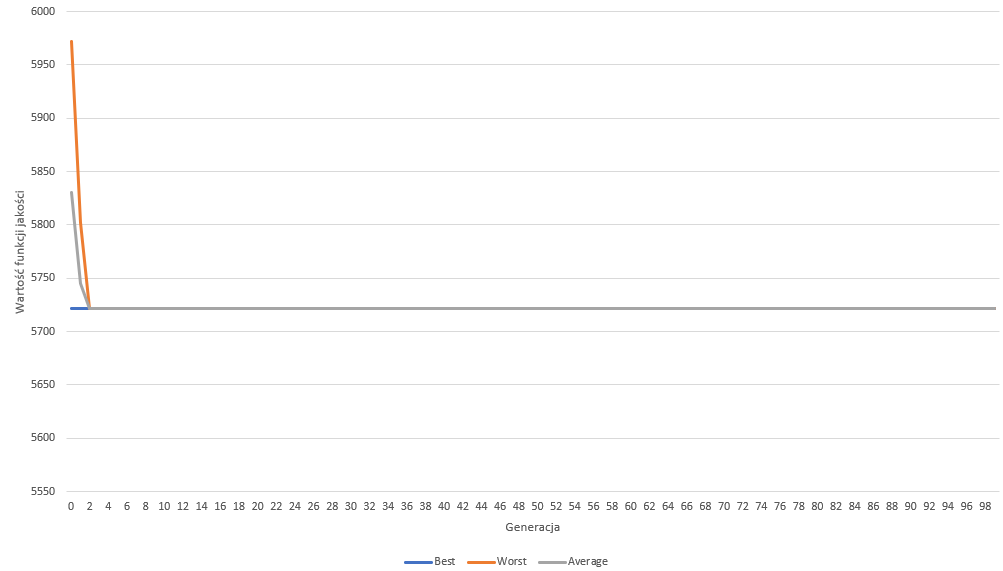
****

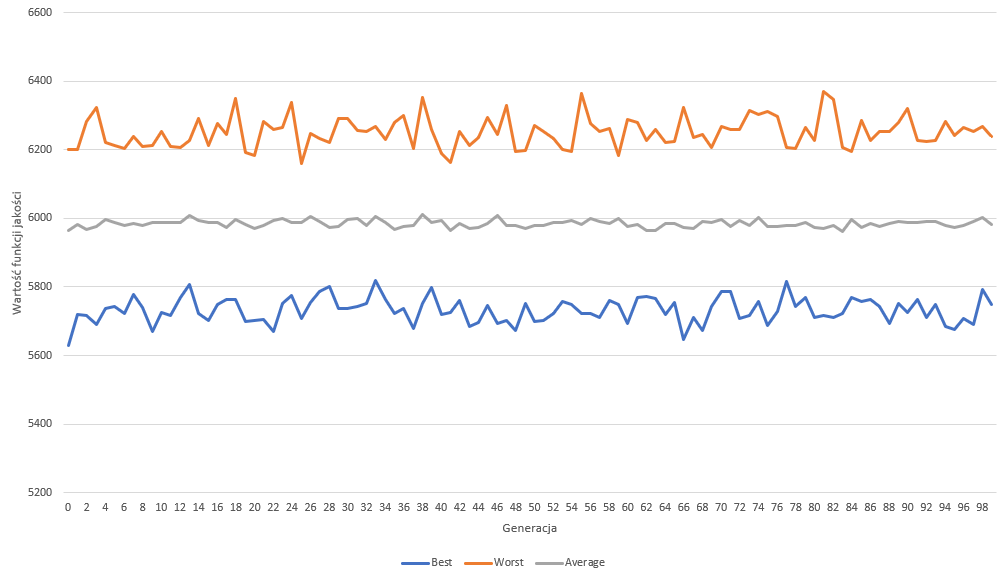
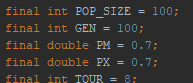


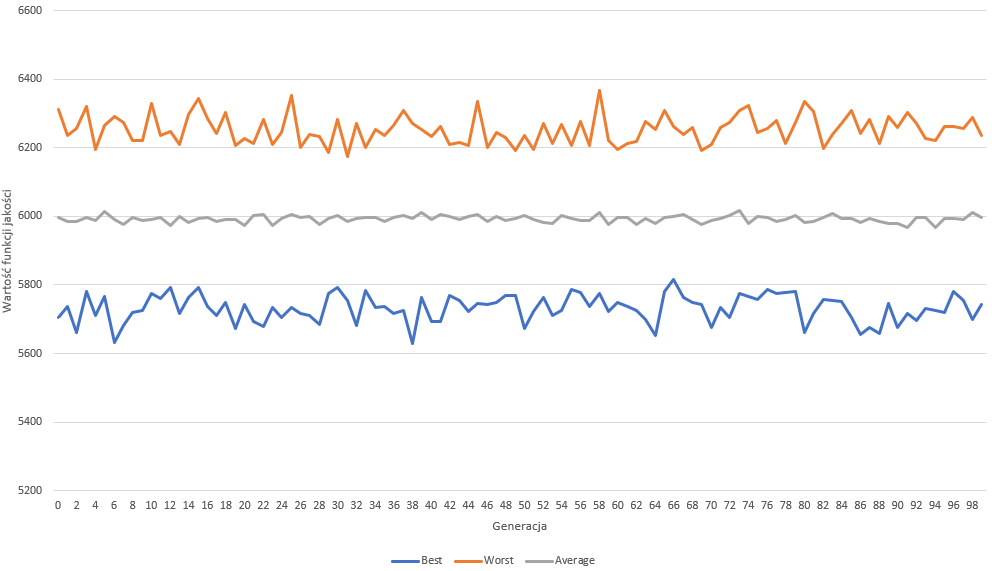
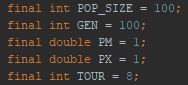




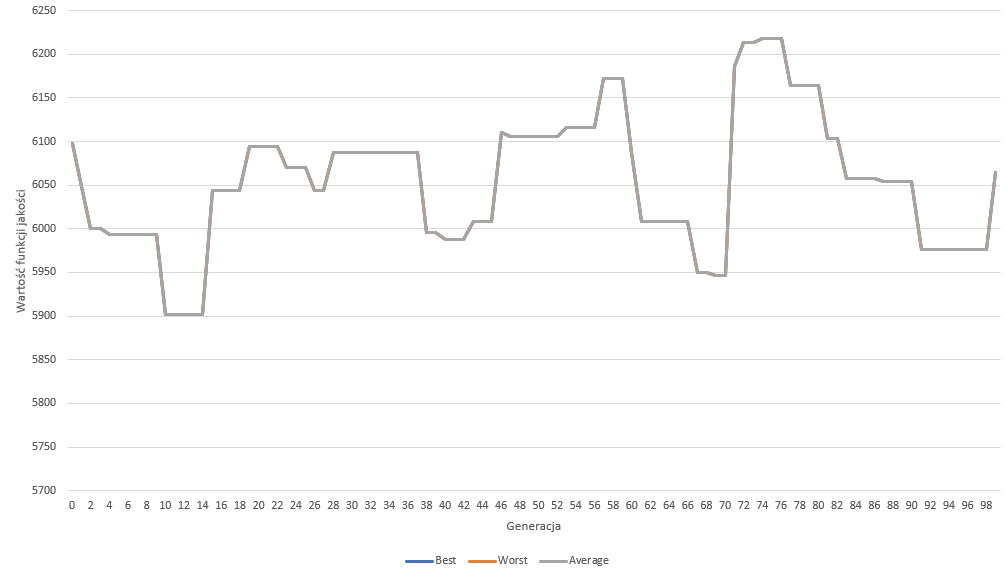
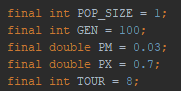


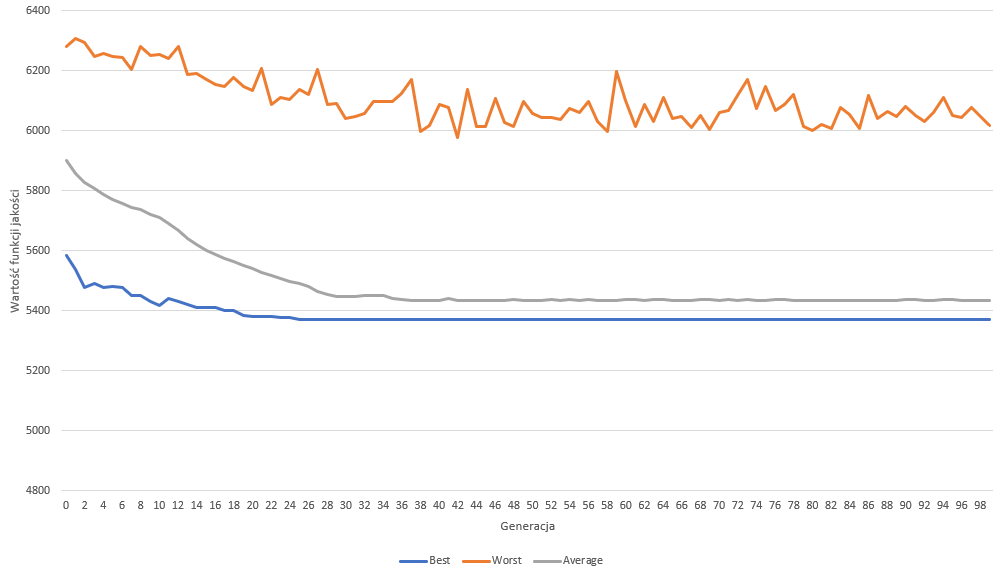
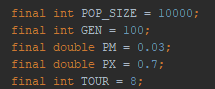


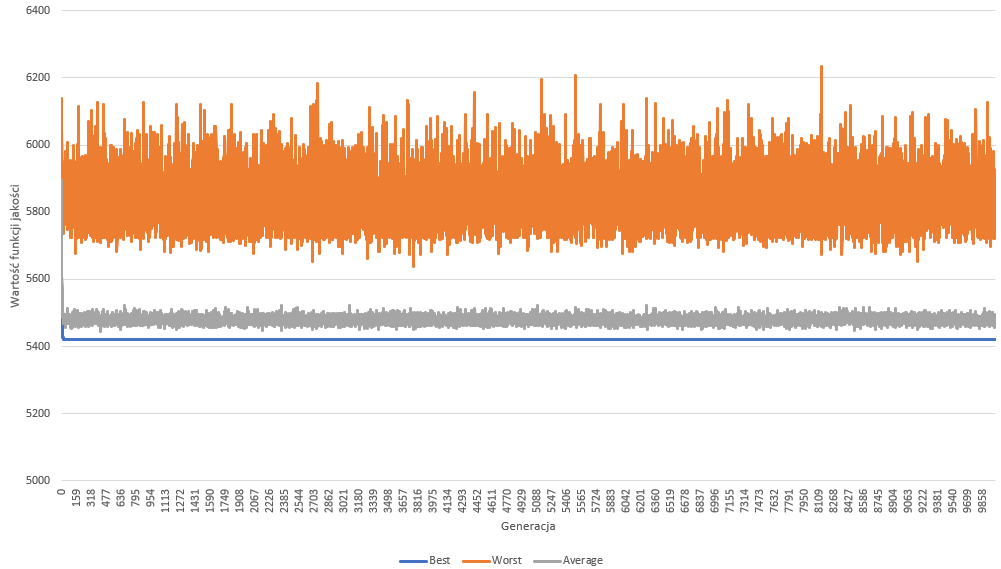
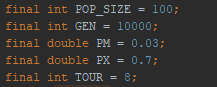




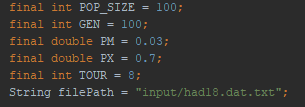
* 1. Zbadanie wpływu rozmiaru populacji *pop\_size* i liczby pokoleń *gen* na wyniki działania GA



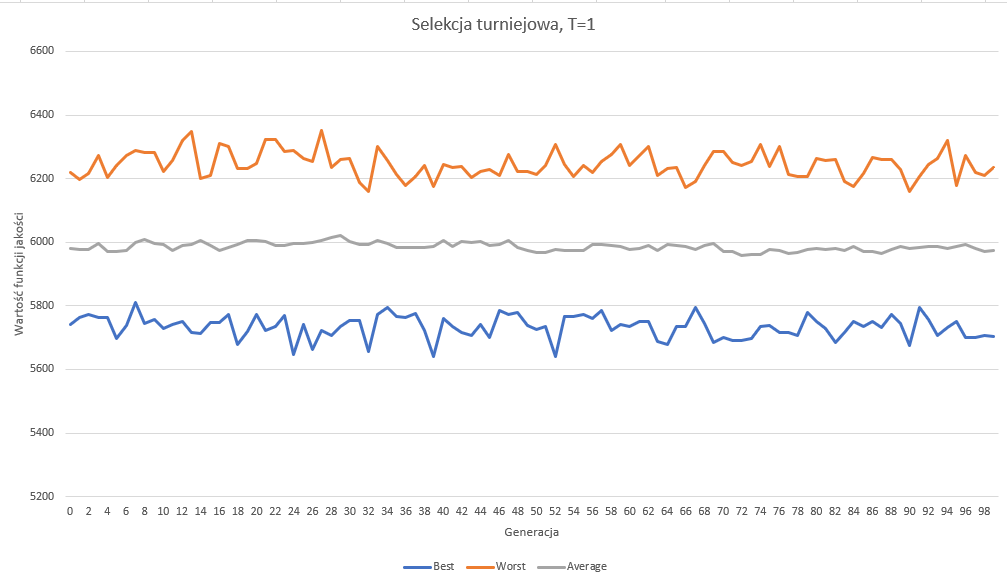




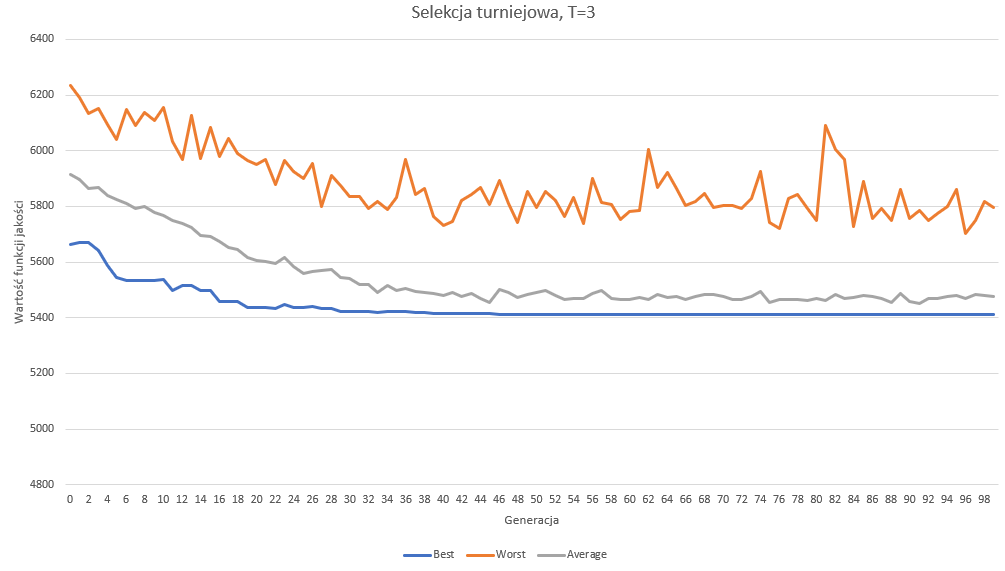
* 1. Zbadanie wpływu selekcji na skuteczność GA – turniej, ruletka, losowa



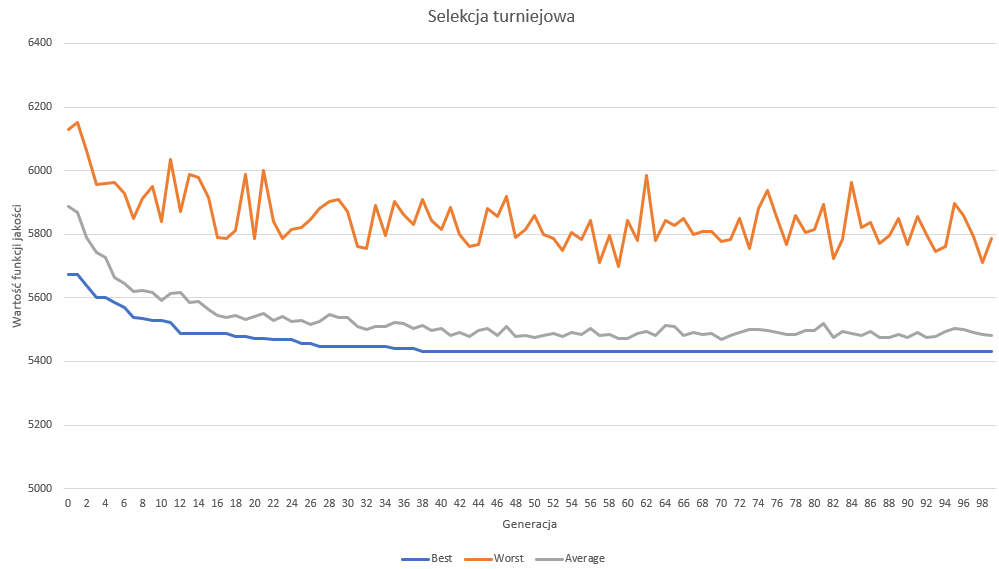
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 1): 198ms



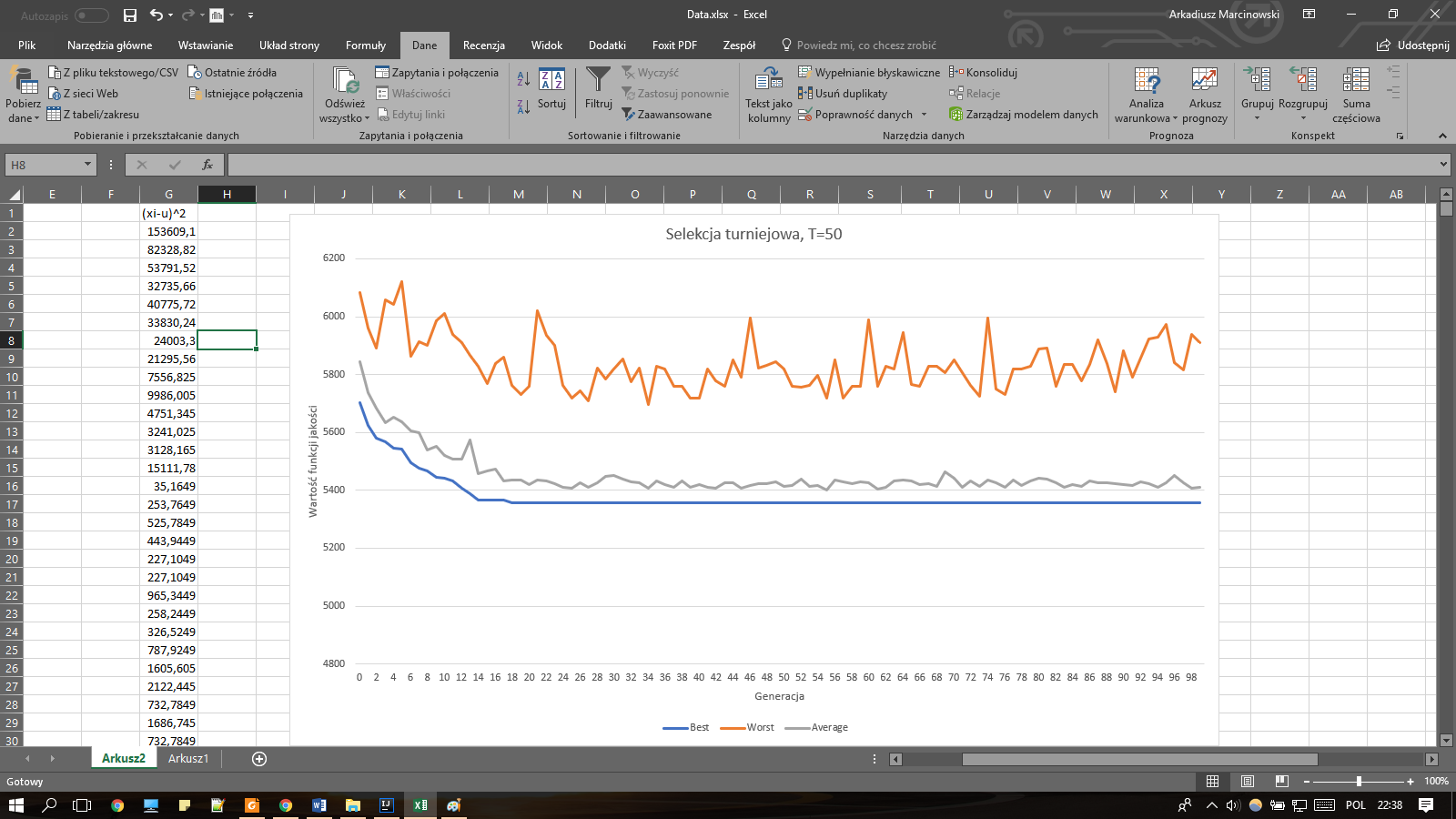
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 3): 201ms



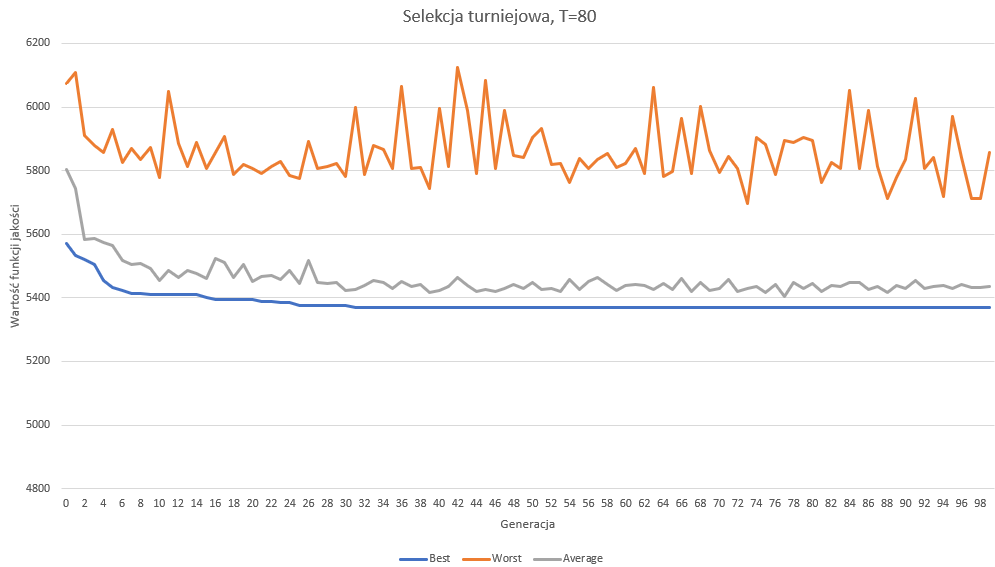
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 8): 225ms



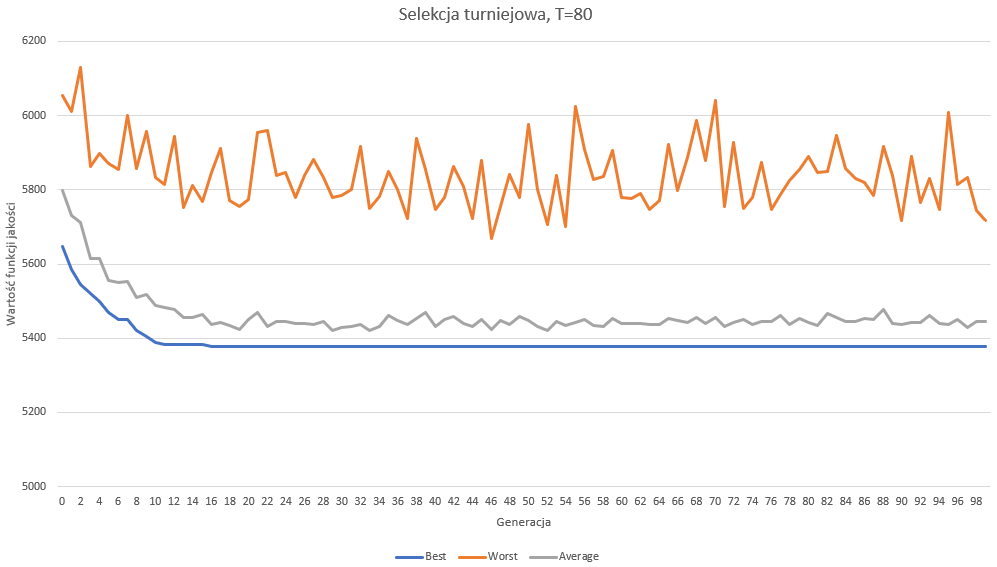
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 50): 480ms



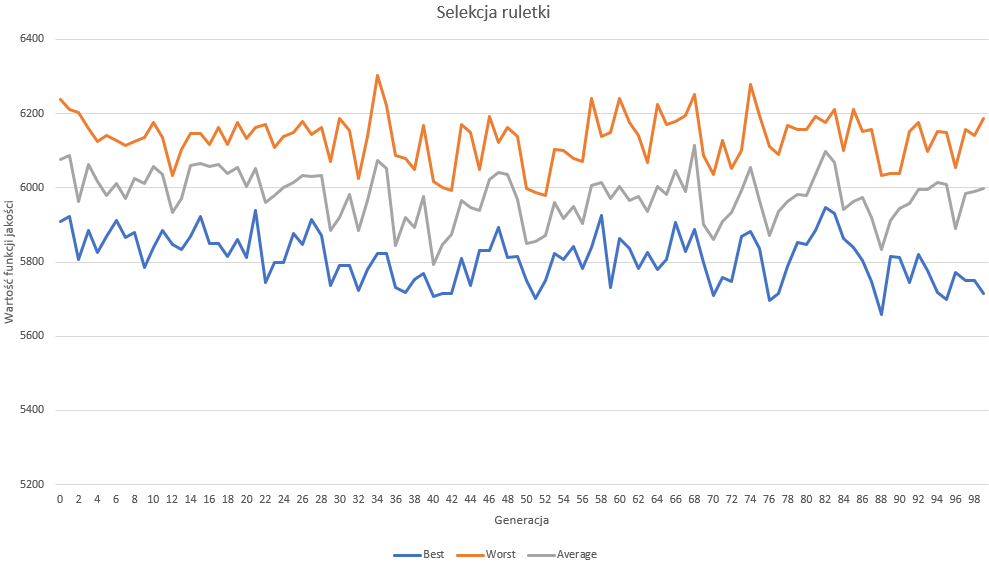
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 80): 680ms



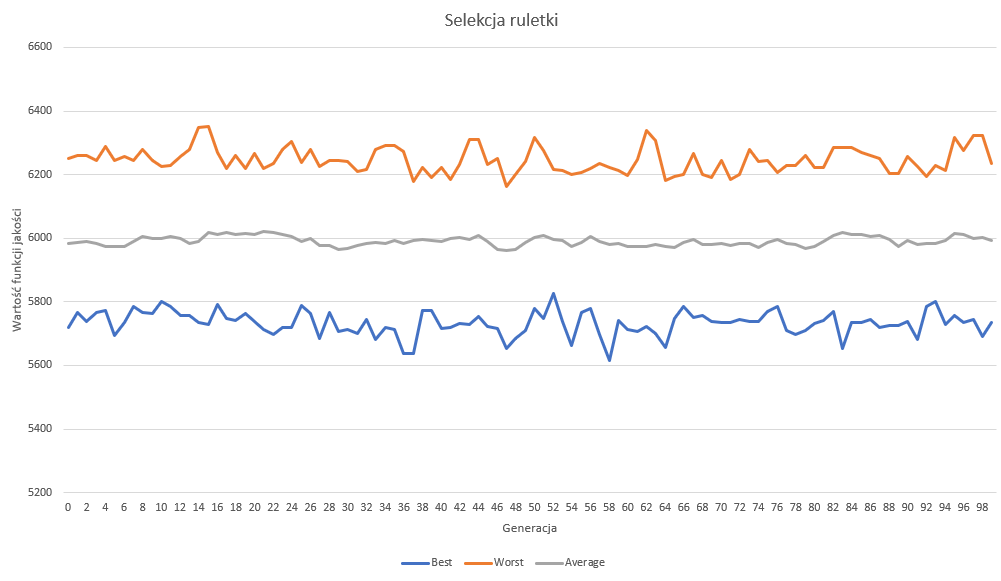
Średni czas wykonania programu dla selekcji z wykorzystaniem turnieju (T = 100): 787ms



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



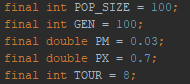
Średni czas wykonania programu dla selekcji losowej: 176ms

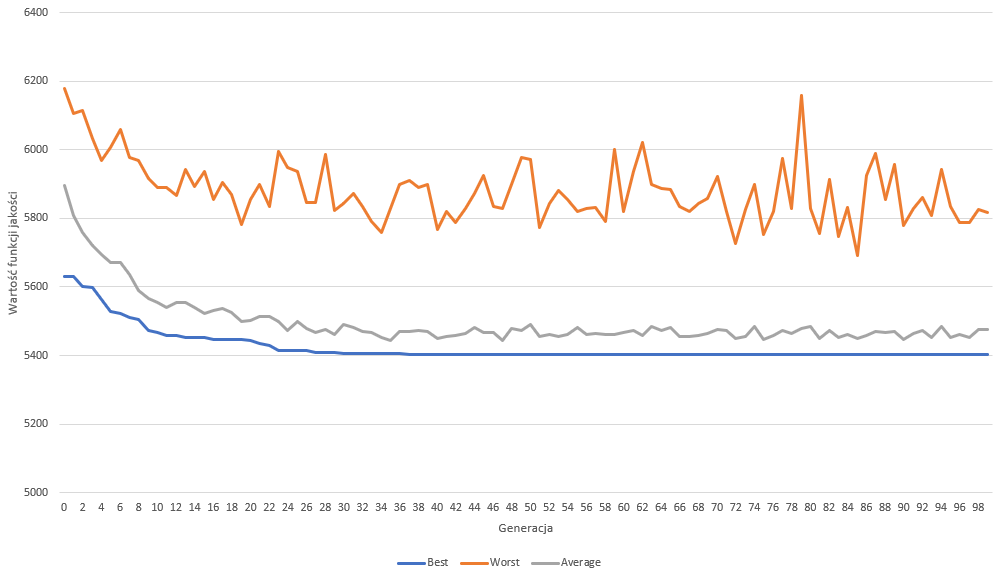


* 1. Porównanie skuteczności GA z wynikami dwóch innych metod nieewolucyjnych, badanie dla 5 plików tekstowych

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **EA (x10)** | | **inicjalizacja losowa** | | **Greedy (x10)** | |
|  | **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 |

Algorytm generyczny z selekcją turniejową:

****

****

1. **Podsumowanie**