Jan Potocki
nr indeksu 226104

SPRAWOZDANIE

Projektowanie efektywnych algorytmów
Projekt nr 3:
Implementacja i analiza efektywności algorytmu genetycznego (ewolucyjnego) dla wybranego problemu optymalizacji
pon 11:15
15.01.2018

1. Informacje teoretyczne

Problem klasy NP jest problemem, dla którego rozwiązanie można zweryfikować w czasie wielomianowym – natomiast samo znalezienie rozwiązania problemu prawie na pewno wymaga czasu ponadwielomianowego – nie znaleziono dotąd algorytmu o wielomianowej złożoności czasowej. Dla rozwiązania problemów klasy NP można zastosować algorytmy dokładne znajdujące rozwiązanie optymalne, oparte na przykład na metodzie podziału i ograniczeń lub programowania dynamicznego (czego dotyczył poprzedni projekt), jednak w rozsądnym czasie pozwalają one rozwiązywać jedynie niewielkie instancje problemów. Z kolei algorytmy heurystyczne, których przykładem mogą być algorytmy genetyczne (lub Tabu Search, którego dotyczył poprzedni projekt), pozwalają rozwiązywać w zasadzie dowolnie duże instancje problemów, jednak bez jakiejkolwiek gwarancji optymalności otrzymanego wyniku.

Problem komiwojażera:

Parametrami zadania są: skończony zbiór miast $C = \{c^1, c^2, \ldots, c^n\}$ oraz odległości d_i z miasta c_i do miasta c_j (dla symetrycznej wersji problemu istnieje wymóg $d_{ij} = d_{ji}$, nie ma go dla wersji asymetrycznej). Należy określić kolejność odwiedzania wszystkich miast ich permutację) $<c_{i[1]}, c_{i[2]}, \ldots, ci_{[n]}>$, aby sumaryczna trasa była jak najkrótsza przy założeniu, że każde miasto zostało odwiedzone dokładnie jeden raz. Wszystkie parametry zadania są liczbami naturalnymi.

2. Opis algorytmu

Napisany przeze mnie algorytm wykorzystuje metodę krzyżowania *OX* i jedną z dwóch metod mutacji – zamianę miejscami wierzchołków lub zamianę miejscami krawędzi. Selekcja dokonywana jest metodą listy rankingowej, do dalszych operacji genetycznych wybierana jest zależna od rozmiaru populacji określona liczba najkrótszych tras. Krzyżowane między sobą są wszystkie osobniki (każdy z każdym), przy czym wybór, czy dany osobnik weźmie udział w tworzeniu pary dokonywany jest losowo według współczynnika krzyżowania. Mutacja może zajść losowo dla każdego z osobników według współczynnika mutacji Rozwiązanie początkowe generowane jest za pomocą algorytmu losowo-zachłannego – pierwsze 3 wierzchołki są losowane, kolejne wybierane zachłannie. Zatrzymanie algorytmu następuje po określonym czasie, przy czym warunek stopu sprawdzany jest w każdej iteracji po selekcji, krzyżowaniu, mutacjach i ocenie przystosowania – co oznacza, że ponieważ teoretycznie dla dużych instancji problemu wykonanie jednej iteracji może trwać dłużej od zdefiniowanego czasu, rzeczywisty czas pracy algorytmu przekroczy ustawienia.

Złożoność obliczeniowa:

Ponieważ algorytmy genetyczne są algorytmami niedeterministycznymi, nie można dla nich określić czasowej złożoności obliczeniowej, można jednak upłynięcie określonego czasu pracy potraktować jako kryterium stopu algorytmu.

3. Opis implementacji algorytmu

Opisany wyżej algorytm zaimplementowałem w języku C++. Graf jest reprezentowany za pomocą własnej struktury danych (stworzonej jeszcze przy pisaniu projektów ze SDiZO): klasy abstrakcyjnej *Graph*, posiadającej dwie implementacje: *ArrayGraph* jako macierz sąsiedztwa i *ListGraph* jako listy sąsiedztwa. Populację zaimplementowałem w postaci tablicy za pomocą kontenera *std::vector z* STL. Do reprezentacji tras wchodzących w skład populacji wykorzystałem własną umowną strukturę danych: kontener *std::vector*, którego pierwszym elementem jest długość trasy obliczana na etapie oceny przystosowania, a kolejnymi elementami wierzchołki należące do danego częściowego rozwiązania. Ostateczna trasa zwracana przez algorytm jest reprezentowana jako *std::vector*, którego kolejnymi elementami są znajdujące się na niej wierzchołki.

4. Środowisko pracy i sposób prowadzenia pomiarów

Przy projektowaniu programu wykorzystanego do przeprowadzenia eksperymentu przyjąłem następujące założenia:

- struktury wykorzystywane do reprezentacji danych są alokowane dynamicznie, a struktury pomocnicze (np. zmienne do komunikacji z użytkownikiem) statycznie
- do alokacji i zwalniania pamięci wykorzystuję funkcje *new* i *delete*
- w macierzy i listach sąsiedztwa wierzchołki oraz krawędzie grafu są reprezentowane jako 32-bitowe liczby naturalne (*unsigned*)
- wszystkie pomiary są wykonywane raz, bez uśredniania wyników
- pomiary wykonywane są na 3 zestawach danych pochodzących ze strony[1]: *ftv33.atsp*, *berlin52.tsp* i *ftv70.atsp*
- pomiary na każdym z zestawów danych wykonywane są automatycznie z wykorzystaniem obydwu metod mutacji dla 4 warunków zatrzymania algorytmu: po 5, 10, 20 i 40 sekundach
- każdy z pomiarów przeprowadzany jest dla 3 wielkości populacji: 20, 50 i 80
- w każdym z pomiarów współczynnik krzyżowania wynosi 0.8, a współczynnik mutacji 0.001
- program powinien być wieloplatformowy, z tego względu do pomiarów czasu wykorzystuję napisaną przez siebie klasę *Stopwatch*, bazującą na niezależnej od żadnego systemu operacyjnego bibliotece standardowej *ctime*

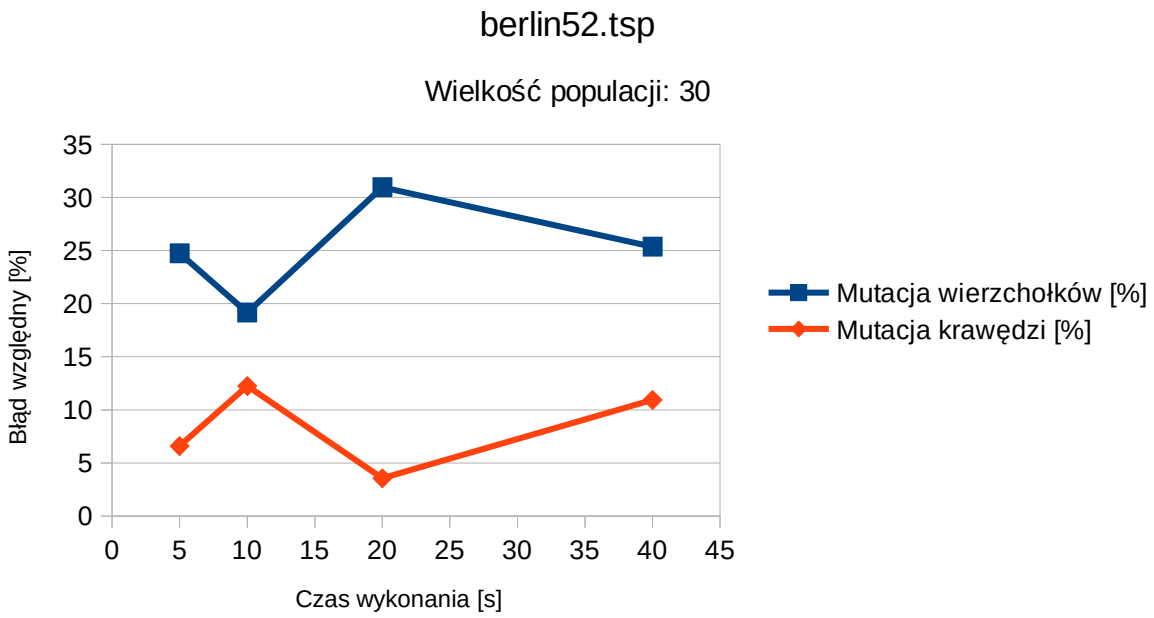
Pomiary przeprowadziłem na laptopie Lenovo ThinkPad T420, wyposażonym w dwueniowy procesor Intel Core i5-2520 M pracujący z częstotliwością bazową 2.5 GHz i 4 GB pamięci RAM DDR3 pracującej z częstotliwością 1333 MHz w trybie Dual Channel, pod kontrolą 64-bitowego systemu operacyjnego Kubuntu Linux 17.04 ze środowiskiem graficznym KDE. Pracowałem w zintegrowanym środowisku programistycznym (IDE) Code::Blocks 16.01, wykorzystując kompilator G++ 5 z ustawioną pełną optymalizacją pod kątem szybkości (flaga *-O3*). Przy pomiarach wykorzystałem reprezentację grafu w postaci macierzy sąsiedztwa.

5. Wyniki

berlin52.tsp – 52 miasta, najlepsza znana trasa: 7542
wielkość populacji: 30

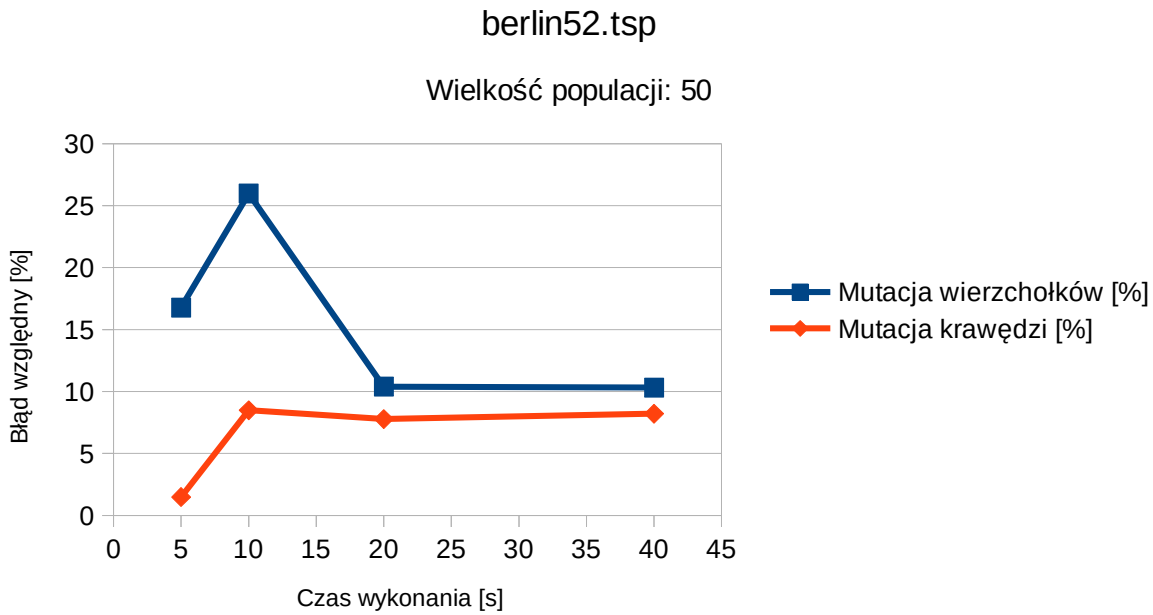
Czas [s]	Mutacja wierzchołków [%]	Mutacja krawędzi [%]	Mutacja wierzchołków	Mutacja krawędzi
5	24,73	6,59	9407	8039
10	19,16	12,24	8987	8465
20	30,95	3,55	9876	7810
40	25,35	10,93	9454	8366

1 http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/



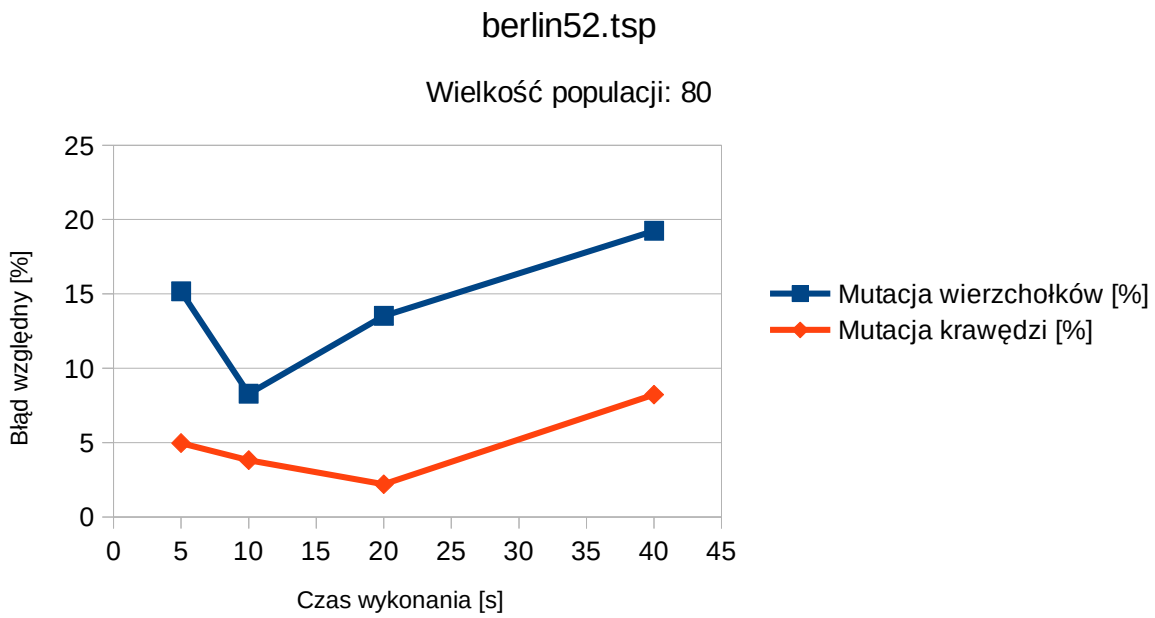
berlin52.tsp – 52 miasta, najlepsza znana trasa: 7542
wielkość populacji: 50

Czas [s]	Mutacja wierzchołków [%]	Mutacja krawędzi [%]	Mutacja wierzchołków	Mutacja krawędzi
5	16,77	1,49	8807	7654
10	25,97	8,49	9501	8182
20	10,40	7,78	8326	8129
40	10,32	8,21	8320	8161



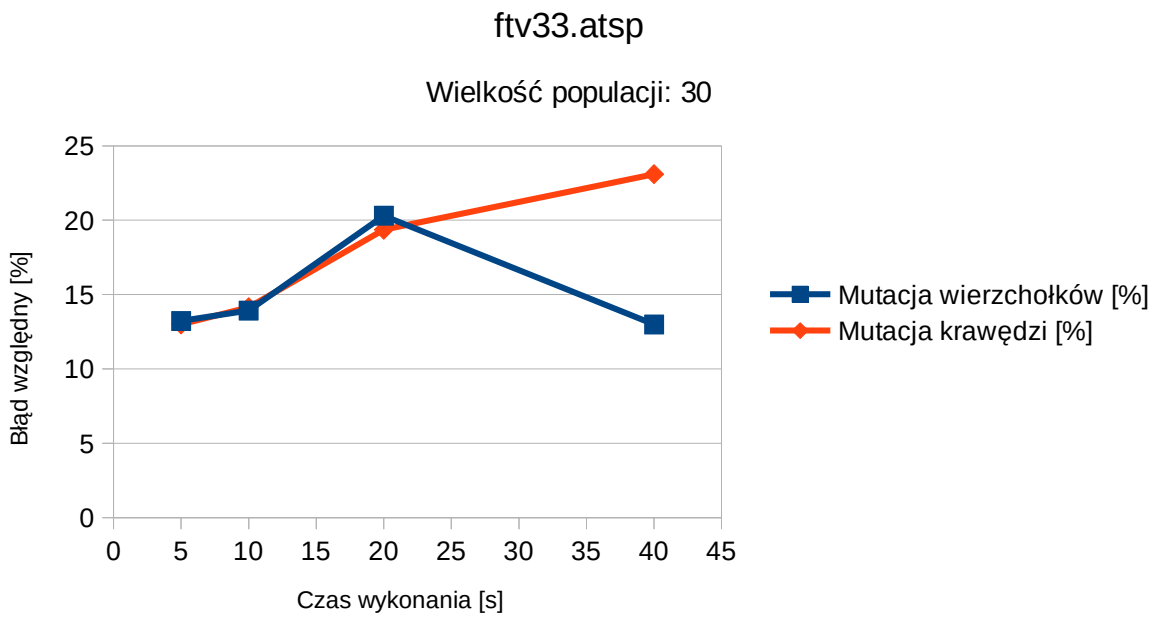
berlin52.tsp – 52 miasta, najlepsza znana trasa: 7542
wielkość populacji: 80

Czas [s]	Mutacja wierzchołków [%]	Mutacja krawędzi [%]	Mutacja wierzchołków	Mutacja krawędzi
5	15,17	4,97	8686	7917
10	8,29	3,82	8167	7830
20	13,52	2,20	8562	7708
40	19,24	8,23	8993	8163



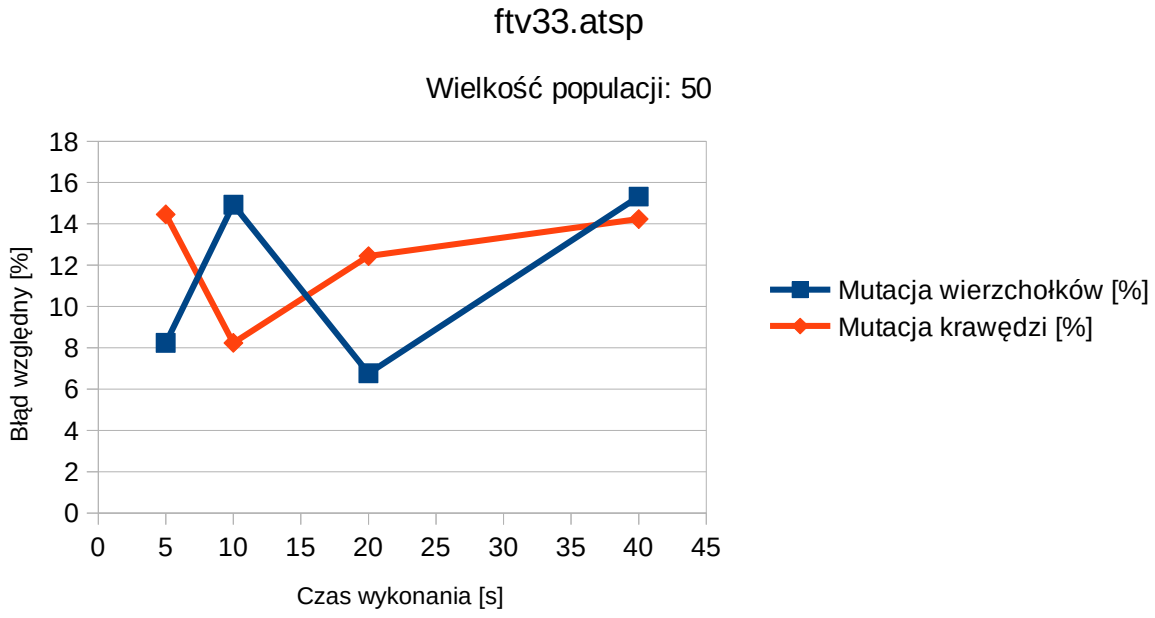
ftv33.atsp – 34 miasta, najlepsza znana trasa: 1286
wielkość populacji: 30

Czas [s]	Mutacja wierzchołków [%]	Mutacja krawędzi [%]	Mutacja wierzchołków	Mutacja krawędzi
5	13,22	12,99	1456	1453
10	13,92	14,15	1465	1468
20	20,30	19,36	1547	1535
40	12,99	23,09	1453	1583



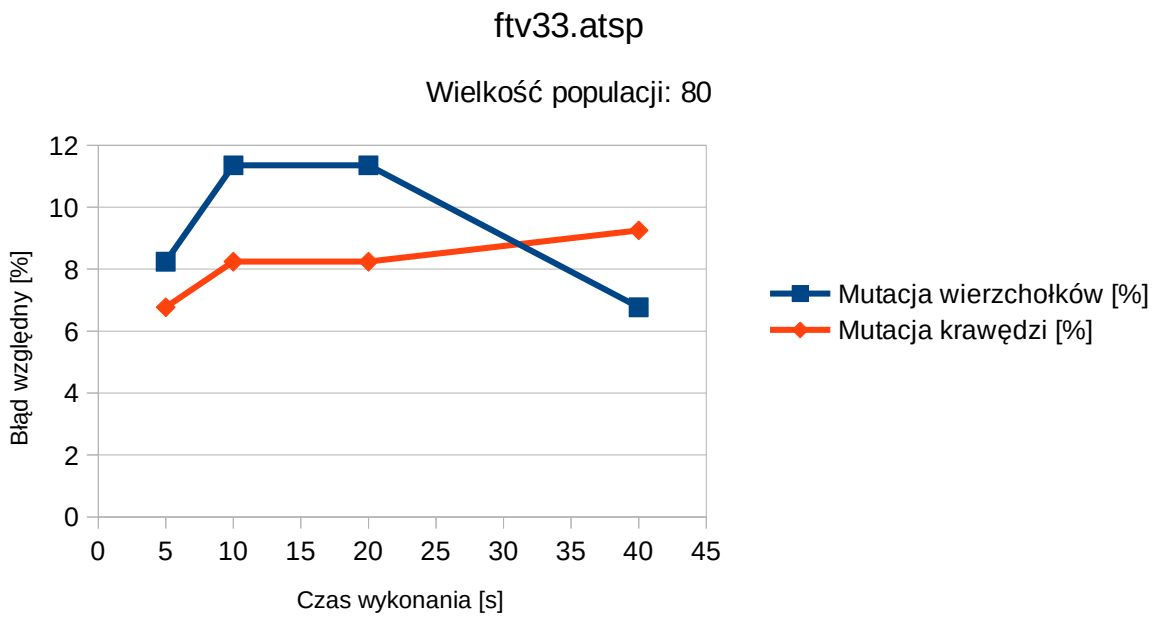
ftv33.atsp – 34 miasta, najlepsza znana trasa: 1286
wielkość populacji: 50

Czas [s]	Mutacja wierzchołków [%]	Mutacja krawędzi [%]	Mutacja wierzchołków	Mutacja krawędzi
5	8,24	14,46	1392	1472
10	14,93	8,24	1478	1392
20	6,77	12,44	1373	1446
40	15,32	14,23	1483	1469



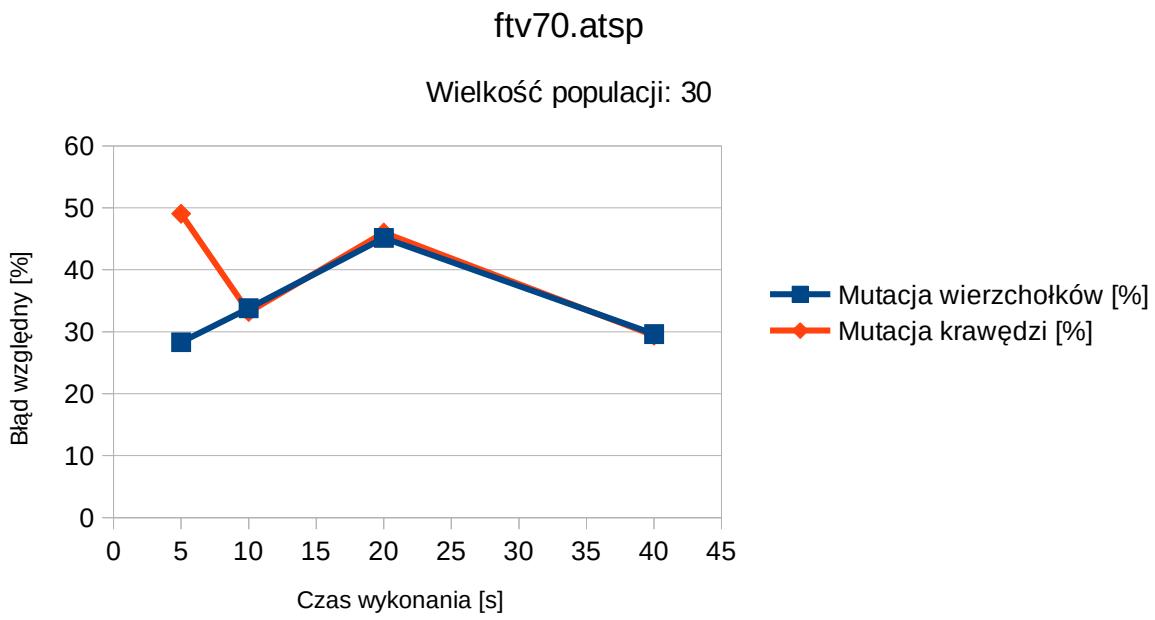
ftv33.atsp – 34 miasta, najlepsza znana trasa: 1286
wielkość populacji: 80

Czas [s]	Mutacja wierzchołków [%]	Mutacja krawędzi [%]	Mutacja wierzchołków	Mutacja krawędzi
5	8,24	6,77	1392	1373
10	11,35	8,24	1432	1392
20	11,35	8,24	1432	1392
40	6,77	9,25	1373	1405



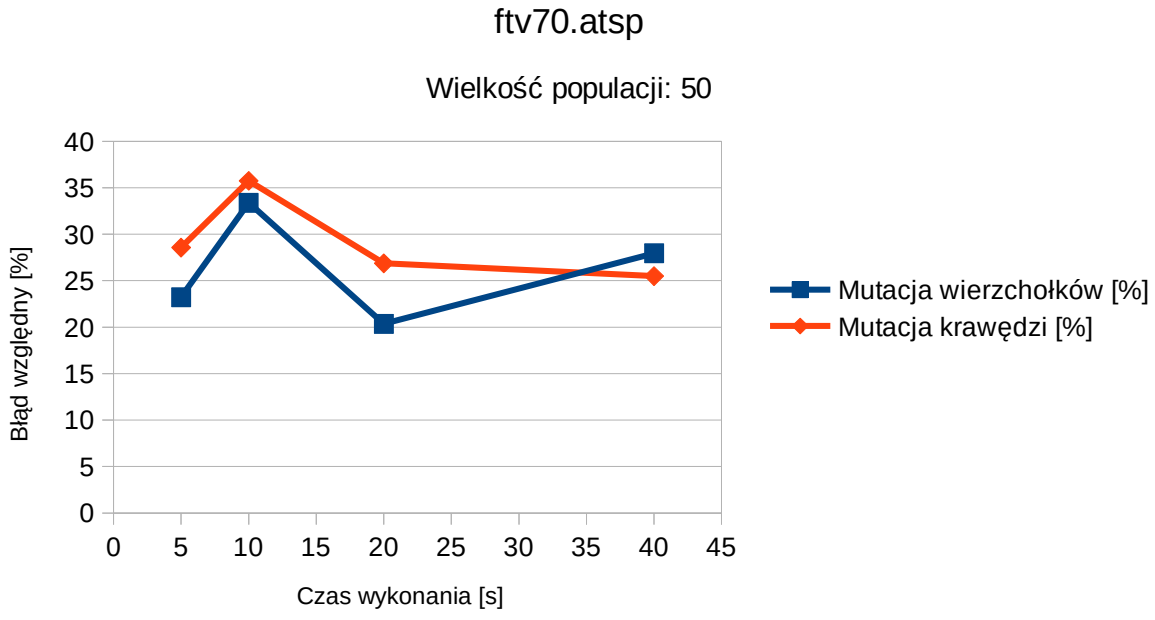
ftv70.atsp – 71 miast, najlepsza znana trasa: 1950
wielkość populacji: 30

Czas [s]	Mutacja wierzchołków [%]	Mutacja krawędzi [%]	Mutacja wierzchołków	Mutacja krawędzi
5	28,31	49,03	2502	2906
10	33,79	33,18	2609	2597
20	45,13	46,00	2830	2847
40	29,64	29,38	2528	2523



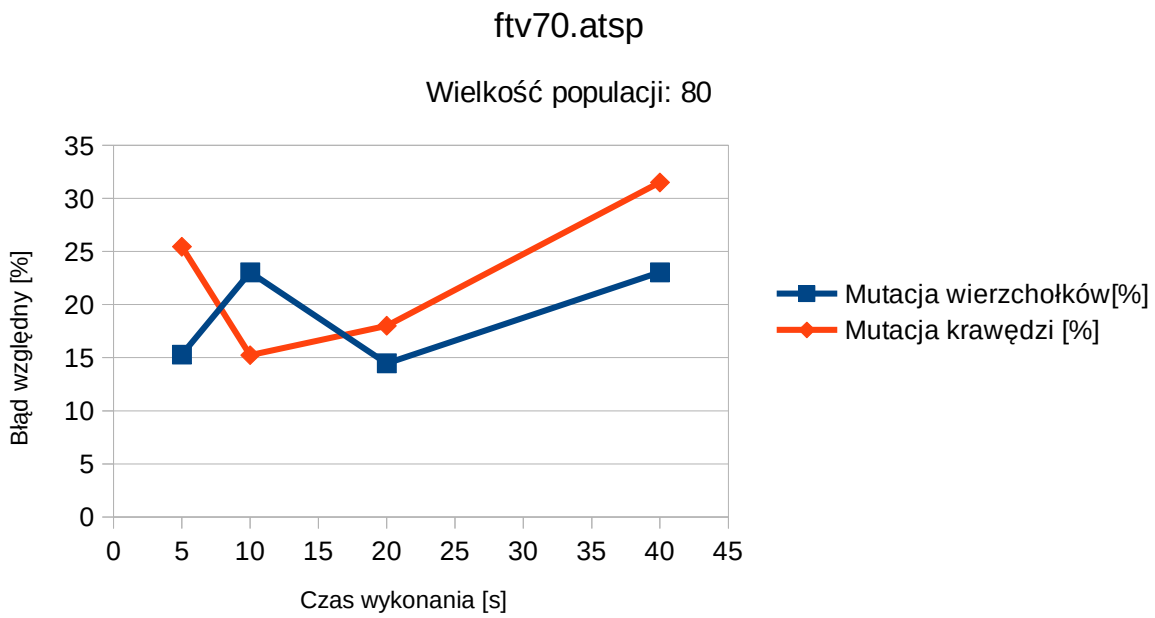
ftv70.atsp – 71 miast, najlepsza znana trasa: 1950
wielkość populacji: 50

Czas [s]	Mutacja wierzchołków [%]	Mutacja krawędzi [%]	Mutacja wierzchołków	Mutacja krawędzi
5	23,23	28,56	2403	2507
10	33,38	35,74	2601	2647
20	20,36	26,87	2347	2474
40	27,95	25,49	2495	2447



ftv70.atsp – 71 miast, najlepsza znana trasa: 1950
wielkość populacji: 80

Czas [s]	Mutacja wierzchołków[%]	Mutacja krawędzi [%]	Mutacja wierzchołków	Mutacja krawędzi
5	15,28	25,44	2248	2446
10	23,03	15,23	2399	2247
20	14,46	18,00	2232	2301
40	23,03	31,49	2399	2564



Zestaw danych	Rozwiązanie	Tabu Search		Algorytm genetyczny	
		Najlepszy wynik	Błąd względny [%]	Najlepszy wynik	Błąd względny [%]
berlin52.tsp	7542	8925	9,97	7654	1,49
ftv33.atsp	1286	1286	0	1373	6,77
ftv70.atsp	1950	2215	13,53	2232	14,46

Tabela 1: Porównanie algorytmu TS z algorytmem genetycznym

6. Wnioski

Z przeprowadzonych badań widać dobrze, że algorytm genetyczny należy do algorytmów niedeterministycznych. Chociaż jest w stanie czasami dać lepsze wyniki niż Tabu Search, jest od niego mniej powtarzalny i nie widać w jego przypadku reguły, że wraz ze wzrostem czasu wykonania wzrasta dokładność rozwiązania. Podobnie nie ma reguły, który z operatorów mutacji da lepsze rezultaty – nawet na tym samym zestawie danych. Jedynym zestawem, dla którego jednoznacznie lepsza była mutacja przez zamianę wierzchołków jest *berlin52.tsp*. Wszystkie te obserwacje generalnie pokazują, że nie ma uniwersalnie dobrych algorytmów heurystycznych. Dla jednego problemu lepsza może okazać się jedna metaheurystyka, dla innego – inna.

Wrocław, 15.01.2018