Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Czwartek 17:25*
Temat *Algorytmy populacyjne*	**Problem** *TSP*
Skład grupy *241406 Krzysztof Jopek*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *28 stycznia 2020*

1 Opis problemu

Problem komiwojażera jest problemem optymalizacyjnym, polegającym na odnalezieniu minimalnego cyklu Hamiltona w grafie pełnym ważonym. Nazwa pochodzi od typowej ilustracji problemu: dane jest n miast, które muszą zostać odwiedzone przez komiwojażera oraz odległości między nimi. Zadaniem jest znalezienie najkrótszej drogi przechodzącej przez wszystkie miasta oraz wracającej do miasta początkowego. Problem komiwojażera jest klasy NP-trudny. W realizacji trzeciego etapu projektu został wykorzystany algorytm genetyczny.

2 Metoda rozwiązania

2.1 Algorytm genetyczny

Algorytm genetyczny należy do klasy algorytmów probabilistycznych. Są w nim stosowane mechanizmy selekcji, reprodukcji oraz mutacji, których inspiracją jest biologiczny proces ewolucji. W opisywanym algorytmie zachodzą procesy wzorowane na tych z biologii – tylko najlepiej przystosowane osobniki, o najlepszych genach, mają szansę przetrwać i mieć potomstwo, które będzie lepsze od nich samych. W algorytmie stosowana jest selekcja wstępna rodziców na podstawie selekcji proporcjonalnej, a następnie, ze stałym prawdopodobieństwem [0.9] tworzone są dzieci poprzez krzyżowanie Ox oraz również z ustalonym prawdopodobienstwem [0.1] poprzez jedną z trzech mutacji (swap, invert, scramble). Każdy cykl wiąże się z pełną wymianą pokoleń, czyli wyprodukowaniem takiej samej ilości potomków co początkowa populacja.
W procesie optymalizacji przeszukiwane są przestrzenie potencjalnych rozwiązań dla danego problemu w celu znalezienia najlepszego rozwiązania. Wcześniej wspomniana mutacja polega na tym, że zależnie od wartości prawdopodobieństwa może dojść do losowych zmian wartości w kodzie. Im większa wartość parametru tym więcej mutacji.

Listing 1: Algorytm genetyczny - mutacja invert

```
1  //Funkcja pomocnicza do mutacji invert
2  void Genetic::invert(int i, int j, vector<int>& wektor) {
3      int tempi;
4      for (; i < j; i++, j--) {
5          tempi = wektor[i];
6          wektor[i] = wektor[j];
7          wektor[j] = tempi;
8      }
9  }
10 void Genetic::inversionMutation(vector<int>& wektor) {
11     int i, j;
12     do {
13         i = rand() % size;
14         j = rand() % size;
15     } while (i == j);
16     if (i <= j) {
17         invert(i, j, wektor);
18     }
19     else {
20         invert(j, i, wektor);
21     }
22 }
```

Listing 2: Algorytm genetyczny - mutacja scramble

```
void Genetic::scrambleMutation(vector<int>& wektor) {
    vector<int> pozycje (wektor.size());
    int l = pozycje.size();
    //kopiowanie osobnika do mutacji
    for (int i = 0; i < wektor.size(); i++) {
        pozycje[i] = wektor[i];
    }
    for (int k = 0; k < 5; k++) {
        int r1 = rand() % size;
        int r2 = rand() % size;
        //upewniamy sie, ze r1 jest wieksze od r2
        while(r1 >= r2){ r1 = rand() % size; r2 = rand() % size;}
        //scrambling
        for (int i = 0; i < 10; i++) {
            int i1 = rand() % (r2 + 1 - r1 + 1) + r1;
            int i2 = rand() % (r2 + 1 - r1 + 1) + r1;
            int a = pozycje[i1];
            pozycje[i1] = pozycje[i2];
            pozycje[i2] = a;
        }
    }
    //przepisywanie do osobnika zmutowanej wersji
    for (int i = 0; i < wektor.size(); i++) {
        wektor[i] = pozycje[i];
    }
}
```

Krzyżowanie polega z kolei na wymianie fragmentów kodu pomiędzy rodzicami. W wyniku tego procesu tworzy się potomstwo. Zastosowano krzyżowanie Ox, gdzie przedział dziedziczonej części w pierwszym etapie jest generowany losowo.

Listing 3: Algorytm genetyczny - krzyżowanie Ox

```
vector<int> Genetic::krzyzowanieOx(vector<int> pierwszyWektor,
vector<int> drugiWector) {
    int odkad = rand() % size;
    int dokad = rand() % size;
    vector<int> wynik(size);
    vector<int> czyBylo(size);
    for (int i = 0; i < size; i++) {
        czyBylo[i] = 0;
    }

    for (int i = odkad; i != dokad; i = (i + 1) % size) {
        wynik[i] = pierwszyWektor[i];
        czyBylo[pierwszyWektor[i]] = 1; // wpisuje wartosc
        //1 w miejsce numerow genow, kt re wzialem z pierwszego wektora
    }
    int gdzie = (dokad) % size;
    for (int i = 0; i < size; i++) {
        if (czyBylo[drugiWector[(dokad + i) % size]] == 0) {
```

```
19              //tam gdzie jest zero przepisuje wartosci z drugiego wektora
20              wynik[gdzie] = drugiWector[(dokad + i) % size];
21              gdzie = (gdzie + 1) % size;
22          }
23      }
24      return wynik;
```

W zaimplementowanym algorytmie reprodukcja polega na wybraniu najlepszych rozwiązań ze zbioru rozwiązań potomnych i aktualnych. Ilość tych rozwiązań jest równa początkowej wielkości populacji. Wykorzystywana jest tutaj funkcja przystosowania (f(x)). Aby zabezpieczyć przed utratą różnorodności genetycznej populacji zastosowano mutacje wywołujące sporadyczne zmiany w chromosomach osobników.

2.1.1 Schemat działania oraz kod głównej funkcji algorytmu

Schemat działania algorytmu:
1. Numer populacji t=0.
2. Wygeneruj początkową populację P(t).
3. Dla każdego osobnika populacji wylicz wartość f(x).
4. Wybierz osobniki o najlepszych wskaźnikach f(x).
5. Utwórz nową populację P(t+1) na podstawie mutacji i krzyżowania wybranych osobników z prawdopodobieństwem Pm oraz Pk.
6. Idź do punktu nr 3, dopóki nie spełniony jest warunek stopu.

Listing 4: Algorytm genetyczny - główna funkcja

```
1  vector<int> Genetic::geneticAlg(int liczbaOsobnikow,
2  int liczbaNajlepszychOsobnik w ,float jakCzestoMutacja,
3  float jakCzestoKrzyzowanie, int mutation) {
4
5      vector<int>* osobniki;
6      osobniki = new vector<int>[liczbaOsobnikow];
7      vector<int>* noweOsobniki;
8      noweOsobniki = new vector<int>[liczbaOsobnikow];
9      vector<int> najlepszyOsobnik;
10
11     // przygotowanie macierzy - pierwszy losowy zbior populacji
12     for (int i = 0; i < liczbaOsobnikow; i++) {
13
14         osobniki[i].resize(size);
15         noweOsobniki[i].resize(size);
16         randomPerm(osobniki[i]);
17     }
18     // tworzymy struktur  pary, kt ra wskazuje funckj  celu
19     // osobnika oraz jego indeks
20     pair <double, int>* rozwiazanie;
21     rozwiazanie = new pair <double, int>[liczbaOsobnikow];
22
23     int pokolenie = 10; //liczba pokolen po ktorych zakonczy sie algorytm
24     double ostatni = INT_MAX;
25
26     //g  wna p tla
```

```
27      while (pokolenie > 0) {
28
29          //przypisywanie kosztu i numeru danego osobnika
30          for (int i = 0; i < liczbaOsobnikow; i++) {
31              rozwiazanie[i].first = road(osobniki[i]);
32              rozwiazanie[i].second = i;
33          }
34
35          //sortowanie rosn ce
36          sort(rozwiazanie, rozwiazanie + liczbaOsobnikow);
37
38          // warunek stopu
39          if (ostatni > rozwiazanie[0].first) {
40              ostatni = rozwiazanie[0].first;
41              pokolenie = 10;
42              // przypisanie najlepszego osobnika o najni szym koszcie
43              najlepszyOsobnik = osobniki[rozwiazanie[0].second];
44          }
45          else {
46              pokolenie--;
47          }
48
49          // krzyzowanie i mutowanie osobnikow
50          for (int i = 0; i < liczbaOsobnikow; i++) {
51              // je eli prawdopodobienstwo jest dobre,
52              //to nastepuje krzyzowanie
53              if (float(rand()) / RAND_MAX < jakCzestoKrzyzowanie) {
54                  // wybranie dw ch najlepszych osobnik w do krzy owania
55                  noweOsobniki[i] =
56                  krzyzowanieOx(osobniki[rozwiazanie
57                  [rand() % liczbaNajlepszychOsobnik w].second],
58                  osobniki[rozwiazanie[rand() %
59                  liczbaNajlepszychOsobnik w].second]);
60              }
61              else {
62                  noweOsobniki[i] = osobniki[i];
63              }
64              // je eli prawdopodobie stwo jest dobre, to nastepuje mutacja
65              if (float(rand()) / RAND_MAX < jakCzestoMutacja) {
66                  if (mutation = 1) {
67                      swapMutation(noweOsobniki[i]);
68                  }
69                  if (mutation == 2) {
70                      inversionMutation(noweOsobniki[i]);
71                  }
72                  if (mutation == 3) {
73                      scrambleMutation(noweOsobniki[i]);
74                  }
75              }
76          }
```

```
77          vector <int>* temp = osobniki;
78          osobniki = noweOsobniki; //wrzucenie nowego pokolenia
79          //po zmianach genetycznych
80          noweOsobniki = temp;
81      }
82      if (testy == true) {
83          cout << "Koszt: " << rozwiazanie[0].first << endl;
84      }
85      delete[] osobniki;
86      delete[] noweOsobniki;
87      delete[] rozwiazanie;
88      return najlepszyOsobnik;
89  }
```

3 Eksperymenty obliczeniowe

Obliczenia zastały wykonane na laptopie z procesorem i7-6700HQ, kartą graficzną NVIDIA GeForce GTX 960M, 8GB RAM i DYSK SSD. Jako miarę jakości algorytmu przyjęto średnie procentowe odchylenie (Percentage Relative Deviation, PRD) najlepszego otrzymanego rozwiązania π względem rozwiązania referencyjnego π^{ref}:

$$PRD(\pi) = 100\%(C_{max}(\pi) - C_{max}(\pi^{ref}))/C_{max}(\pi^{ref}) \tag{1}$$

Wszystkie wyniki oraz wykresy zebrano i przedstawiono na rysunkach na końcu sprawozdania, gdzie:

- $PRD_{GA}(\%)$ - średnie procentowe odchylenie dla algorytmu genetycznego,
- π - najlepsze możliwe rozwiązanie (optimum) dla algorytmu genetycznego,
- π^{ref} - rozwiązanie referencyjne dla algorytmu genetycznego,
- n - rozmiar instancji,
- t - czas wykonywania algorytmu,
- $P_k = 0.9$ - prawdopodobieństwo krzyżowania,
- $P_m = 0.1$ - prawdopodobieństwo mutacji,
- $p = 10$ - liczba populacji

4 Wnioski

Zaimplementowany algorytm genetyczny, którego wyniki są zaprezentowane poniżej, cechuje się szybkim czasem działania dla stosunkowo małej wielkości populacji początkowej. Czas działania algorytmu uzależniony jest w dużym stopniu uzależniony od wielkości instancji jaka została użyta do testów oraz liczby potomków w populacji. Im większa instancja oraz większa populacja tym dłuszy czas wykonywania algorytmu.
Podczas testów zauważono znaczący wzrost jakości algorytmu, gdy podwyższono liczbę osobników o drugą potęgę rozmiaru instancji (n^2). Widać również poprawę jakości, gdy zwiększono procent wybranych najlepszych osobników. Wynik niestety jest jednak czasami dosyć trudny do przewidzenia ze względu na to, że dużą wagę w wyborze parametrów odgrywa w nim prawdopodobieństwo, co wpływa na pewną losowość zaprezentowanego problemu.

Literatura

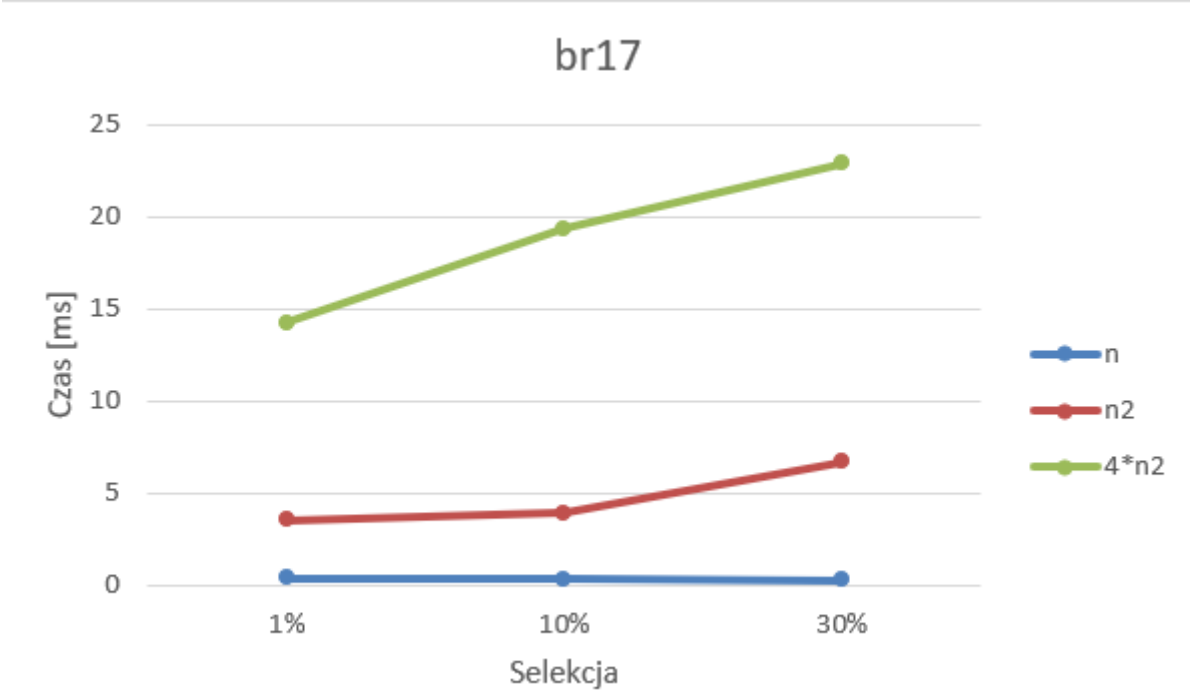
[1] 4. klasyczny algorytm genetyczny. https://www.math.uni.lodz.pl/ marta/$2012_2013_z/zz/gen.pdf$. *Accessed* : $2019-01-26$.

[2] Algorytmy genetyczne (ag). http://www.zio.iiar.pwr.wroc.pl/pea/$w9_g a_t sp.pdf$. *Accessed* : $2019-12-19$.

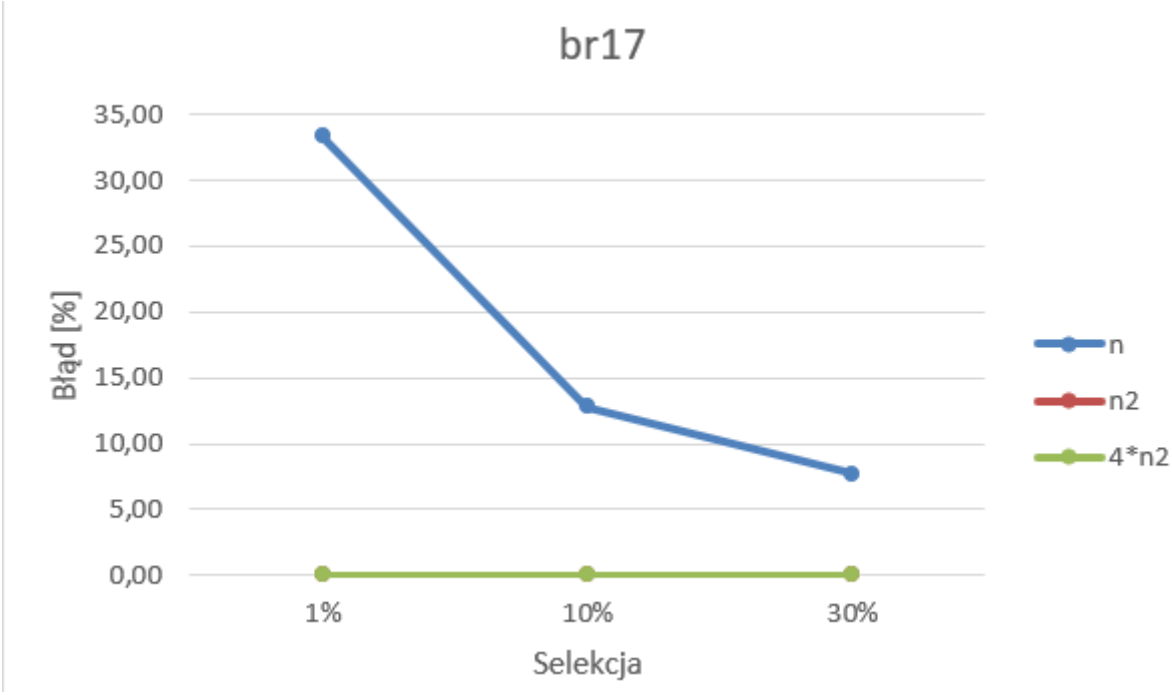
[3] Algorytmy genetyczne dla problemu komiwojażera. http://aragorn.pb.bialystok.pl/ wkwedlo/EA5.pdf. Accessed: 2019-01-26.

Mutacja Swap	$\pi = 39$		
% najlepszych	1%	10%	30%
l. osobników	t		
n	0,39	0,36	0,34
n^2	3,56	3,93	6,69
$4*n^2$	14,24	19,36	22,86
	π^{ref}		
n	52,00	44,00	42,00
n^2	39,00	39,00	39,00
$4*n^2$	39,00	39,00	39,00
	$PRD_{GA}[\%]$		
n	33,33	12,82	7,69
n^2	0,00	0,00	0,00
$4*n^2$	0,00	0,00	0,00

Rysunek 1: Pomiary dla pliku br17.atsp



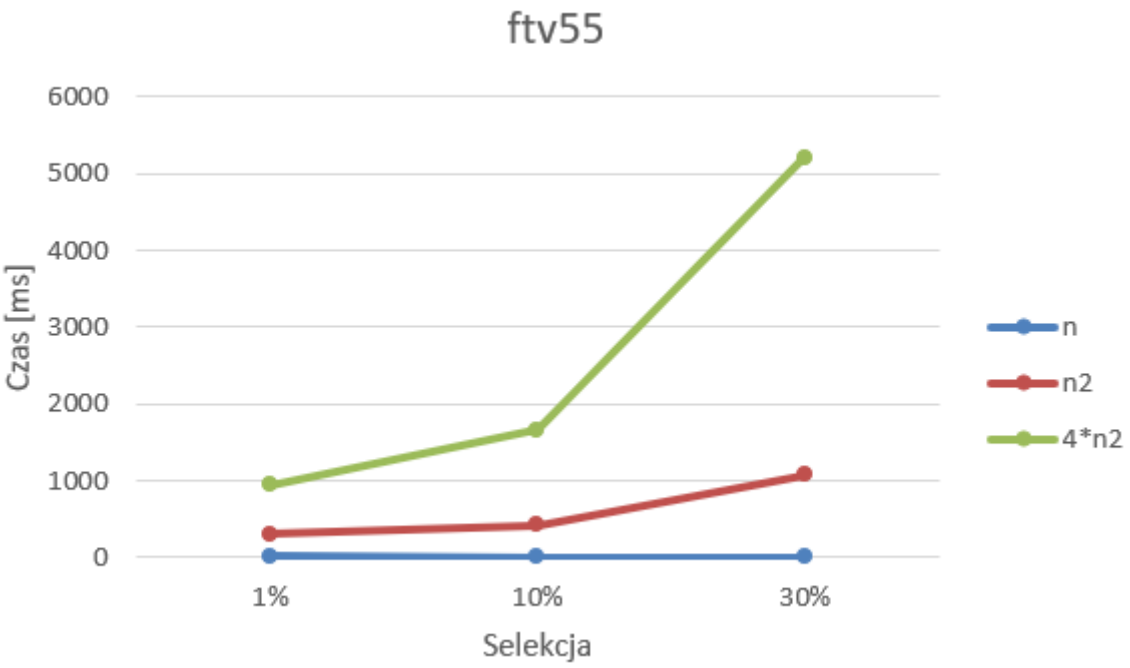
Rysunek 2: Wykres czasu dla pliku br17.atsp



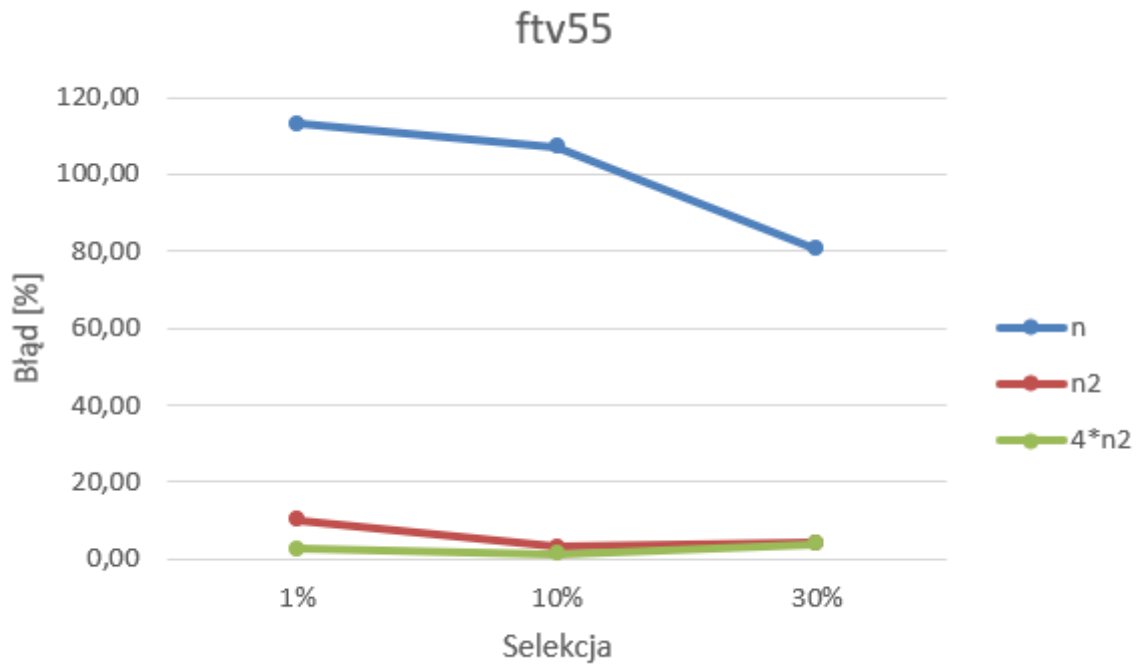
Rysunek 3: Wykres błędu dla pliku br17.atsp

Mutacja Swap	π = 1608		
% najlepszych	1%	10%	30%
l. osobników	t		
n	13,27	10,64	9,23
n^2	310,57	417,94	1077,19
$4*n^2$	942,73	1665,21	5205,55
	π^{ref}		
n	3428,00	3330,00	2903,00
n^2	1770,00	1657,00	1674,00
$4*n^2$	1651,00	1628,00	1670,00
	PRD_{GA}[%]		
n	113,18	107,09	80,53
n^2	10,07	3,05	4,10
$4*n^2$	2,67	1,24	3,86

Rysunek 4: Pomiary dla pliku ftv55.atsp



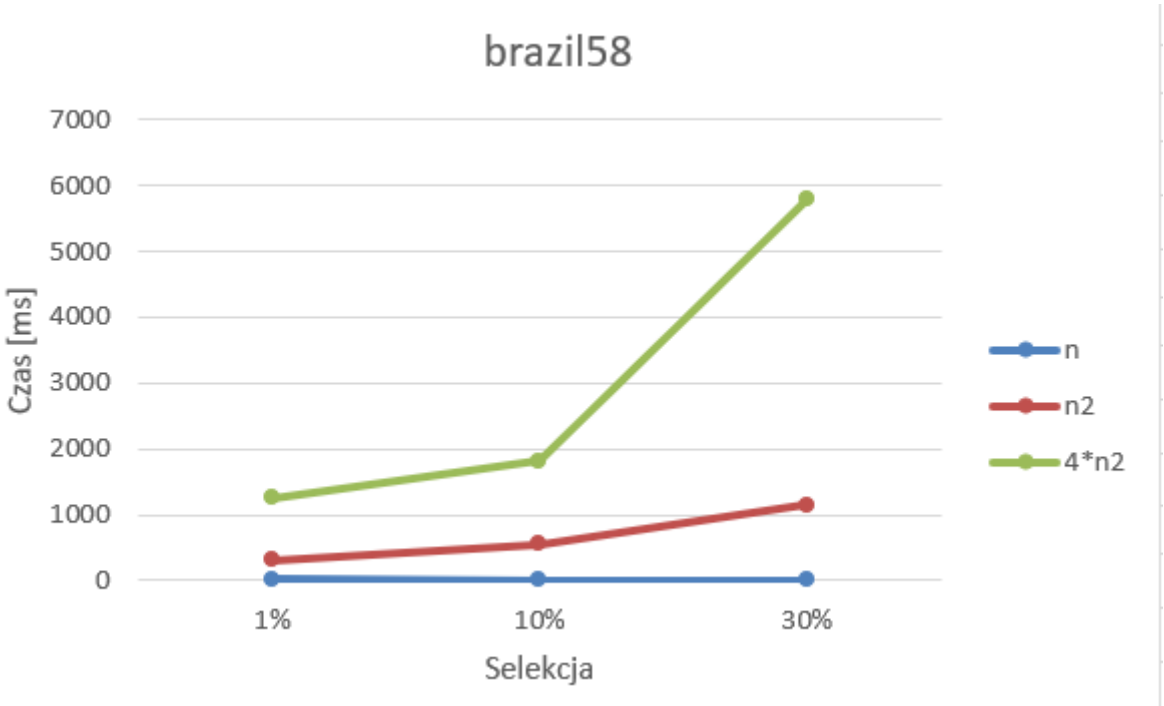
Rysunek 5: Wykres czasu dla pliku ftv55.atsp



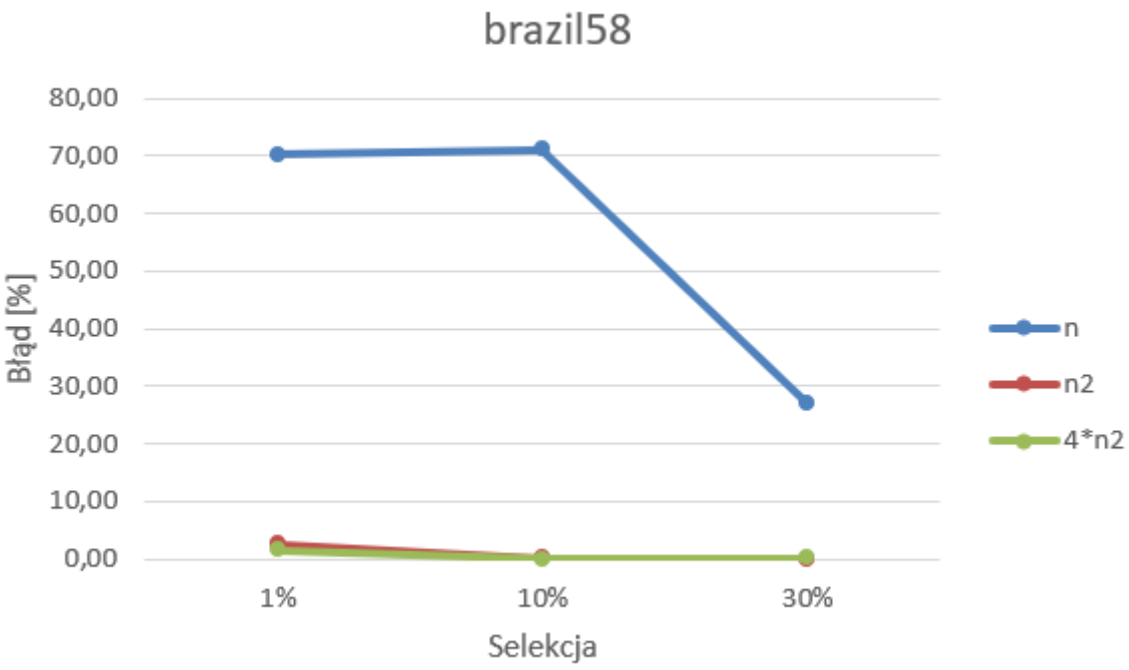
Rysunek 6: Wykres błędu dla pliku ftv55.atsp

Mutacja Scramble	π = 25395		
% najlepszych	1%	10%	30%
l. osobników	t		
n	19,91	14,39	13,82
n^2	317,88	559,00	1153,91
$4*n^2$	1250,56	1823,42	5790,44
	π^{ref}		
n	43243,00	43446,00	32225,00
n^2	26054,00	25445,00	25395,00
$4*n^2$	25813,00	25395,00	25445,00
	PRD_{GA}[%]		
n	70,28	71,08	26,90
n^2	2,59	0,20	0,00
$4*n^2$	1,65	0,00	0,20

Rysunek 7: Pomiary dla pliku brazil58.tsp



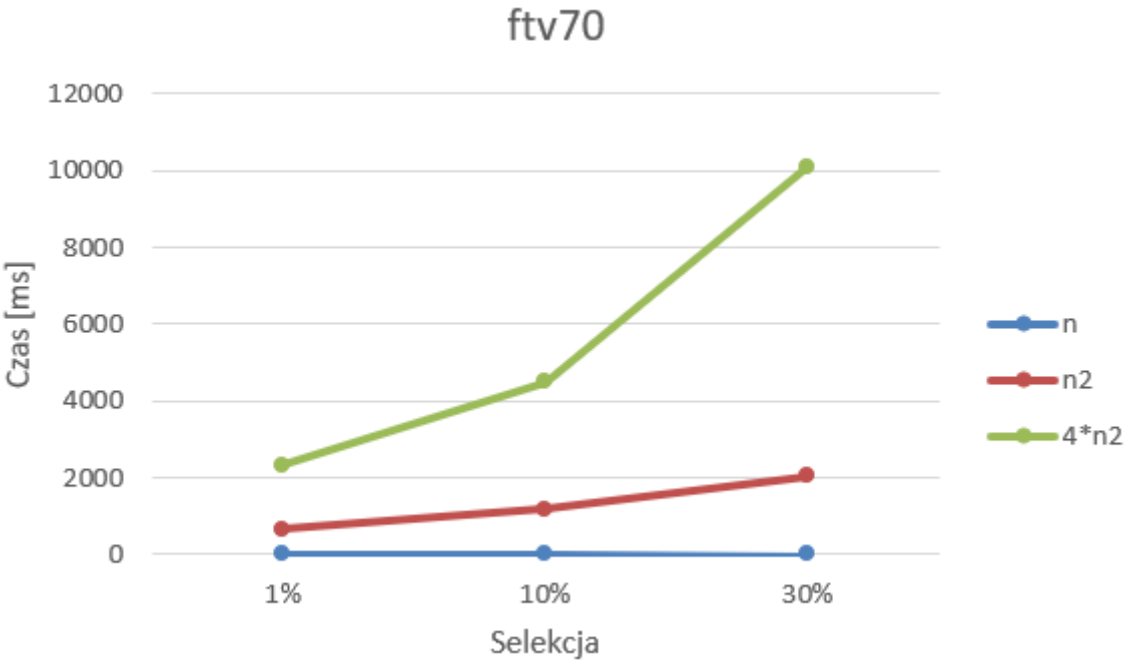
Rysunek 8: Wykres czasu dla pliku brazil58.tsp



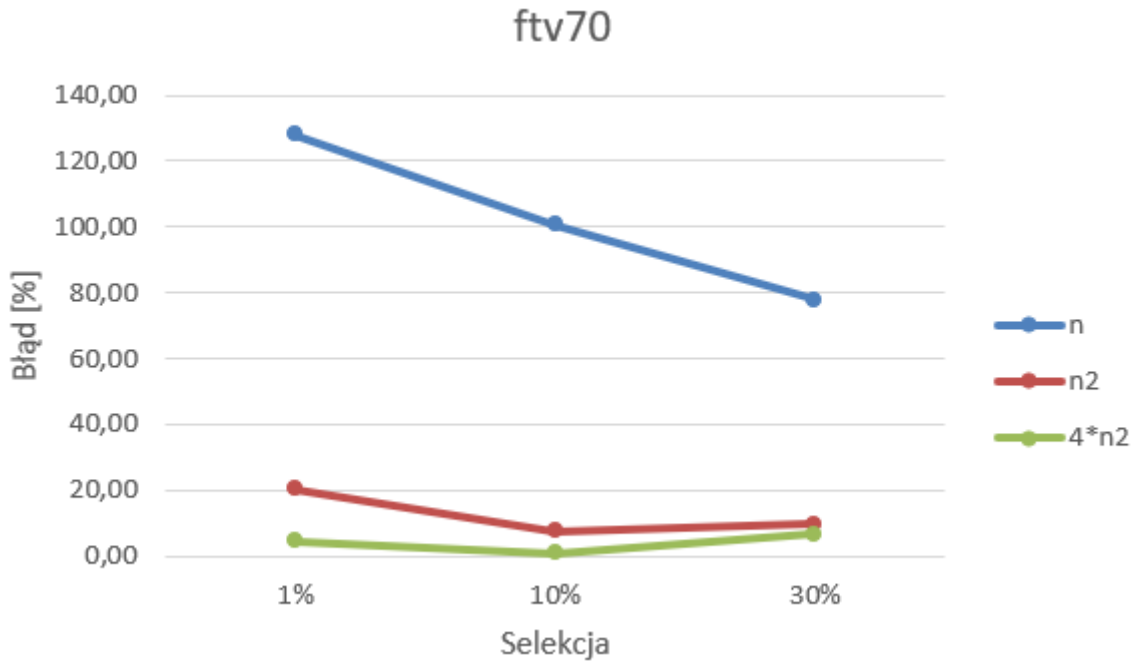
Rysunek 9: Wykres błędu dla pliku brazil58.tsp

Mutacja Invert	π = 1950		
% najlepszych	1%	10%	30%
l. osobników	t		
n	21,48	17,25	14,08
n^2	671,68	1180,83	2044,01
$4*n^2$	2326,40	4488,14	10089,70
	π^{ref}		
n	4449,00	3912,00	3467,00
n^2	2345,00	2096,00	2141,00
$4*n^2$	2037,00	1968,00	2082,00
	PRD_{GA}[%]		
n	128,15	100,62	77,79
n^2	20,26	7,49	9,79
$4*n^2$	4,46	0,92	6,77

Rysunek 10: Pomiary dla pliku ftv70.atsp



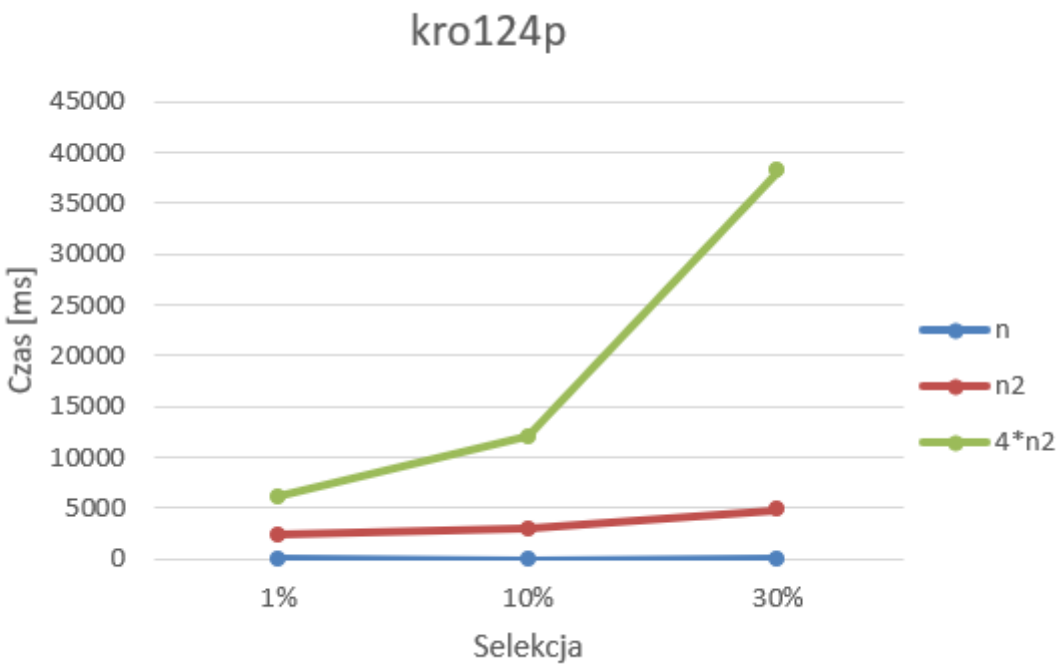
Rysunek 11: Wykres czasu dla pliku ftv70.atsp



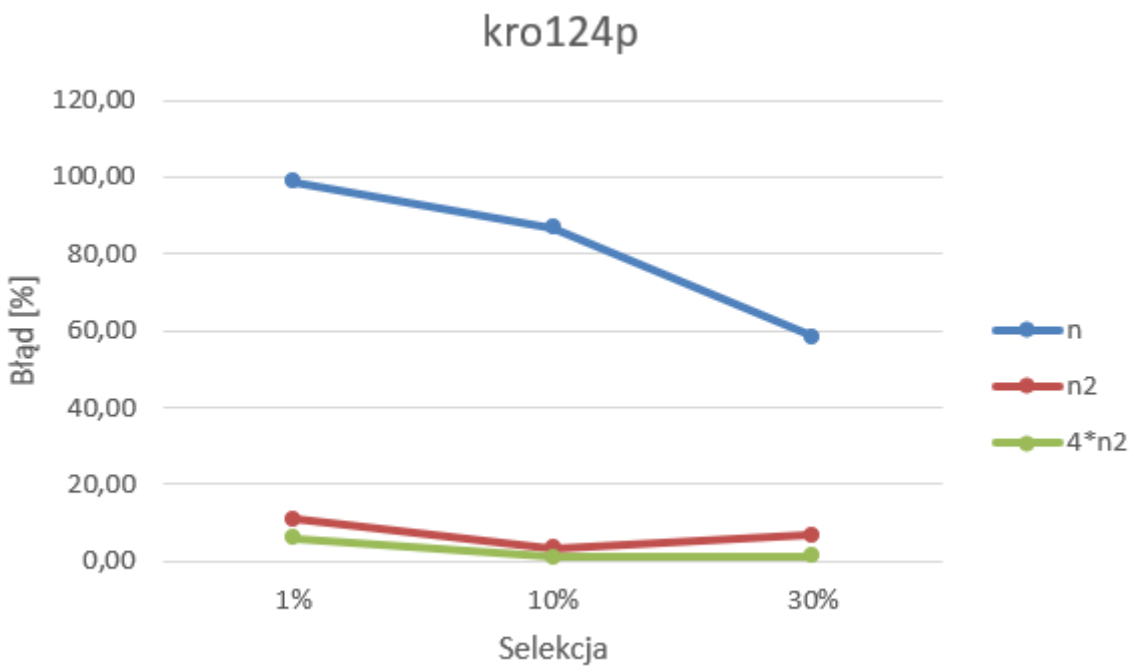
Rysunek 12: Wykres błędu dla pliku ftv70.atsp

Mutacja Swap	π = 36230		
% najlepszych	1%	10%	30%
l. osobników	t		
n	42,48	40,78	50,89
n^2	2344,41	2975,94	4841,70
$4*n^2$	6157,90	12021,71	38209,11
	π^{ref}		
n	72039,00	67652,00	57367,00
n^2	40201,00	37499,00	38681,00
$4*n^2$	38368,00	36604,00	36734,00
	PRD_{GA}[%]		
n	98,84	86,73	58,34
n^2	10,96	3,50	6,77
$4*n^2$	5,90	1,03	1,39

Rysunek 13: Pomiary dla pliku kro124p.atsp



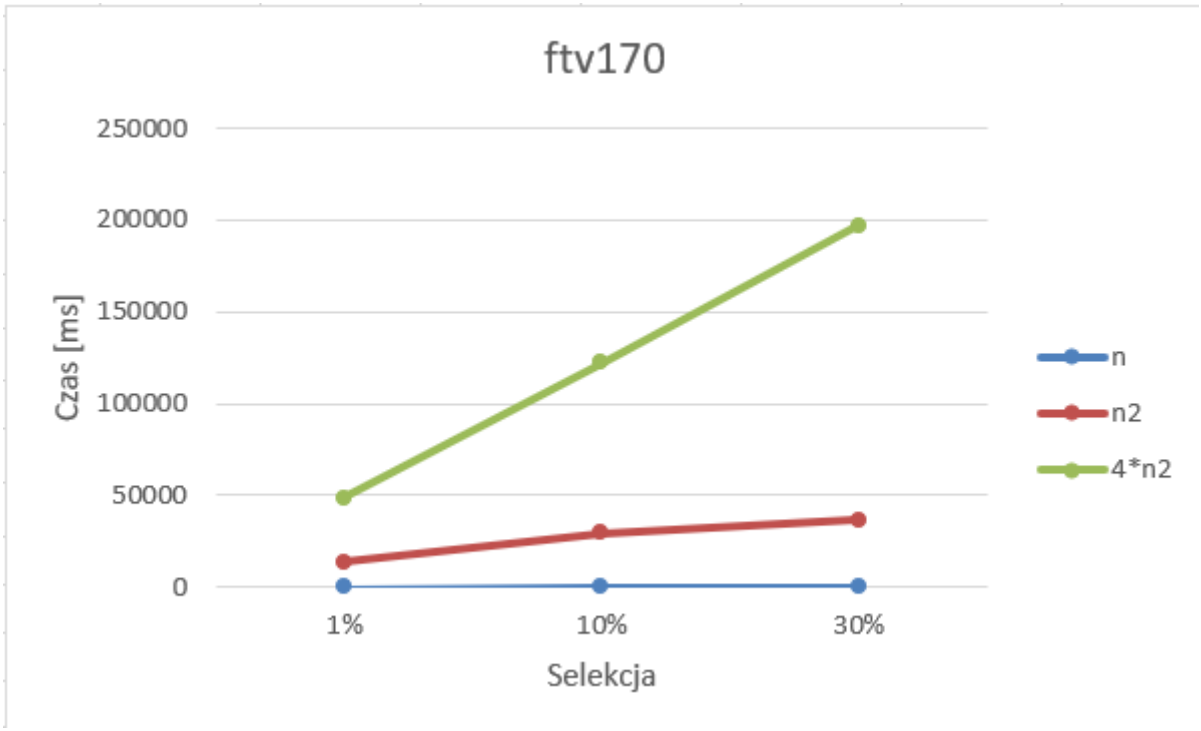
Rysunek 14: Wykres czasu dla pliku kro124p.atsp



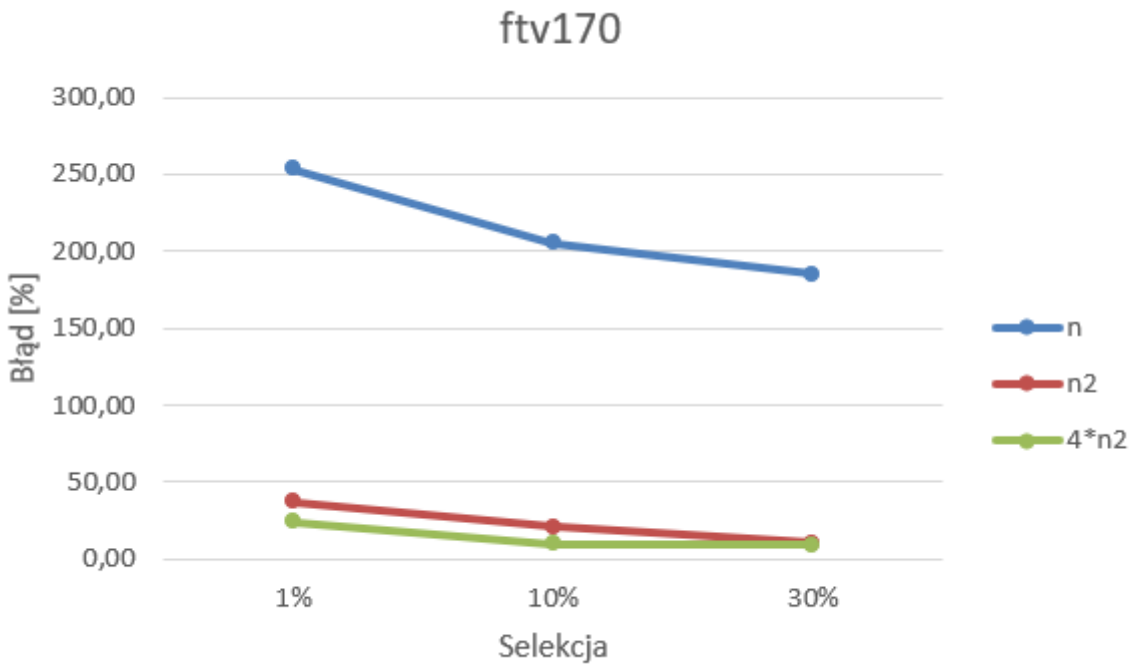
Rysunek 15: Wykres błędu dla pliku kro124p.atsp

Mutacja Swap	π = 2755		
% najlepszych	1%	10%	30%
l. osobników	t		
n	132,46	136,17	215,51
n^2	13819,40	29510,62	36789,75
$4*n^2$	48572,12	122296,40	197126,42
	π^{ref}		
n	9742,00	8412,00	7847,00
n^2	3782,00	3334,00	3042,00
$4*n^2$	3428,00	3026,00	3011,00
	PRD_{GA}[%]		
n	253,61	205,34	184,83
n^2	37,28	21,02	10,42
$4*n^2$	24,43	9,84	9,29

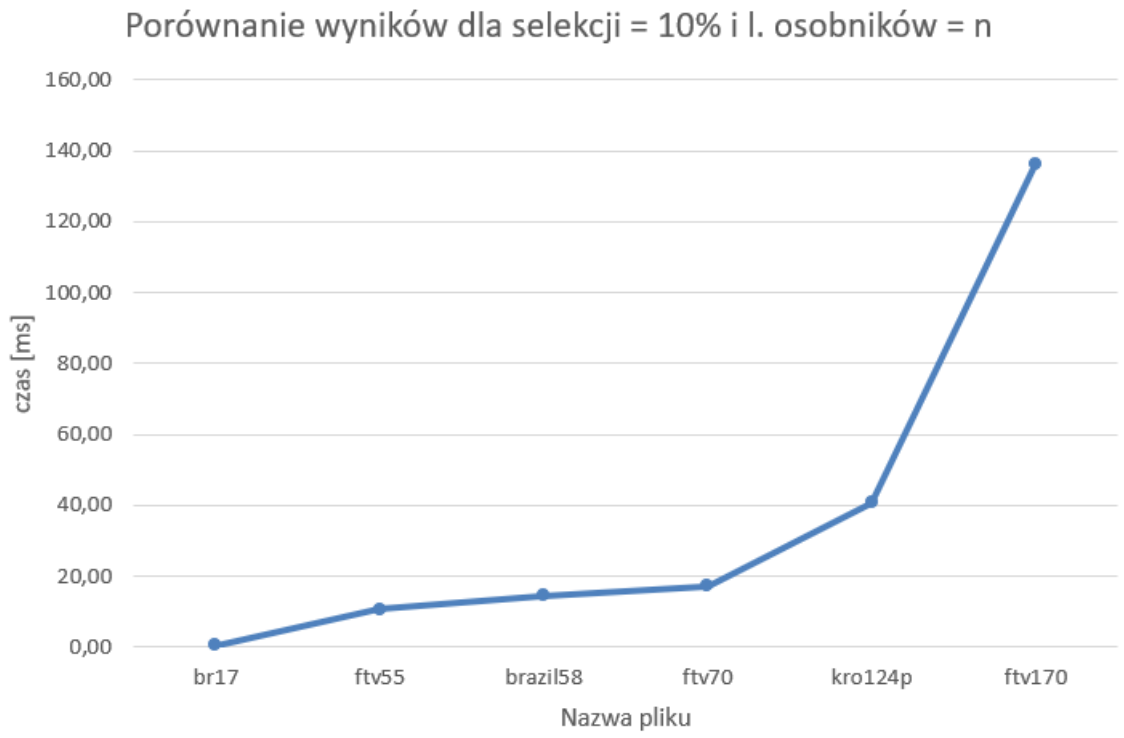
Rysunek 16: Pomiary dla pliku ftv170.atsp



Rysunek 17: Wykres czasu dla pliku ftv170.atsp



Rysunek 18: Wykres błędu dla pliku ftv170.atsp



Rysunek 19: Porównanie czasu wykonania algorytmu dla różnych instacji przy selekcji najlepszych osobników = 10% oraz liczbie osobników = n