Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

# Projektowanie efektywnych algorytmów

Algorytm genetyczny

Etap nr 3

Prowadzący:

*dr inż. Jarosław Rudy*

Grupa:

*A - Wtorek 17:05 TP*

Autor:

*Daniel Glazer, 252743*

Wrocław 23.01.2022

# Spis treści

# Spis tabel

# Spis rysunków

# 1 Wprowadzenie teoretyczne

## 1.1 Opis metody

### 1.1.1 Algorytm genetyczny

Algorytm genetyczny (ang. *Genetic Algorithm (GA)*) jest jednym z metaheurystycznych algorytmów przeszukujących przestrzeń alternatywnych rozwiązań problemu. Swoje założenia bierze z naturalnego procesu ewolucji biologicznej i zalicza się do algorytmów ewolucyjnych. W trakcie ewolucji gatunki stykają się z problemem lepszej adaptacji do skomplikowanego i zmiennego środowiska. Problem ten można też przełożyć na sposób przeszukiwania przestrzeni problemu, w taki sposób, że naśladuje on pewne procesy naturalne, takie jak dziedziczenie genetyczne czy darwinowską walkę o przeżycie. Dlatego też do opisu algorytmu genetycznego używa się słownictwa zapożyczonego z genetyki naturalnej. Mówi się o populacji, która zawiera osobniki lub inaczej zwane chromosomy, każdy chromosom zawiera geny, które świadczą o jakiejś cesze. W przypadku problemu *TSP*, populacja to zbiór rozwiązań problemu. Każde jedno rozwiązanie zapisane jest w jednym chromosomie, a geny odpowiadają za kolejne wierzchołki ścieżki. Aby zastosować algorytm w zadaniach optymalizacyjnych stosuje się go według struktury algorytmu ewolucyjnego. Działanie algorytmu rozpoczyna się od wylosowania populacji początkowej, następnie stosuje się metody selekcji, które mają za zadanie wybrać chromosomy do krzyżowania. Podczas krzyżowania wymieniane są geny osobników w celu utworzenia potomków, zawierających połączone geny rodziców. Potem przeprowadzana jest mutacja, która polega na losowej zmianie jednego lub więcej genów wybranego osobnika. Ma to na celu wprowadzenie dodatkowej zmienności w populacji. W kolejnym kroku populacja przechodzi do następnej generacji, w której zostawia się określoną liczbę osobników. Algorytm ten charakteryzuje się brakiem pewności znalezienia optimum brak jawnie sprecyzowanej złożoności obliczeniowej, ponieważ praca algorytmu kończy się wraz z warunkiem zakończenia, który jest zależny od parametrów.

Szczególnym przypadkiem algorytmu genetycznego jest algorytm memetyczny nazywany również hybrydowym algorytmem ewolucyjnym. Różni się od zwykłego algorytmu genetycznego tym, że poprzez dołączenie algorytmu poszukiwań lokalnych do metody wyboru najlepszego sąsiedztwa podczas mutacji, można uzyskać zdecydowaną poprawę osiągów algorytmu. Wynika to z tego, że algorytmy ewolucyjne są dobre w odnajdywaniu obszaru, w którym znajduje się optimum, ale kiepsko radzą sobie w odnalezieniu optimum lokalnego wewnątrz tego obszaru.

# 2 Opis implementacji algorytmu

## 2.1 Wstęp

Na starcie wywoływanego algorytmu, wywoływana jest klasa *Matrix*, której celem jest wczytanie danych z pliku i utworzenie dwuwymiarowej tablicy dynamicznej posiadającej długości ścieżek.

## 2.2 Parametry startowe algorytmu

- probability - współczynnik odpowiadający za prawdopodobieństwo wystąpienia mutacji dla potomka
- populationSize - liczba krzyżowań w jednej generacji
- populationCopyNumber - początkowa liczba osobników oraz liczba przenoszonych osobników do kolejnej generacji
- generationNumber - liczba generacji
- selectionType - parametr odpowiadający za wybór sposobu selekcji rodziców do krzyżowania, gdzie:
    0. selekcja koła ruletki
    1. selekcja turniejowa
    2. selekcja rankingowa
- crossoverType - parametr pozwalający na wybranie jednej z dwóch metod krzyżownia, gdzie:
    0. *Partially Mapped Crossover*
    1. *Order Crossover Operator*

## 2.3 Opis działania

Działanie algorytmu rozpoczyna się od podania parametrów startowych. Następnie za pomocą funkcji *shuffle* z biblioteki *algorithm* generowana jest populacja startowa, której wielkość zależy od parametru *populationCopyNumber*. Podczas generowania osobników, obliczany jest dla nich koszt ścieżki, który następnie zapisywany jest do mapy wraz z wektorem zawierającym kolejność wierzchołków w ścieżce. Jeśli koszt ścieżki będzie mniejszy niż obecne najlepsze rozwiązanie, wartość najlepszego rozwiązania zostaje nadpisana wraz z ścieżką, dla której zostało policzone najlepsze rozwiązanie. W kolejnym kroku wywoływana jest metoda odpowiedzialna za funkcjonowanie głównej pętli programu. Główna pętla programu zawiera zagnieżdżoną pętlę, której liczba iteracji jest zależna od parametru *populationSize*. Na początku każdej iteracji wewnętrznej pętli tworzone są dwa wektory wypełnione zerami odpowiadające za ścieżki potomków. W kolejnym kroku na postawie parametru *selectionType* wykorzystywana jest jedna z trzech metod selekcji. Metoda selekcji zwraca indeksy rodziców przechowywanych w mapie. Następnie rodzice wraz z wektorami potomków są przekazywani do jednej z dwóch metod krzyżowania zależnych od *crossoverType*. Po krzyżowaniu osobników i uzupełnieniu

potomków genami, obliczana jest dla nich wartość ścieżki. Po obliczeniu kosztu drogi, potomkowie są porównywani z globalnie najlepszym rozwiązaniem. W momencie, w którym posiadają lepsze rozwiązanie, zastępują je. Przed samym zakończeniem zagnieżdżonej pętli, sprawdzany jest warunek wystąpienia mutacji. Jeśli wylosowane prawdopodobieństwo jest mniejsze niż *probability* i większe niż 0.1 to zostanie wykonana mutacja na podstawie przeszukiwania całego sąsiedztwa. W poszukiwaniu uwzględniona jest taka zamiana wierzchołków, która jest najlepszym lokalnym rozwiązaniem. Aby uniknąć zbiegania algorytmu oraz w idei dywersyfikacji dla wylosowanego prawdopodobieństwa mniejszego i równego 0.1 wykonuje się metoda, która dokonuje zamiany wierzchołków w sposób losowy. Taki sposób może doprowadzić do lokalnie gorszego rozwiązania, ale w kolejnych iteracjach istnieje szansa, że znajdzie rozwiązanie lepsze niż obecne najlepsze rozwiązanie globalne. Osobnik, który jest zmutowaną wersją potomka dodawany jest do populacji. Po otrzymaniu warunku stopu dla zagnieżdżonej pętli, wykonywane jest sortowanie populacji, po którym następuje skopiowanie najlepszych osobników z populacji o liczbie równej *populationCopyNumber* do populacji z kolejnej generacji, po skopiowaniu zwiększamy iterator głównej pętli. Dla głównej pętli warunkiem stopu jest liczba iteracji, która nie przekroczy parametru odpowiedzialnego za liczbę generacji (*generationNumber*). Po otrzymaniu warunku stopu dla pętli głównej, program wyświetla najlepsze znalezione rozwiązanie i kończy swoje działanie.

## 2.4 Opis metod selekcji

### 2.4.1 Selekcja koła ruletki

Selekcja koła ruletki wprowadza do implementacji dodatkowy element, którym jest tablica zdatności. Tablica ta zawiera prawdopodobieństwa na wybranie osobników populacji. Prawdopodobieństwa są liczone na postawie odchylenia od obecnego najlepszego rozwiązania. Im rozwiązanie jest gorsze tym ma mniejsze prawdopodobieństwo na wybranie. Aby wybrać chromosomy metodą koła ruletki, należy sumować prawdopodobieństwa zapisane w tablicy do momentu aż przekroczą, lub będą równe wartości prawdopodobieństwa, które zostało wylosowane. W takim momencie zwracany jest osobnik, dla którego suma spełniła warunek.

### 2.4.2 Selekcja rankingowa

Selekcja rankingowa podobnie jak koła ruletki wprowadza tablicę zdatności. Odróżnia ją jedynie sposób liczenia prawdopodobieństwa. Prawdopodobieństwo jest liczone na podstawie pozycji w rankingu posortowanej populacji. Pierwszy osobnik ma największą szansę na wylosowanie, a każdy kolejny mniejszą od poprzedniego. Tak samo jak w przypadku selekcji koła ruletki sumujemy prawdopodobieństwa i zwracamy osobnika, dla którego zostanie spełniony warunek przekroczenia lub równości sumy.

### 2.4.3 Selekcja turniejowa

Selekcja turniejowa opiera się na algorytmie, w którym losuje się 2 osobników i wybiera się najlepszego z nich. Aby móc wybrać dwóch osobników do krzyżowania, potrzeba przeprowadzić dwie takie selekcje.

## 2.5 Operatory krzyżowania

### 2.5.1 Partially Mapped Crossover (PMX)

Wybierane są losowo dwa punkty przecięcia (w tym wypadku "|") w rodzicach aby móc je przekopiować do potomków. W taki sposób, że z rodzica o indeksie 1 ($R_1$) kopiowany jest obszar wycięcia do dziecka o indeksie 2 ($D_2$) i schemat powtarzany jest dla rodzica 2 i dziecka 1:

$$R_1 = (3\ 4 \mid 8\ 2\ 7\ 1 \mid 6\ 5)$$
$$\mathrm{R}_2 = (4\ 2 \mid 5\ 1\ 6\ 8 \mid 3\ 7)$$
$$\mathrm{D}_1 = (x\ x \mid 5\ 1\ 6\ 8 \mid x\ x)$$
$$\mathrm{D}_2 = (x\ x \mid 8\ 2\ 7\ 1 \mid x\ x)$$

Następnie kopiowane są liczby na tych samym pozycjach z $R_1 \rightarrow D_1$ i $R_2 \rightarrow D_2$, które nie występują w poszczególnych chromosomach:

$$D_1 = (3\ 4 \mid 5\ 1\ 6\ 8 \mid x\ x)$$
$$D_2 = (4\ x \mid 8\ 2\ 7\ 1 \mid 3\ x)$$

W ostatnim kroku mapowane są wierzchołki w miejscach, w których nie udało się wprowadzić żadnej liczby. Dla przykładu na przedostatnim indeksie $D_1$ próbowano przepisać 6 z rodzica $R_1$, ale istniała już w tablicy genów, dlatego też mapujemy ją na liczbę z drugiego potomka, więc uzyskujemy, że $6 \rightarrow 7$. Na ostatnim indeks próbowano przepisać 5, 5 mapowana jest na 8, która występuje już w chromosomie, więc próbujemy mapować dalej aż dochodzimy do tego, że $5 \rightarrow 8 \rightarrow 1 \rightarrow 2$. Ostatecznie wpisujemy na ostatnim indeksie 2, w analogiczny sposób robimy to dla drugiego potomka i otrzymujemy:

$$D_1 = (3\ 4 \mid 5\ 1\ 6\ 8 \mid 7\ 2)$$
$$D_2 = (4\ 5 \mid 8\ 2\ 7\ 1 \mid 3\ 6)$$

### 2.5.2 Order Crossover Operator (OX)

Podobnie jak w przypadku Partially Mapped Crossover wybierane są losowo dwa punkty przecięcia w rodzicach. Ale w odróżnieniu od *PMX* kopiowane są obszary z rodzica do dziecka o tych samych indeksach:

$$R_1 = (3\ 4 \mid 8\ 2\ 7\ 1 \mid 6\ 5)$$
$$\mathrm{R}_2 = (4\ 2 \mid 5\ 1\ 6\ 8 \mid 3\ 7)$$
$$\mathrm{D}_1 = (x\ x \mid 8\ 2\ 7\ 1 \mid x\ x)$$
$$\mathrm{D}_2 = (x\ x \mid 5\ 1\ 6\ 8 \mid x\ x)$$

Następnie tworzona jest sekwencja kolejnych wierzchołków dla rodziców. Sekwencja ta zaczyna się od drugiego punktu przecięcia czyli dla $R_1$ jest to $6 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 8 \rightarrow 2 \rightarrow 7 \rightarrow 1$. W kolejnym kroku usuwamy liczby, które występują już w $D_2$. Po usunięciu liczb, sekwencja wygląda w taki sposób: $3 \rightarrow 4 \rightarrow 2 \rightarrow 7$. Tą sekwencje umieszczamy w drugim potomku od miejsca drugiego punktu cięcia. Analogicznie robimy dla $R_2$ i $D_1$ i otrzymujemy:

$$D_1 = (5\ 6 \mid 8\ 2\ 7\ 1 \mid 3\ 4)$$
$$D_2 = (2\ 7 \mid 5\ 1\ 6\ 8 \mid 3\ 4)$$

## 2.6 Teoretyczne oszacowanie złożoności

### 2.6.1 Czasowa złożoność

Algorytm zawiera dwie zagnieżdżone pętle, które wykonują się populationSize·generationNumber razy, w każdej generacji wykonuje się sortowanie wszystkich osobników populacji metodą *std::sort()*, której złożoność wynosi $O(N \cdot log_2(N))$. Liczba osobników, która jest sortowana może wynosić $x = 4$·populationSize + populationCopyNumber (gdzie populationSize > populationCopyNumber), ponieważ w każdej iteracji wewnętrznej pętli tworzonych jest dwóch potomków, a każdy z nich może zmutować co daje maksymalnie 4 osobników na iteracje. Dla selekcji koła ruletki i selekcji rankingowej również wykonywane jest sortowanie populacji oraz wykonywane są trzy pętle związane z zdatnością, z których każda wykonuje się populationSize razy. Następnie wykonywane jest krzyżowanie dwóch osobników. Rozpoczyna się od kopiowania w pętli obszaru przecięcia, która wykonuje się maksymalnie n - 1 razy (gdzie n to rozmiar instancji), potem kopiowany jest obszar, który nie został wcześniej skopiowany. Wykonywane jest to maksymalnie w dwóch pętlach o limicie iteracji nie przekraczającym n. Na sam koniec pętli wewnętrznej wykonywana jest mutacja dla maksymalnie dwóch osobników, którzy mogą wyznaczać mutacje na podstawie przeszukiwania sąsiedztwa w dwóch zagnieżdżonych pętlach wykonywających się maksymalnie (n-3)(n-2) razy. Podsumowując otrzymujemy złożoność:

$$O(\text{generationNumber}(x \log_2 x) + \text{generationNumber populationSize } (x \log_2 x + 3x + (n-1) + 2n + (n-3)(n-2)))$$
$$O(\text{generationNumber} \cdot (x \log_2 x + \text{populationSize}(x \log_2 x + n^2)))$$
$$O(\text{generationNumber} \cdot (\text{populationSize}^2 \cdot log_2(\text{populationSize}) + \text{populationSize } n^2))$$
$$O(\text{generationNumber} \cdot \text{populationSize}(\text{populationSize} \cdot log_2(\text{populationSize}) + n^2))$$

### 2.6.2 Pamięciowa złożoność

Algorytm zawiera mapę, która przechowuje wszystkich osobników populacji 4·populationSize + populationCopyNumber. Każdy osobnik zawiera koszt ścieżki i ścieżkę czyli n + 1. Oraz istnieje wektor zdatności, którzy jest o rozmiarze populacji, więc złożoność pamięciowa wynosi:

$$O((4 \text{ populationSize} + \text{populationCopyNumber}) \cdot (n+1) + (4 \text{ populationSize} + \text{populationCopyNumber}))$$
$$O(\text{populationSize} \cdot n)$$

# 3 Testy zaimplementowanego algorytmu

## 3.1 Poprawność badanego algorytmu

```
Otworzono plik m6.atsp
Wczytano dane
Podaj prawdopodobienstwo
0.6
Podaj rozmiar populacji
50
Podaj liczbe kopiowanych osobnikow
10
Podaj liczbe generacji
1
Podaj rodzaj selekcji
0 - selekcja kola ruletki
1 - selekcja turniejowa
2 - selekcja rankingowa
1
Podaj typ krzyzowania
0 - Partially Mapped Crossover
1 - Order Crossover Operator
1
0 152   90%
4 146   82.5%
7 118   47.5%
3 100   25%
13 98   22.5%
33 80   0%
1->2->3->4->0->5->1
50 80   0%
```

(a) Algorytm genetyczny

```
Otworzono plik m6.atsp
Wczytano dane
Podaj startowy wierzcholek
0
Podaj alfe
0.99
Podaj liczbe epok
100
Podaj liczbe iteracji w epoce
100
Podaj startowa temperature
0
0   159   98.75%
9   118   47.5%
21   116   45%
99   100   25%
149   92   15%
154   85   6.25%
164   80   0%
0->5->1->2->3->4->0
80
```

(b) Symulowane wyżarzanie

Rysunek 1: Badanie poprawności algorytmu dla pliku m6.atsp, porównując otrzymane wyniki z algorytmem symulowanego wyżarzania

```
Otworzono plik m10.atsp
Wczytano dane
Podaj prawdopodobienstwo
0.6
Podaj rozmiar populacji
10
Podaj liczbe kopiowanych osobnikow
5
Podaj liczbe generacji
300
Podaj rodzaj selekcji
0 - selekcja kola ruletki
1 - selekcja turniejowa
2 - selekcja rankingowa
0
Podaj typ krzyzowania
0 - Partially Mapped Crossover
1 - Order Crossover Operator
1
0 462   117.925%
0 408   92.4528%
1 332   56.6038%
12 319   50.4717%
12 270   27.3585%
37 240   13.2075%
91 212   0%
2->4->3->0->5->1->9->6->7->8->2
3000 212   0%
```

(a) Algorytm genetyczny

```
Otworzono plik m10.atsp
Wczytano dane
Podaj startowy wierzcholek
0
Podaj alfe
0.99
Podaj liczbe epok
100
Podaj liczbe iteracji w epoce
1000
Podaj startowa temperature
0
0   576   171.698%
1   549   158.962%
2   511   141.038%
3   475   124.057%
8   426   100.943%
24   388   83.0189%
185   347   63.6792%
253   345   62.7359%
409   317   49.5283%
626   274   29.2453%
790   271   27.8302%
1311   265   25%
7238   244   15.0943%
38952   239   12.7358%
47992   234   10.3774%
53531   219   3.30189%
88997   212   0%
0->3->4->2->8->7->6->9->1->5->0
212
```

(b) Symulowane wyżarzanie

Rysunek 2: Badanie poprawności algorytmu dla pliku m10.atsp, porównując otrzymane wyniki z algorytmem symulowanego wyżarzania

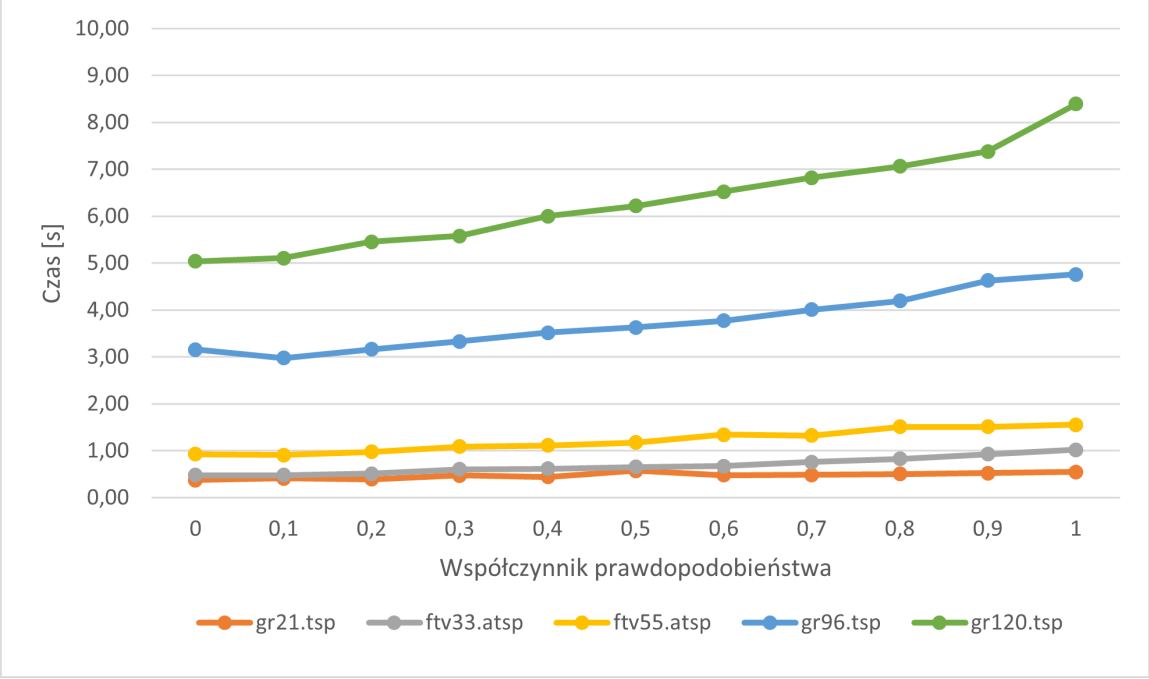
## 3.2 Wpływ poszczególnych parametrów na prace algorytmu

### 3.2.1 Probability

W ramach tej części testów wybrano 5 instancji, na podstawie, których przeprowadzono testy mające na celu pokazanie wpływu na czas i wartość błędu względnego poszczególnych parametrów.

| Prawd. | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| 0 | 0,37 | 0,48 | 0,93 | 3,16 | 5,04 |
| 0,1 | 0,41 | 0,48 | 0,91 | 2,97 | 5,11 |
| 0,2 | 0,39 | 0,51 | 0,98 | 3,16 | 5,46 |
| 0,3 | 0,48 | 0,61 | 1,09 | 3,33 | 5,58 |
| 0,4 | 0,44 | 0,62 | 1,11 | 3,52 | 6,00 |
| 0,5 | 0,57 | 0,65 | 1,18 | 3,63 | 6,22 |
| 0,6 | 0,48 | 0,67 | 1,34 | 3,77 | 6,52 |
| 0,7 | 0,49 | 0,76 | 1,33 | 4,01 | 6,82 |
| 0,8 | 0,51 | 0,83 | 1,51 | 4,19 | 7,07 |
| 0,9 | 0,52 | 0,93 | 1,51 | 4,63 | 7,38 |
| 1 | 0,55 | 1,02 | 1,56 | 4,76 | 8,40 |

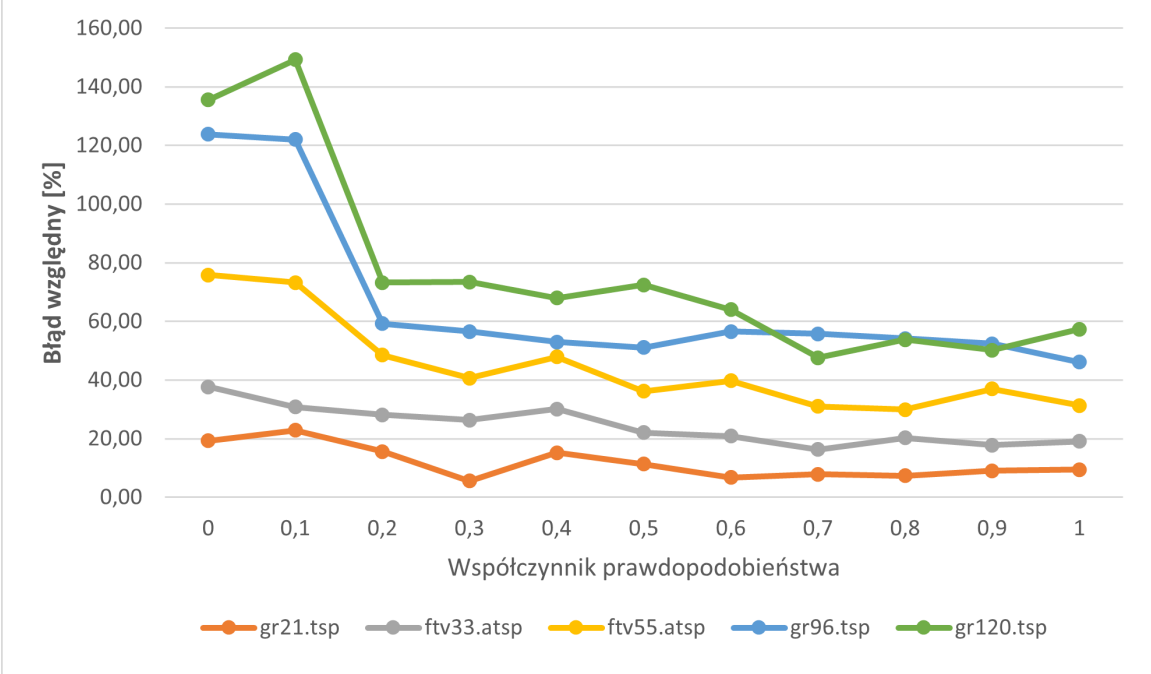
Tabela 1: Tabela wpływu prawdopodobieństwa na czas wykonywania się algorytmu w sekundach




Rysunek 3: Wykres wpływu prawdopodobieństwa na czas wykonywania się algorytmu w zależności od wielkości instancji

| Prawd. | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| 0 | 19,28 | 37,65 | 75,81 | 123,86 | 135,51 |
| 0,1 | 22,91 | 30,78 | 73,22 | 121,99 | 149,26 |
| 0,2 | 15,62 | 28,12 | 48,54 | 59,23 | 73,29 |
| 0,3 | 5,56 | 26,38 | 40,63 | 56,58 | 73,49 |
| 0,4 | 15,18 | 30,14 | 47,91 | 52,95 | 67,95 |
| 0,5 | 11,35 | 22,02 | 36,13 | 51,10 | 72,42 |
| 0,6 | 6,79 | 20,86 | 39,79 | 56,52 | 63,98 |
| 0,7 | 7,86 | 16,27 | 31,00 | 55,80 | 47,56 |
| 0,8 | 7,34 | 20,28 | 29,95 | 54,14 | 53,78 |
| 0,9 | 9,06 | 17,74 | 36,99 | 52,32 | 50,18 |
| 1 | 9,49 | 19,11 | 31,31 | 46,08 | 57,30 |

Tabela 2: Tabela wpływu prawdopodobieństwa na błąd względny wyrażony w procentach

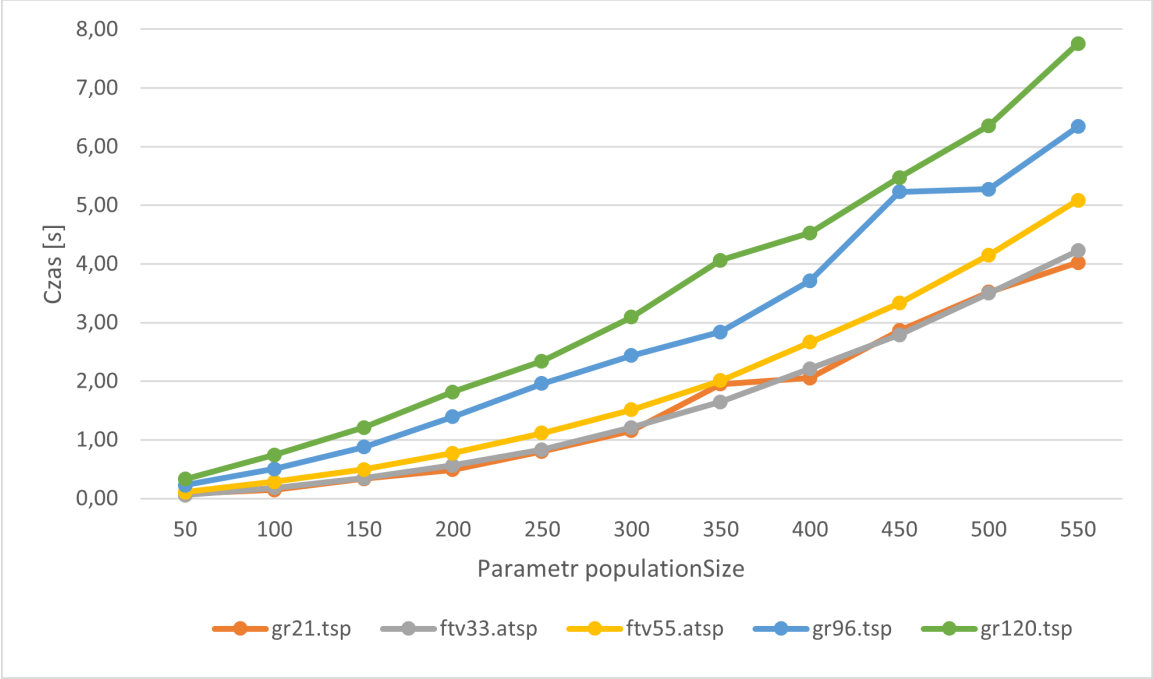



Rysunek 4: Wykres wpływu prawdopodobieństwa na błąd względny wyniku algorytmu w zależności od wielkości instancji

### 3.2.2 PopulationSize

| Populacja | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| 50 | 0,08 | 0,06 | 0,11 | 0,23 | 0,33 |
| 100 | 0,15 | 0,18 | 0,28 | 0,51 | 0,74 |
| 150 | 0,34 | 0,35 | 0,49 | 0,88 | 1,21 |
| 200 | 0,49 | 0,57 | 0,77 | 1,39 | 1,82 |
| 250 | 0,80 | 0,83 | 1,12 | 1,96 | 2,34 |
| 300 | 1,16 | 1,21 | 1,51 | 2,44 | 3,09 |
| 350 | 1,95 | 1,64 | 2,01 | 2,84 | 4,06 |
| 400 | 2,06 | 2,21 | 2,67 | 3,71 | 4,53 |
| 450 | 2,87 | 2,79 | 3,33 | 5,23 | 5,47 |
| 500 | 3,52 | 3,50 | 4,15 | 5,27 | 6,36 |
| 550 | 4,02 | 4,22 | 5,09 | 6,34 | 7,76 |

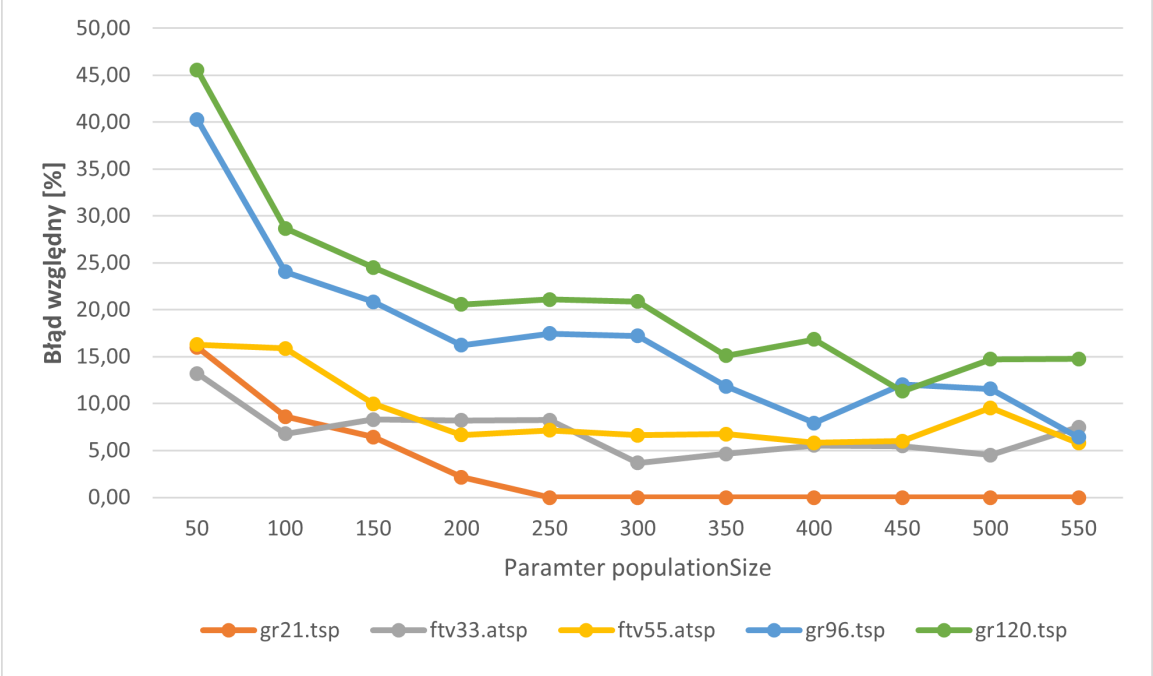
Tabela 3: Tabela wpływu parametru *populationSize* na czas wykonywania się algorytmu w sekundach




Rysunek 5: Wykres wpływu parametru *populationSize* na czas wykonywania się algorytmu w zależności od wielkości instancji

| Populacja | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| 50 | 0,08 | 0,06 | 0,11 | 0,23 | 0,33 |
| 100 | 0,15 | 0,18 | 0,28 | 0,51 | 0,74 |
| 150 | 0,34 | 0,35 | 0,49 | 0,88 | 1,21 |
| 200 | 0,49 | 0,57 | 0,77 | 1,39 | 1,82 |
| 250 | 0,80 | 0,83 | 1,12 | 1,96 | 2,34 |
| 300 | 1,16 | 1,21 | 1,51 | 2,44 | 3,09 |
| 350 | 1,95 | 1,64 | 2,01 | 2,84 | 4,06 |
| 400 | 2,06 | 2,21 | 2,67 | 3,71 | 4,53 |
| 450 | 2,87 | 2,79 | 3,33 | 5,23 | 5,47 |
| 500 | 3,52 | 3,50 | 4,15 | 5,27 | 6,36 |
| 550 | 4,02 | 4,22 | 5,09 | 6,34 | 7,76 |

Tabela 4: Tabela wpływu parametru *populationSize* na błąd względny wyrażony w procentach

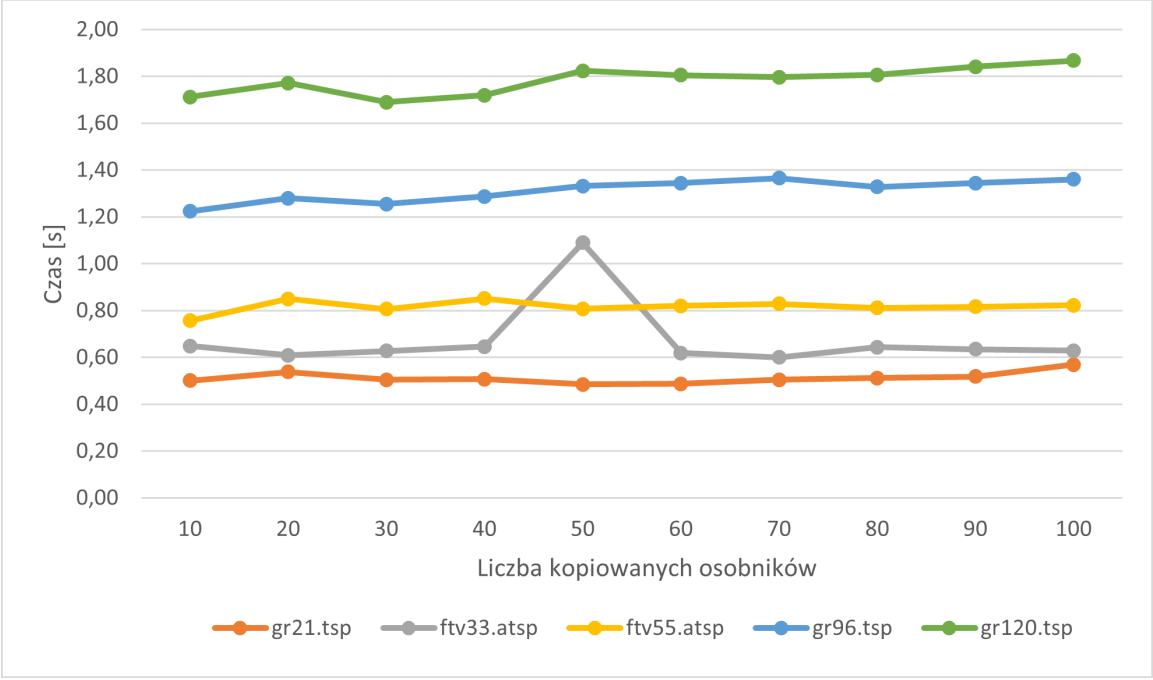



Rysunek 6: Wykres wpływu parametru *populationSize* na błąd względny wyniku algorytmu w zależności od wielkości instancji

### 3.2.3 PopulationCopyNumber

| L. kopiowanych osobników | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| 10 | 0,50 | 0,65 | 0,76 | 1,22 | 1,71 |
| 20 | 0,54 | 0,61 | 0,85 | 1,28 | 1,77 |
| 30 | 0,50 | 0,63 | 0,81 | 1,25 | 1,69 |
| 40 | 0,51 | 0,65 | 0,85 | 1,29 | 1,72 |
| 50 | 0,48 | 1,09 | 0,81 | 1,33 | 1,82 |
| 60 | 0,49 | 0,62 | 0,82 | 1,35 | 1,80 |
| 70 | 0,50 | 0,60 | 0,83 | 1,37 | 1,80 |
| 80 | 0,51 | 0,64 | 0,81 | 1,33 | 1,81 |
| 90 | 0,52 | 0,64 | 0,82 | 1,34 | 1,84 |
| 100 | 0,57 | 0,63 | 0,82 | 1,36 | 1,87 |

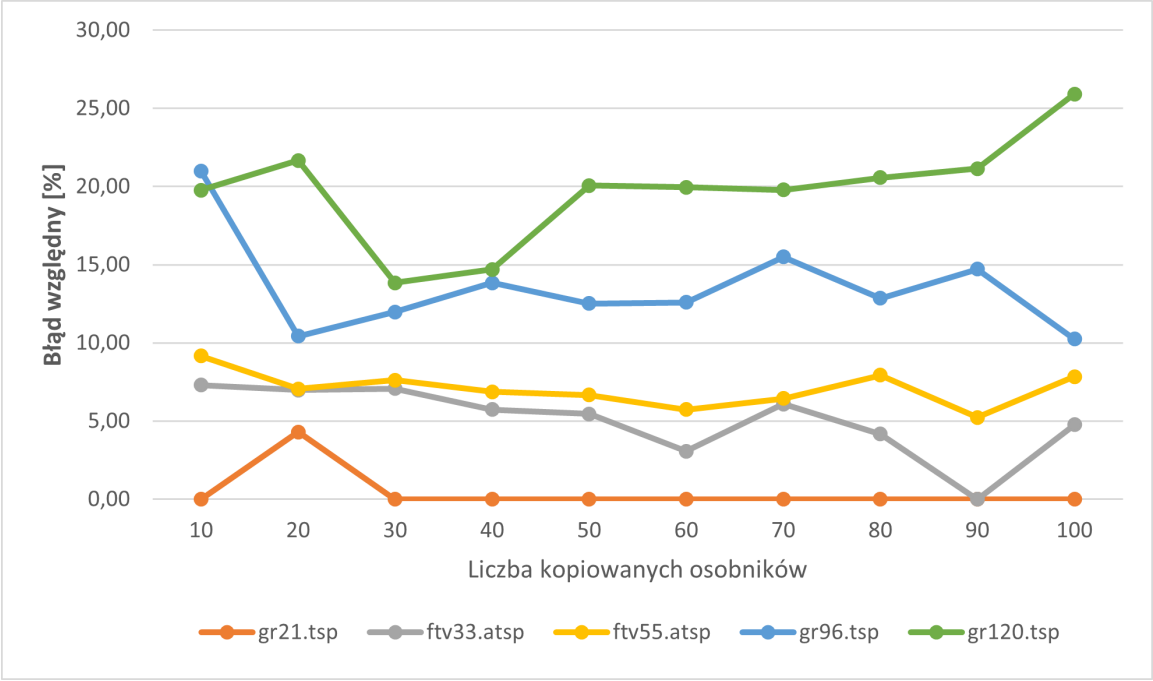
Tabela 5: Tabela wpływu liczby kopiowanych osobników na czas wykonywania się algorytmu w sekundach




Rysunek 7: Wykres wpływu liczby kopiowanych osobników na czas wykonywania się algorytmu w zależności od wielkości instancji

| L. kopiowanych osobników | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| 10 | 0,00 | 7,31 | 9,17 | 20,99 | 19,77 |
| 20 | 4,30 | 6,98 | 7,06 | 10,42 | 21,67 |
| 30 | 0,00 | 7,08 | 7,61 | 11,97 | 13,85 |
| 40 | 0,00 | 5,74 | 6,87 | 13,84 | 14,69 |
| 50 | 0,00 | 5,46 | 6,67 | 12,51 | 20,07 |
| 60 | 0,00 | 3,06 | 5,73 | 12,58 | 19,95 |
| 70 | 0,00 | 6,08 | 6,44 | 15,49 | 19,78 |
| 80 | 0,00 | 4,17 | 7,94 | 12,86 | 20,57 |
| 90 | 0,00 | 0,00 | 5,24 | 14,71 | 21,15 |
| 100 | 0,00 | 4,76 | 7,84 | 10,25 | 25,91 |

Tabela 6: Tabela wpływu liczby kopiowanych osobników na błąd względny wyrażony w procentach

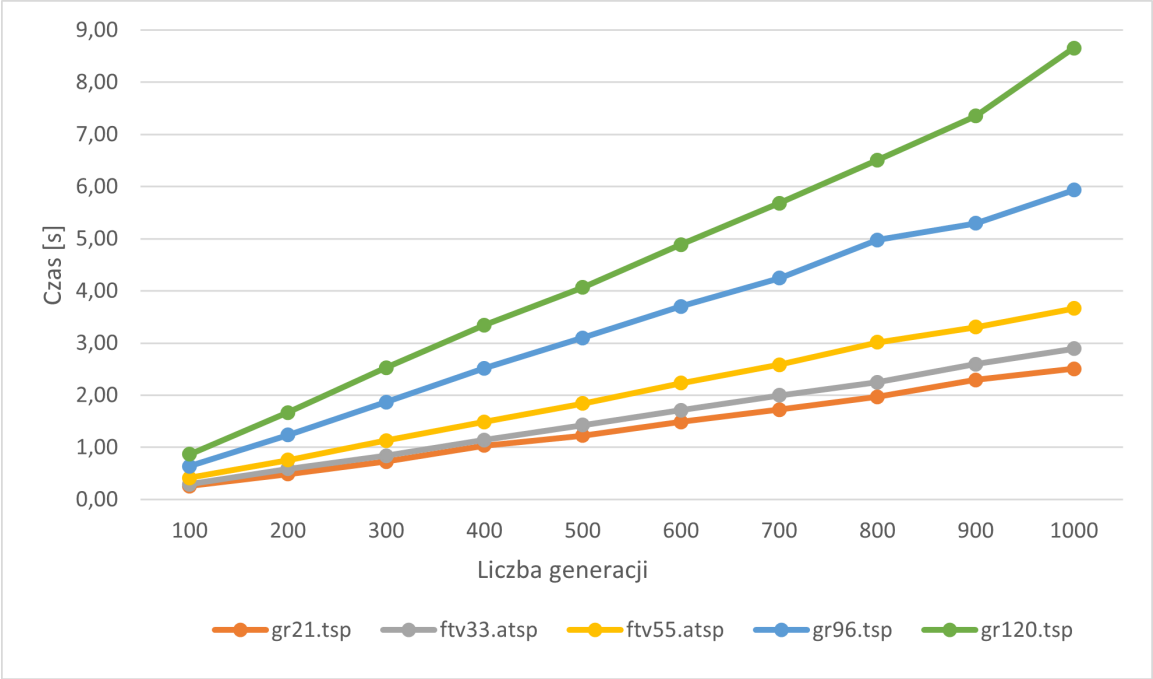



Rysunek 8: Wykres wpływu liczby kopiowanych osobników na błąd względny wyniku algorytmu w zależności od wielkości instancji

### 3.2.4 GenerationNumber

| Liczba generacji | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| 100 | 0,26 | 0,29 | 0,42 | 0,64 | 0,87 |
| 200 | 0,49 | 0,59 | 0,76 | 1,24 | 1,67 |
| 300 | 0,73 | 0,84 | 1,14 | 1,87 | 2,53 |
| 400 | 1,04 | 1,14 | 1,49 | 2,52 | 3,35 |
| 500 | 1,23 | 1,43 | 1,84 | 3,10 | 4,07 |
| 600 | 1,49 | 1,71 | 2,23 | 3,70 | 4,89 |
| 700 | 1,72 | 2,00 | 2,59 | 4,25 | 5,68 |
| 800 | 1,97 | 2,25 | 3,01 | 4,98 | 6,51 |
| 900 | 2,29 | 2,59 | 3,31 | 5,30 | 7,35 |
| 1000 | 2,51 | 2,89 | 3,66 | 5,93 | 8,66 |

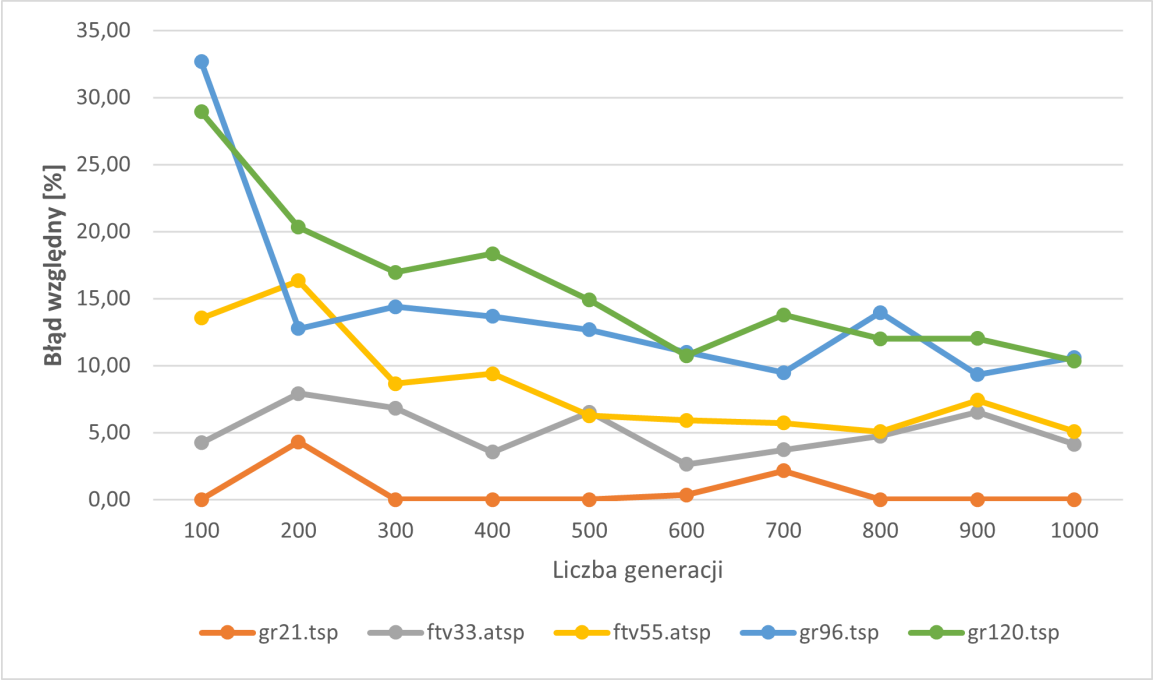
Tabela 7: Tabela wpływu liczby generacji na czas wykonywania się algorytmu w sekundach




Rysunek 9: Wykres wpływu liczby generacji na czas wykonywania się algorytmu w zależności od wielkości instancji

| Liczba generacji | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| 100 | 0,00 | 4,23 | 13,54 | 32,67 | 28,93 |
| 200 | 4,30 | 7,92 | 16,33 | 12,76 | 20,32 |
| 300 | 0,00 | 6,83 | 8,66 | 14,38 | 16,95 |
| 400 | 0,00 | 3,55 | 9,38 | 13,68 | 18,34 |
| 500 | 0,00 | 6,52 | 6,26 | 12,68 | 14,88 |
| 600 | 0,35 | 2,63 | 5,92 | 10,99 | 10,73 |
| 700 | 2,15 | 3,72 | 5,72 | 9,46 | 13,78 |
| 800 | 0,00 | 4,73 | 5,07 | 13,94 | 12,01 |
| 900 | 0,00 | 6,52 | 7,41 | 9,33 | 12,01 |
| 1000 | 0,00 | 4,12 | 5,10 | 10,60 | 10,34 |

Tabela 8: Tabela wpływu liczby generacji na błąd względny wyrażony w procentach

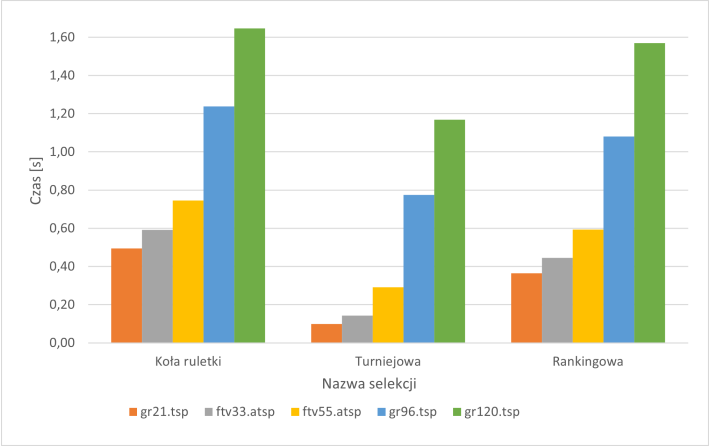



Rysunek 10: Wykres wpływu liczby generacji na błąd względny wyniku algorytmu w zależności od wielkości instancji

### 3.2.5 SelectionType

| Nazwa selekcji | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| Koła ruletki | 0,50 | 0,59 | 0,74 | 1,24 | 1,65 |
| Turniejowa | 0,10 | 0,14 | 0,29 | 0,78 | 1,17 |
| Rankingowa | 0,36 | 0,45 | 0,59 | 1,08 | 1,57 |

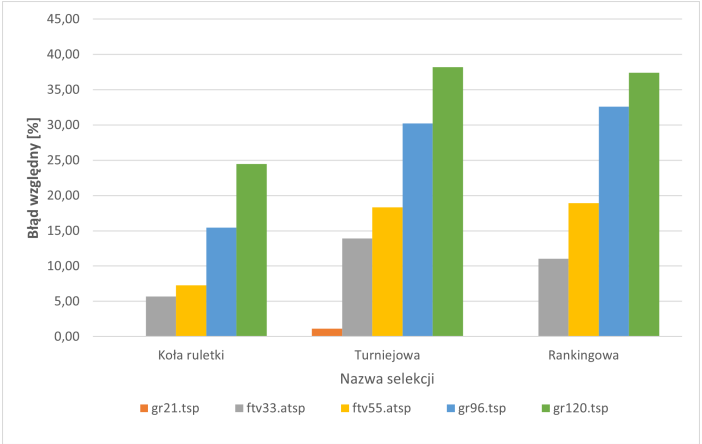
Tabela 9: Tabela wpływu selekcji na czas wykonywania się algorytmu w sekundach




Rysunek 11: Wykres wpływu selekcji na czas wykonywania się algorytmu w zależności od wielkości instancji

| Nazwa selekcji | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| Koła ruletki | 0,00 | 5,69 | 7,26 | 15,46 | 24,47 |
| Turniejowa | 1,11 | 13,92 | 18,35 | 30,20 | 38,19 |
| Rankingowa | 0,00 | 11,06 | 18,89 | 32,61 | 37,38 |

Tabela 10: Tabela wpływu selekcji na błąd względny wyrażony w procentach

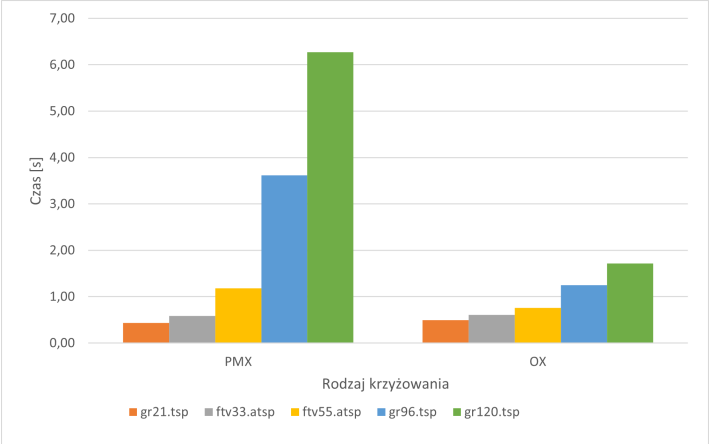



Rysunek 12: Wykres wpływu selekcji na błąd względny wyniku algorytmu w zależności od wielkości instancji

### 3.2.6 CrossoverType

| Rodzaj krzyżowania | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| PMX | 0,43 | 0,58 | 1,18 | 3,61 | 6,27 |
| OX | 0,50 | 0,60 | 0,76 | 1,25 | 1,71 |

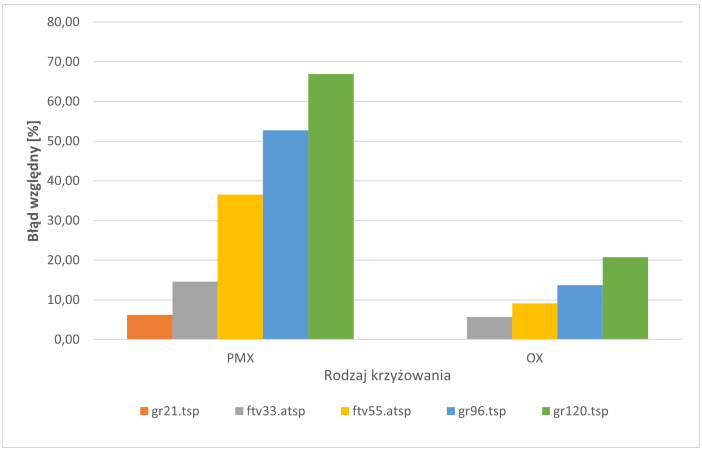
Tabela 11: Tabela wpływu rodzaju krzyżowania na czas wykonywania się algorytmu w sekundach




Rysunek 13: Wykres wpływu rodzaju krzyżowania na czas wykonywania się algorytmu w zależności od wielkości instancji

| Rodzaj krzyżowania | Nazwa instancji | | | | |
|---|---|---|---|---|---|
| | gr21.tsp | ftv33.atsp | ftv55.atsp | gr96.tsp | gr120.tsp |
| PMX | 6,23 | 14,57 | 36,49 | 52,74 | 66,87 |
| OX | 0,00 | 5,68 | 9,13 | 13,74 | 20,73 |

Tabela 12: Tabela wpływu rodzaju krzyżowania na błąd względny wyrażony w procentach




Rysunek 14: Wykres wpływu rodzaju krzyżowania na błąd względny wyniku algorytmu w zależności od wielkości instancji

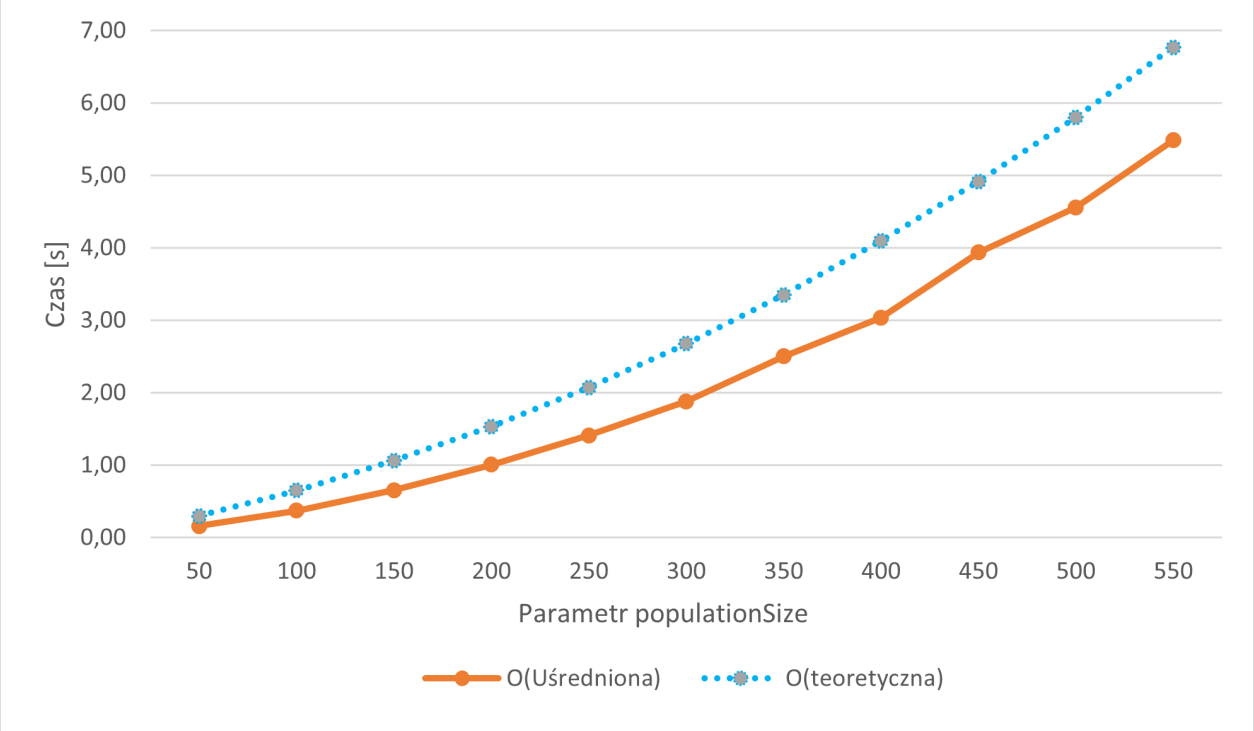
## 3.3 Zbadanie praktycznej złożoności czasowej algorytmu

W celu zbadania praktycznej złożoności przeprowadzone zostały testy dla 5 instancji ("gr21.tsp", "ftv33.atsp", "ftv55.atsp", "gr96.tsp","gr120.tsp"), w których zmieniano za każdym razem tylko jeden parametr, od którego była zależna złożoność teoretyczna zapisana wzorem znajdującym się w rozdziale "Czasowa złożoność". Następnie wyniki z 5 instancji uśredniono dla poszczególnych wartości parametrów i zapisano w poniższych tabelach.

### 3.3.1 Zależność od *populationSize*

| Złożoność | Parametr *populationSize* | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| czasowa | 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 | 500 | 550 |
| $O(sr)$ | 0,16 | 0,37 | 0,65 | 1,01 | 1,41 | 1,88 | 2,50 | 3,03 | 3,94 | 4,56 | 5,49 |
| $O(t)$ | 0,04 | 0,18 | 0,43 | 0,80 | 1,29 | 1,90 | 2,65 | 3,52 | 4,54 | 5,68 | 6,97 |

Tabela 13: Tabela złożoności czasowej dla zmiennego parametru *populationSize*

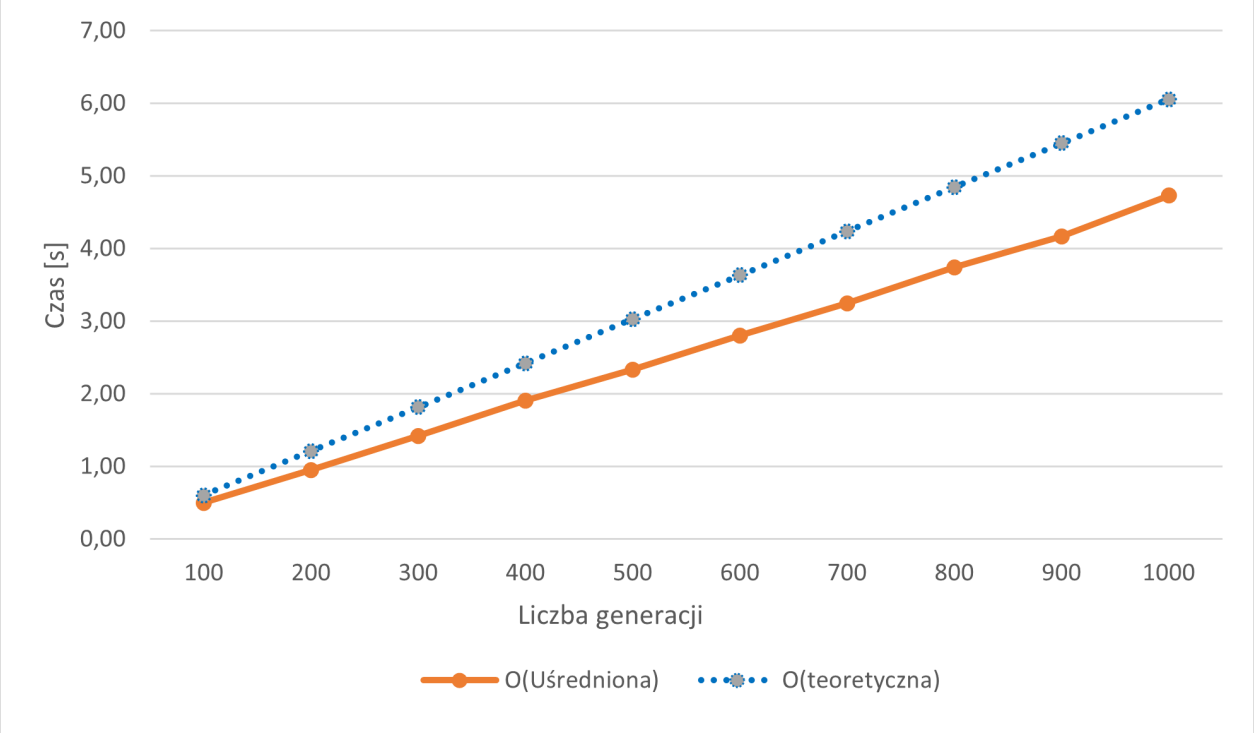



Rysunek 15: Wykres złożoności uśrednionej i teoretycznej dla zmiennego parametru *populationSize*

### 3.3.2 Zależność od *generationNumber*

| Złożoność | Liczba generacji | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| czasowa | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
| $O(sr)$ | 0,50 | 0,95 | 1,42 | 1,91 | 2,33 | 2,81 | 3,25 | 3,74 | 4,17 | 4,73 |
| $O(t)$ | 0,61 | 1,21 | 1,82 | 2,42 | 3,03 | 3,63 | 4,24 | 4,85 | 5,45 | 6,06 |

Tabela 14: Tabela złożoności czasowej dla zmiennej liczby generacji



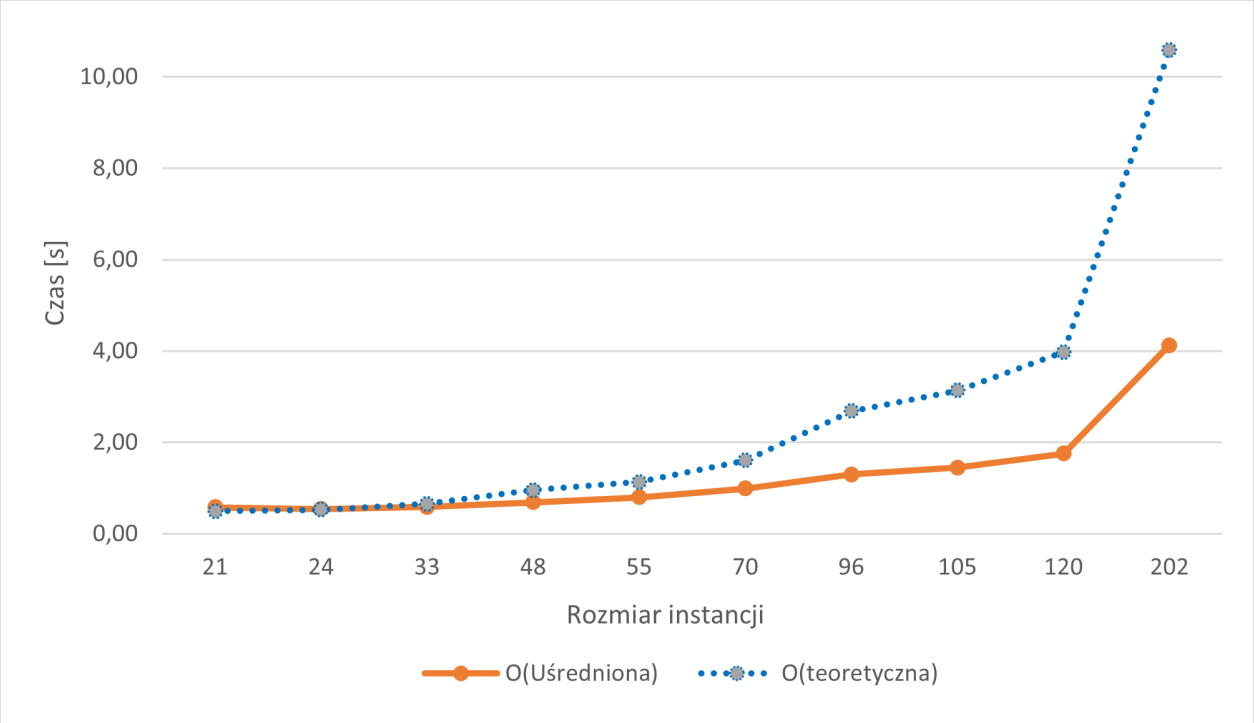

Rysunek 16: Wykres złożoności uśrednionej i teoretycznej dla zmiennej liczby generacji

### 3.3.3 Zależność od $n$

W przypadku badania praktycznej złożoności dla zależności od parametru $n$, wybrano 10 różnych rozmiarów instancji i przeprowadzono po kilka testów czasowych dla algorytmu inicjując go tymi samymi parametrami dla każdej instancji.

| Złożoność czasowa | Rozmiar instancji | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 21 | 24 | 33 | 48 | 55 | 70 | 96 | 105 | 120 | 202 |
| $O(sr)$ | 0,58 | 0,55 | 0,59 | 0,69 | 0,80 | 0,99 | 1,30 | 1,45 | 1,76 | 4,13 |
| $O(t)$ | 0,49 | 0,53 | 0,65 | 0,96 | 1,14 | 1,61 | 2,69 | 3,14 | 3,98 | 10,58 |

Tabela 15: Tabela złożoności czasowej dla zmiennej wielkości instancji




Rysunek 17: Wykres złożoności uśrednionej i teoretycznej dla zmiennej wielkości instancji

# 4 Wnioski

Algorytm genetyczny dzięki swojej inspiracji naturalnym procesem ewolucji biologicznej jest w stanie rozwiązywać problemy optymalizacyjne, bo tak jak natura stara się ewoluować i szukać sposobu do znalezienia jak najlepszego rozwiązania. Analizując otrzymane wyniki dla *GA* można dojść do wniosku, że algorytm ten umożliwia znalezienie w bardzo dobrym czasie rozwiązania, ale zazwyczaj jest to rozwiązanie obarczone błędem. Dlatego też aby zmniejszyć błąd względny otrzymywanych rozwiązań, wprowadzano przeszukiwanie lokalnego sąsiedztwa rozwiązań, metoda ta odróżnia algorytm genetyczny od algorytmu memetycznego. Jak można zauważyć na wykresach wpływu parametru *probability* odpowiedzialnego za częstotliwość występowania tego przeszukiwania, można stwierdzić, że zmniejsza w każdym przypadku błąd rozwiązania przy nieznacznym wpływie na czas wykonywania się algorytmu. Dla prawdopodobieństwa równego 0.1 występuje pogorszenie wyniku, jest to spowodowane tym, że wykonuje się wtedy tylko metoda odpowiedzialna za losowe wybieranie genów do mutacji, co sprawia, iż w populacji pojawiają się osobniki, które mogą zawierać gorsze rozwiązania. W przypadku badania wpływu parametrów *populationSize* oraz *generationNumber* sytuacja jest trochę inna. Parametry te zwiększają liniowo czas wykonywanie się algorytmu i zmniejszają o trochę błąd wraz z wzrostem ich wartości. Wpływ *populationCopyNumber* na działanie algorytmu jest znikomy, wykresy czasu są wykresami stałymi, a zmiana wartości błędu występuje tylko dla pewnych losowych przypadków.

W sprawie typu selekcji rodziców zaobserwować można, że najszybszą metodą wyboru jest selekcja turniejowa, wynika to z faktu, że w jej przypadku nie potrzeba sortować populacji, tak jak w dwóch pozostałych metodach. Ale za to selekcją, która znajduje rozwiązania o najmniejszym błędzie względnym wyniku jest selekcja koła ruletki. Metoda selekcji, która najgorzej wpływa na pracę algorytmu jest selekcja rankingowa. Uzyskuje ona najgorsze czasy i wyniki spośród pozostałych metod.

W odniesieniu do tabeli i wykresów przedstawiających uzyskane wyniki dla rodzajów krzyżowania, można zauważyć, że zdecydowanie lepszą metodą jest *OX*, ponieważ uzyskuje lepsze wyniki w lepszym czasie niż *PMX*.

Jeśli mowa o wynikach badań złożoności spostrzec można, że uzyskane wyniki praktycznej złożoności czasowej są zbliżone do oszacowanej złożoności teoretycznej wynoszącej:

$$O(\text{generationNumber} \cdot \text{populationSize}(\text{populationSize} \cdot log_2(\text{populationSize}) + n^2)).$$

Natomiast jeśli mowa o pamięciowej złożoności algorytmu, wynosi najprawdopodobniej $O(\text{populationSize} \cdot n)$, ponieważ nie została ona zbadana, a jedynie została oszacowana na podstawie analizy kodu algorytmu.

Podsumowując, algorytm genetyczny jest dobrym algorytmem do rozwiązywania problemu komiwojażera, w momencie, w którym zależy na czasie i rozwiązaniu, które nie będzie rozwiązaniem optymalnym, więc będzie obarczone pewnym błędem, który można zmniejszyć za pomocą odpowiedniego dobrania parametrów kosztem czasu wykonywania się algorytmu, albo wzbogacając *GA* o metodę działania algorytmu memytycznego.