Problem komiwojażera

Rozwiązanie za pomocą algorytmu genetycznego

Wojciech Regulski 132312
wojciech.regulski@student.put.poznan.pl
Paweł Kuffel 132264
pawel.kuffel@student.put.poznan.pl

Opis działania generatora instancji:

Generator instancji oczekuje od użytkownika podania rozmiaru instancji do wygenerowania. Następnie zgodnie z przyjętą konwencją zapisuje do pliku rozmiar instancji. Generowanie odbywa się poprzez losowanie w pętli pozycji x i y miasta (od 0 do 1999) i sprawdzenie w pętli czy nie istnieje już miasto o takich współrzędnych. Jeśli istnieje zostaje wylosowane ponownie, w przeciwnym wypadku pozycja nowego miasta zostaje zapisana w tablicy i do pliku dopisywany jest numer miasta i jego współrzędne.

Opis działania i implementacja algorytmu genetycznego:

Rozwiązanie zostało zaimplementowane w języku C#, w środowisku Microsoft Visual Studio C#.

Reprezentacja danych:

Miasta:

Współrzędne (odcięte i rzędne) miast są zapisane w tablicy citiesPositions o wymiarach n x 2, gdzie n to łączna liczba miast. Odległości między każdą parą miast obliczane są po wczytaniu danych wejściowych i zapisywane w tablicy distancesBetweenCities o wymiarach n x n.

Ścieżki (rozwiązania):

Do reprezentacji ścieżek utworzono klasę Tour dziedziczącą po liście liczb typu int: List<int>. Obiekt klasy Tour przechowuje numery wierzchołków w kolejności odpowiadającej danej ścieżce. Posiada ona pole distance (oraz jego getter) przechowujące jej długość, metodę UpdateDistance() aktualizującą pole distance, metodę Hash(), metodę MakeRandomTour() zapełniającą obiekt klasy Tour losowo wygenerowaną ścieżką oraz metodę CompareTo() która porównuje długość bieżącej ścieżki z długością ścieżki reprezentowanej przez inny obiekt typu Tour podany jako parametr.

Bieżąca populacja (lista ścieżek: List<Tour>) przechowywana jest w zmiennej tourPopulation.

Parametry strojenia:

TOP_SURVIVORS

[0..100]

Z pokolenia n do pokolenia n+1 bezwarunkowo przejdą rozwiązania których długość jest mniejsza lub równa percentylowi TOP_SURVIVORS%

BOTTOM_SURVIVORS

[0..100]

Z pokolenia n do pokolenia n+1 bezwarunkowo przejdą rozwiązania których długość jest większa lub równa percentylowi (100% - BOTTOM_SURVIVORS%)

MUTATION_PROBABILITY

[0..100]

Prawdopodobieństwo wystąpienia przynajmniej jednej mutacji w jednej ścieżce.

MAX_MUTATIONS

[0..n]

Maksymalna liczba mutacji która może wystąpić w jednej ścieżce.

duplicatesRemovalInterval

[1..\inf]

Co duplicatesRemovalInterval pokoleń na danej populacji zostanie uruchomiona procedura usuwania duplikatów.

iterations

[1..\inf]

Liczba iteracji (pokoleń) po której program kończy działanie.

populationSize

[1..\inf]

Liczba rozwiązań w populacji

Opis działania:

Po wczytaniu danych wejściowych z pliku wybranego przez użytkownika z poziomu GUI, działanie programu jest ilustrowane przez poniższy pseudokod:

generuj_populację_pierwotną;
do czasu aż licznik pokoleń nie osiągnął iterations:
    generuj_populację_następnego_kroku;

if (w nowej populacji istnieje ścieżka, której długość jest < od obecnie najkrótszej ścieżki):
przyjmij tę ścieżkę za obecnie najkrótszą
if (od ostatniego szukania duplikatów minęło >= duplicatesRemovalInterval iteracji):
usun_duplikaty;

Generowanie początkowej populacji:

zaimplementowane w void Populate()

Pierwotna populacja budowana jest z n rozwiązań uzyskanych z działania algorytmu zachłannego poprawionego, który rozpoczyna pracę oddzielnie dla każdego z n wierzchołków (miast). Następnie generowane jest n losowych ścieżek w celu zwiększenia różnorodności populacji. Ponieważ liczebność populacji jest stała (nie zmienia się z pokolenia na pokolenie), może zajść konieczność degenerowania dodatkowych osobników (rozwiązań). W celu uzyskania nowego osobnika (rozwiązanie), krzyżuje się dwa istniejące i otrzymane go osobnika poddaje się mutacji.

Generowanie populacji kolejnego kroku:

zaimplementowane w void SimulateGeneration()

Następna populacja generowana jest w oparciu o bieżącą (przechowywaną w liście obiektów typu Tour – tourPopulation). Rozwiązania (osobniki) które mają znaleźć się w nowej generacji są zapisywane w liście obiektów typu Tour: tempPopulation. Ponieważ liczebność populacji ma być stała, po zakończeniu tego kroku, liczba elementów tempPopulation musi być równa liczbie elementów tourPopulation.

Na początku, z bieżącej populacji do nowej kopiowane jest TOP_SURVIVORS% rozwiązań o najniższej długości ścieżki.

Następnie, tworzone są nowe osobniki, każdy przez krzyżowanie dwóch istniejących w bieżącej populacji, tak aby w nowej populacji pozostało BOTTOM_SURVIVORS% miejsca.

BOTTOM_SURVIVORS% rozwiązań o największej długości ścieżki z bieżącej populacji kopiowane jest do nowej populacji.

Ścieżki z tempPopulation zostają przekopiowane do tourPopulation.

Dobór osobników do „rozrodu” oraz mechanizm crossover zostaną opisane później.

Po zakończeniu generowania populacji kolejnego kroku, lista rozwiązań (tempPopulation) jest sortowana rosnąco po długości ścieżki.

Usuwanie duplikatów:

Zaobserwowano, że w większości przypadków, po dostatecznie dużej liczbie iteracji (pokoleń), w liście rozwiązań zaczęły pojawiać się duplikaty. W celu rozwiązania tego problemu opracowano metodę usuwania duplikatów, która uruchamiana jest cyklicznie co duplicatesRemovalInterval iteracji (pokoleń).

Ponieważ podejście oparte na porównywaniu rozwiązań w populacji na zasadzie „każdy z każdym" byłoby bardzo nieefektywne (złożoność obliczeniowa O(n^2) przy założeniu że dwa rozwiązania da się porównać w czasie O(1)), wykorzystano fakt iż lista rozwiązań jest posortowana rosnąco po długościach oraz iż kolizje długości (będącej liczbą zmiennoprzecinkową) dwóch różnych rozwiązań występują bardzo rzadko.

Zastosowany algorytm porównuje sąsiednie rozwiązania parami i jeśli wykryje duplikat, poddaje mutacji drugie rozwiązanie z pary.

Porównywanie rozwiązań odbywa się przez porównanie ich długości, i jeżeli są one równe – porównanie wartości funkcji Hash() obu rozwiązań.

Znajdowanie osobników do crossover:

Zaimplementowane w int FindPartner()

Dobieranie pary osobników do rozrodu odbywa się przez oddzielne znalezienie dwóch różnych od siebie osobników. Funkcja FindPartner() zwraca indeks na liście tourPopulation znalezionego (wylosowanego) osobnika. Postanowiono, iż rozwiązania o najniższych wartościach długości ścieżki będą miały najwyższą szansę na zostanie wybranymi. W implementacji, program przechowuje wszystkie osobniki z danego pokolenia w tourPopulation, gdzie są one posortowane rosnąco pod względem długości ścieżki. Prawdopodobieństwo wyboru i-tego z nich jest liniowo malejące. Prawdopodobieństwo wyboru pierwszego (zerowego) elementu wynosi 2/populationSize.

Crossover:

Zaimplementowane w void Crossover()

Dla dwóch zadanych osobników: parent1 i parent2, tworzony jest nowy osobnik. Z parent1 wybierany jest losowy podciąg (kolejnych wierzchołków) który zostaje przekopiowany do nowego osobnika (rozwiązania). Następnie wierzchołki które jeszcze nie występują w nowym rozwiązaniu, są do niego przekopiowywane (dodawane na koniec) w kolejności takiej, w jakiej występowały w parent2. Ilustruje to poniższy schemat:

```
parent1: 2 3 5 6 4 1

parent2: 6 1 5 4 2 3

child:   2 3 5 6 1 4
```

```
Wybrany podciąg

Pozostałe elementy parent1
```

Mutacja:

Zaimplementowane w void Mutate()

Mutacja na danym osobniku odbywa się przez losowy wybór MAX_MUTATIONS par elementów ścieżki i zamiany miejscami wierzchołków należących do danej pary.

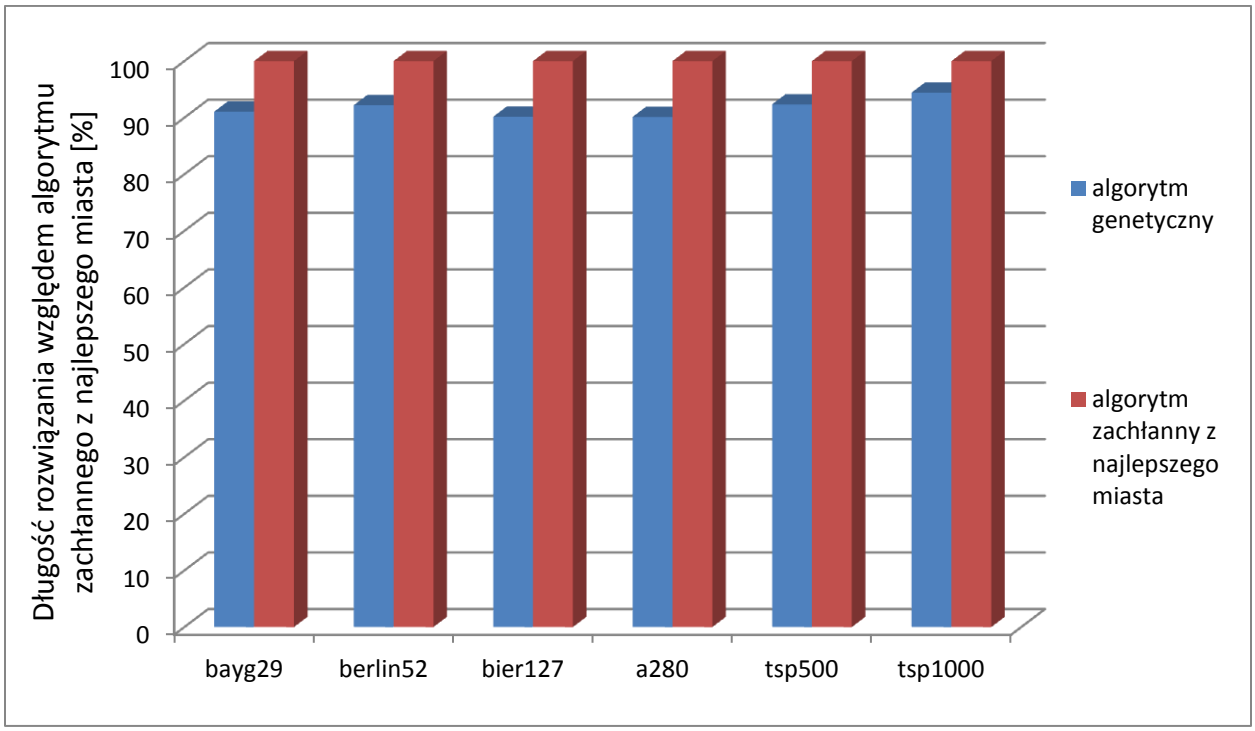
GUI

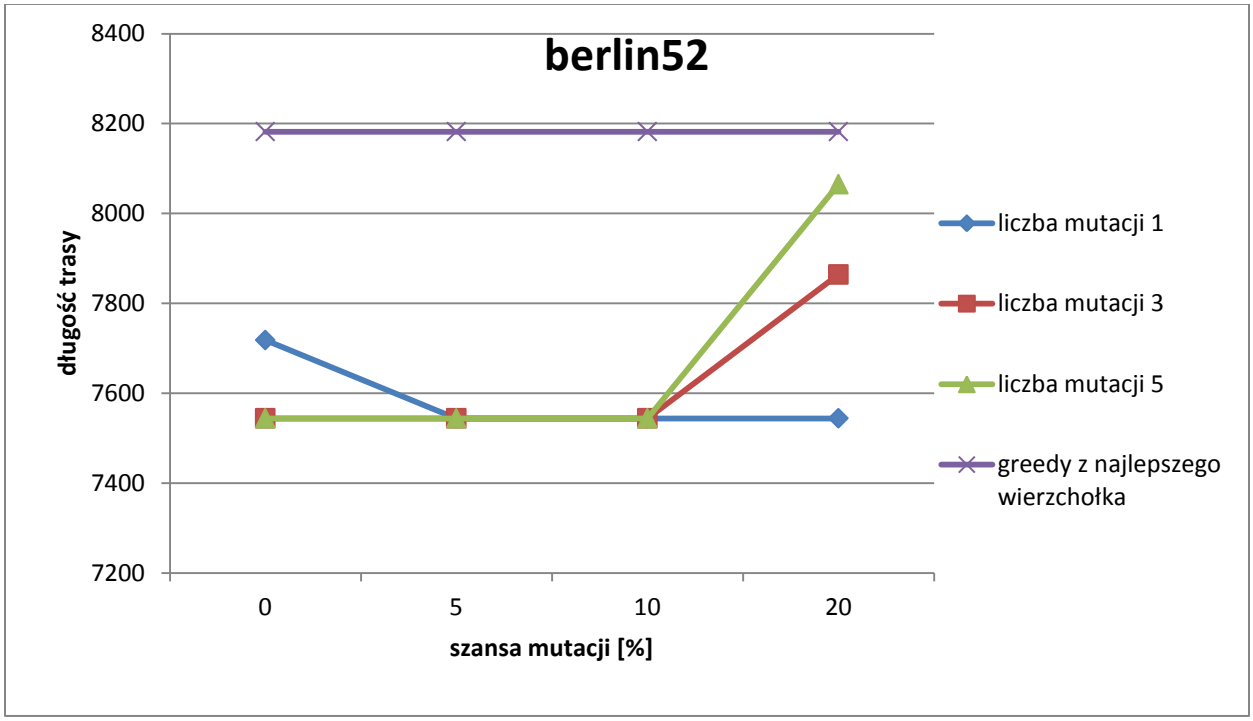
Dla ułatwienia analizy wyników działania programu, zaimplementowano GUI którego funkcjonalność to: ustawianie parametrów strojenia, wczytywanie instancji z pliku, wyświetlanie mapy miast na płaszczyźnie kartezjańskiej, wyświetlanie obecnie najlepszego rozwiązania jako połączeń między miastami na mapie, sterowanie wykonaniem programu (start/stop), podświetlanie różnic między obecnie najlepszym rozwiązaniem a poprzednim, wypisywanie obecnie najlepszego rozwiązania jako ciągu miast. GUI i główny program działają na oddzielnych wątkach.

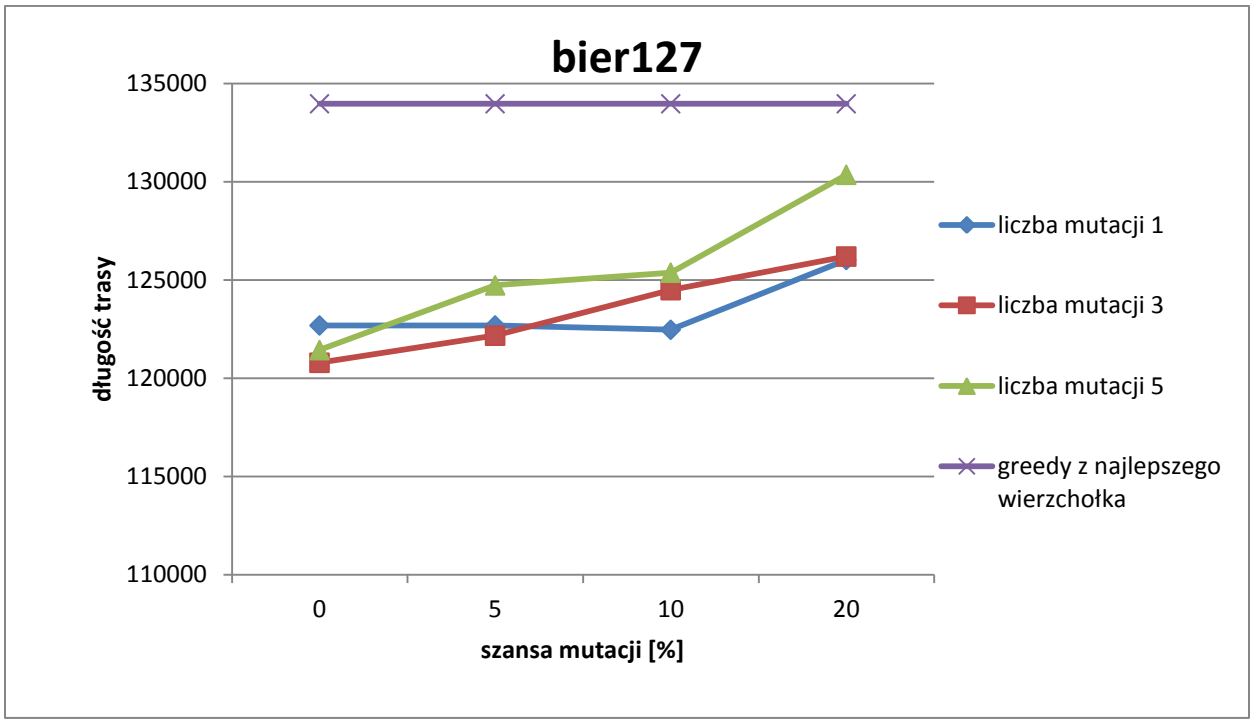
Wyniki testów:

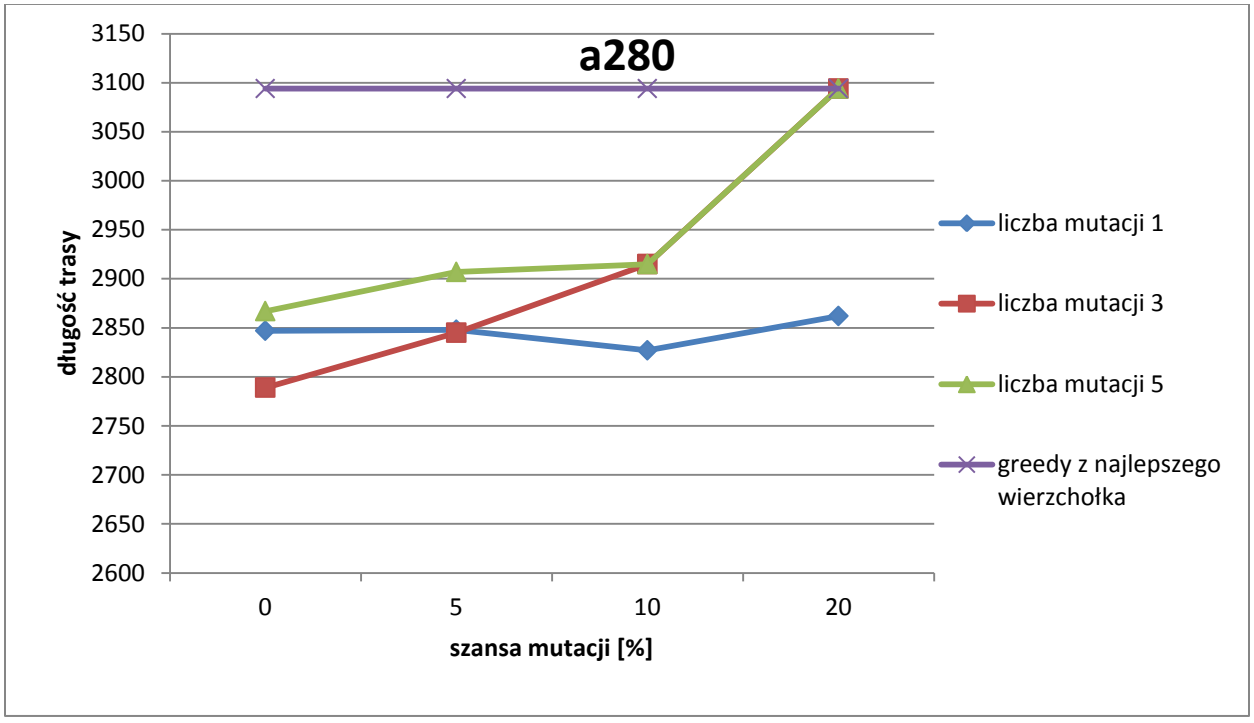
liczba mutacji	szansa mutacji [%]	rozmiar populacji	bayg29	berlin52	bier127	a280	tsp500	tsp1000
greedy z najlepszego wierzchołka			9964	8182	133970	3094	97770	28178
1	0	10000	9074	7718	122687	2847	90597	26609
1	5	10000	9074	7544	122687	2848	90293	27091
1	10	10000	9074	7544	122467	2827	91950	27665
1	20	10000	9074	7544	126020	2862	94880	28178
1	5	1000	9074	7736	122142	2804	91789	27173
3	0	10000	9074	7544	120789	2789	91655	27495
3	5	10000	9074	7544	122168	2845	92720	27911
3	10	10000	9074	7544	124485	2915	94069	26897
3	20	10000	9074	7864	126195	3094	97770	28178
3	5	1000	9311	7598	123062	2905	92494	27745
5	0	10000	9074	7544	121435	2867	91860	27671
5	5	10000	9074	7544	124727	2907	92531	27780
5	10	10000	9074	7544	125374	2915	94445	27509
5	20	10000	9074	8065	130356	3094	97770	28178
5	5	1000	9074	7544	122760	2882	94331	27857

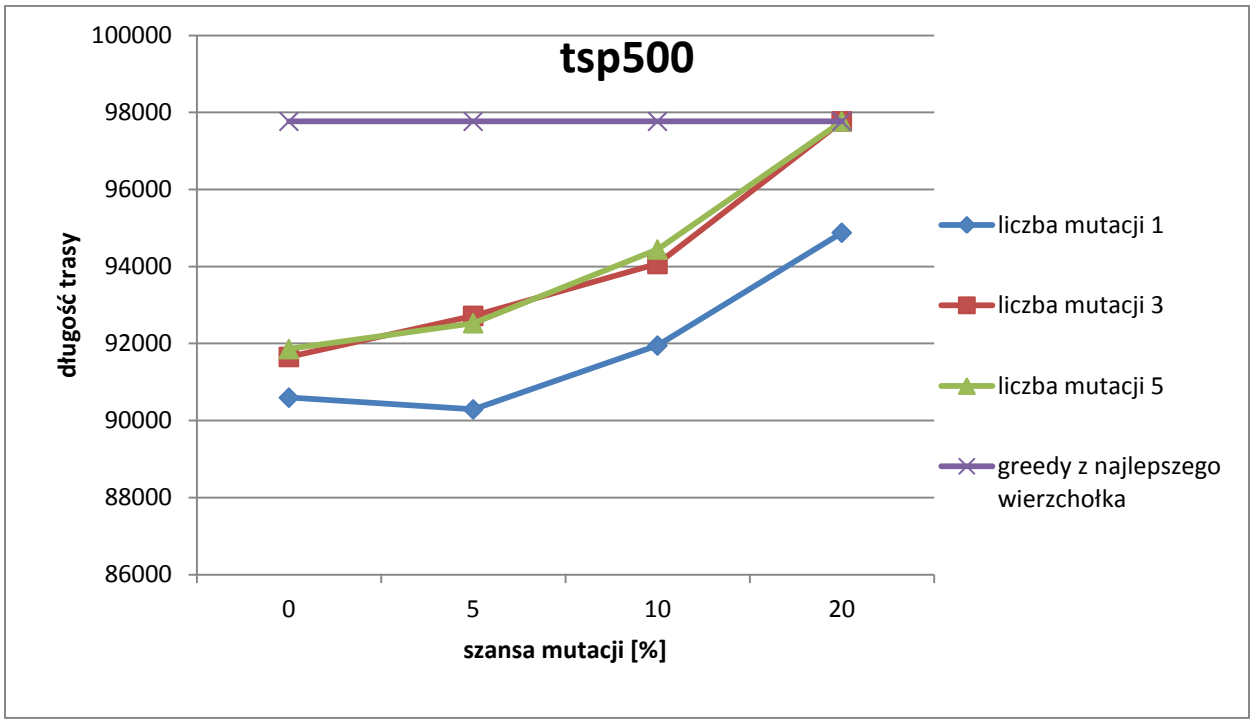
liczba mutacji	szansa mutacji [%]	rozmiar populacji	bayg29	berlin52	bier127	a280	tsp500	tsp1000
greedy z najlepszego wierzchołka			100,00	100,00	100,00	100,00	100,00	100,00
1	0	10000	91,07	94,33	91,58	92,02	92,66	94,41
1	5	10000	91,07	92,20	91,58	92,05	92,35	96,14
1	10	10000	91,07	92,20	91,41	91,37	94,05	98,18
1	20	10000	91,07	92,20	94,07	92,50	97,04	100,00
1	5	1000	91,07	94,55	91,17	90,63	93,88	97,14
3	0	10000	91,07	92,20	90,16	90,14	93,75	97,58
3	5	10000	91,07	92,20	91,19	91,95	94,83	99,05
3	10	10000	91,07	92,20	92,92	94,21	96,21	95,45
3	20	10000	91,07	96,11	94,20	100,00	100,00	100,00
3	5	1000	93,45	92,86	91,86	93,89	94,60	98,46
5	0	10000	91,07	92,20	90,64	92,66	93,96	98,20
5	5	10000	91,07	92,20	93,10	93,96	94,64	98,59
5	10	10000	91,07	92,20	93,58	94,21	96,60	97,63
5	20	10000	91,07	98,57	97,30	100,00	100,00	100,00
5	5	1000	91,07	92,20	91,63	93,15	96,48	98,86

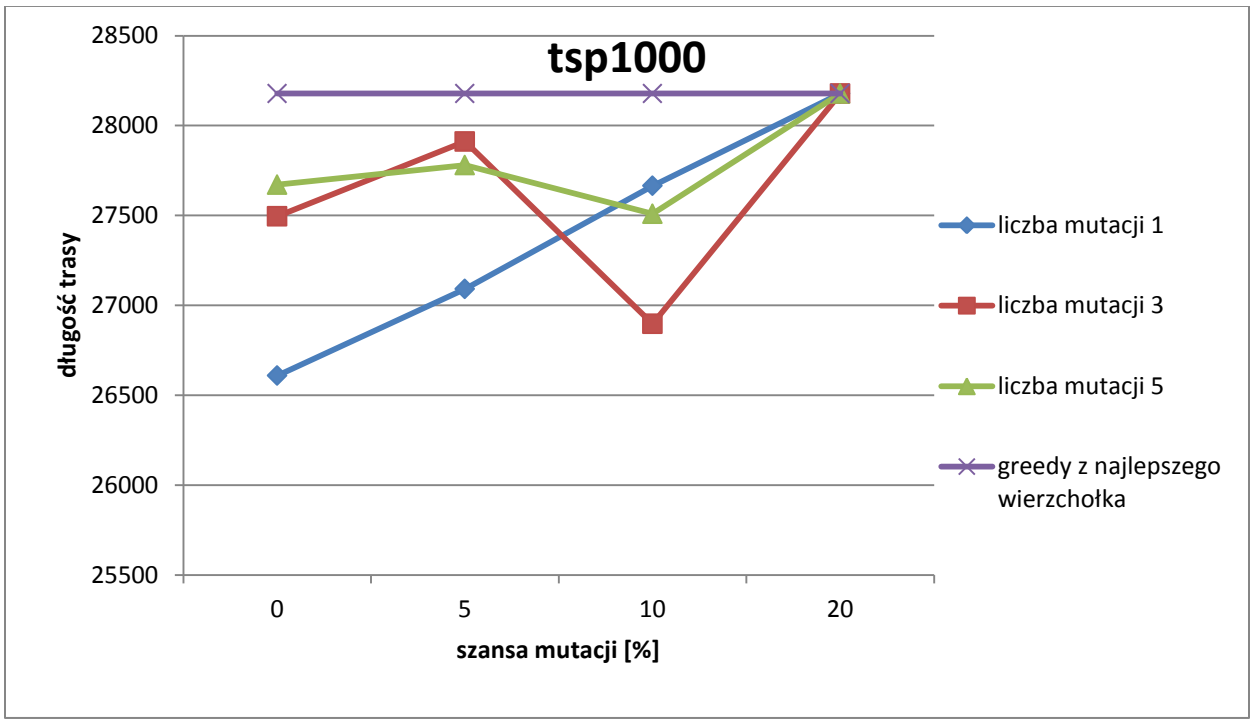

Długość rozwiązania względem algorytmu zachłannego z najlepszego miasta [%]
100
90
80
70
60
50
40
30
20
10
0
bayg29
berlin52
bier127
a280
tsp500
tsp1000
algorytm genetyczny
algorytm zachłanny z najlepszego miasta


berlin52
8400
8200
8000
7800
7600
7400
7200
długość trasy
0
5
10
20
szansa mutacji [%]
liczba mutacji 1
liczba mutacji 3
liczba mutacji 5
greedy z najlepszego wierzchołka

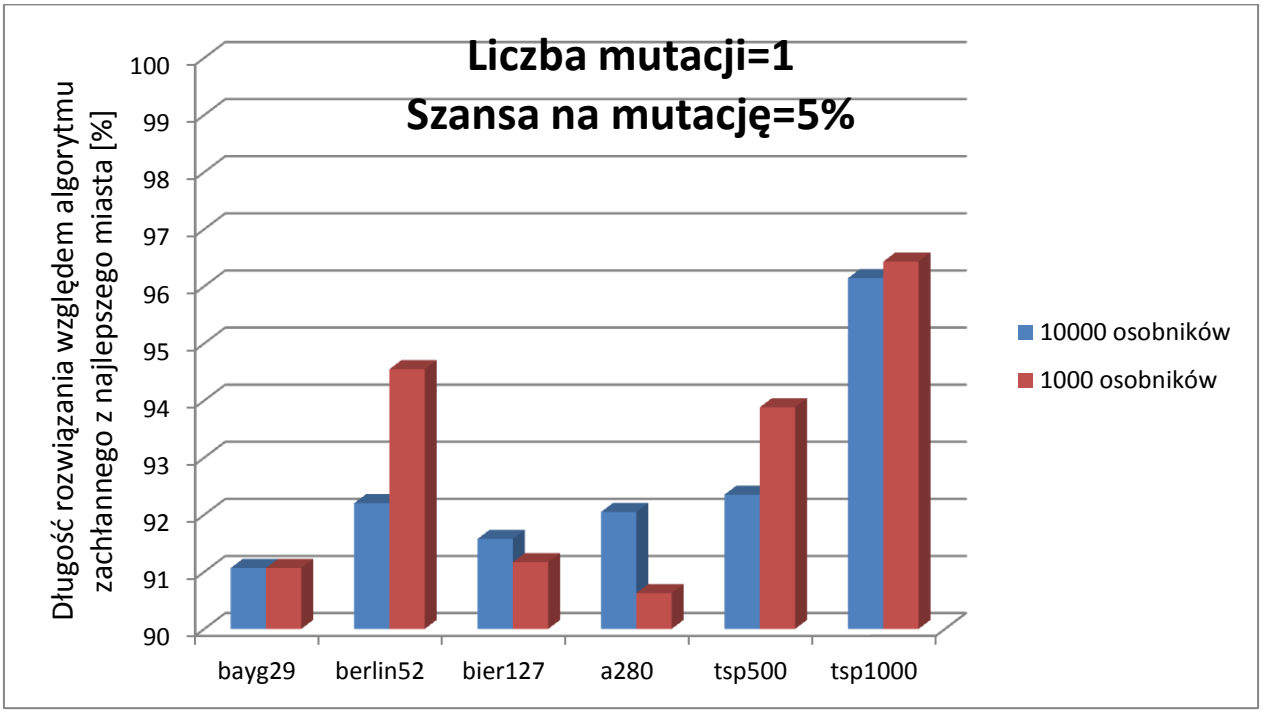

bier127
135000
130000
125000
120000
115000
110000
długość trasy
0
5
10
20
szansa mutacji [%]
liczba mutacji 1
liczba mutacji 3
liczba mutacji 5
greedy z najlepszego wierzchołka

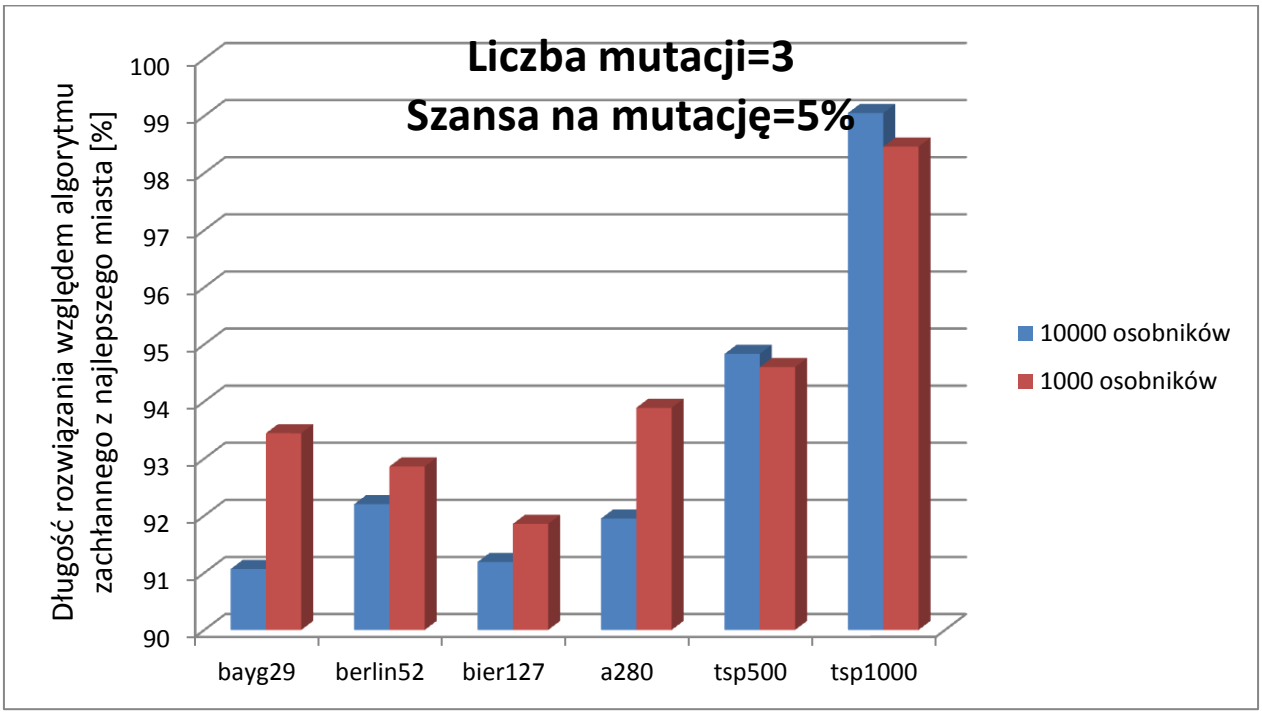

a280
3150
3100
3050
3000
2950
2900
2850
2800
2750
2700
2650
2600
długość trasy
0
5
10
20
szansa mutacji [%]
liczba mutacji 1
liczba mutacji 3
liczba mutacji 5
greedy z najlepszego wierzchołka

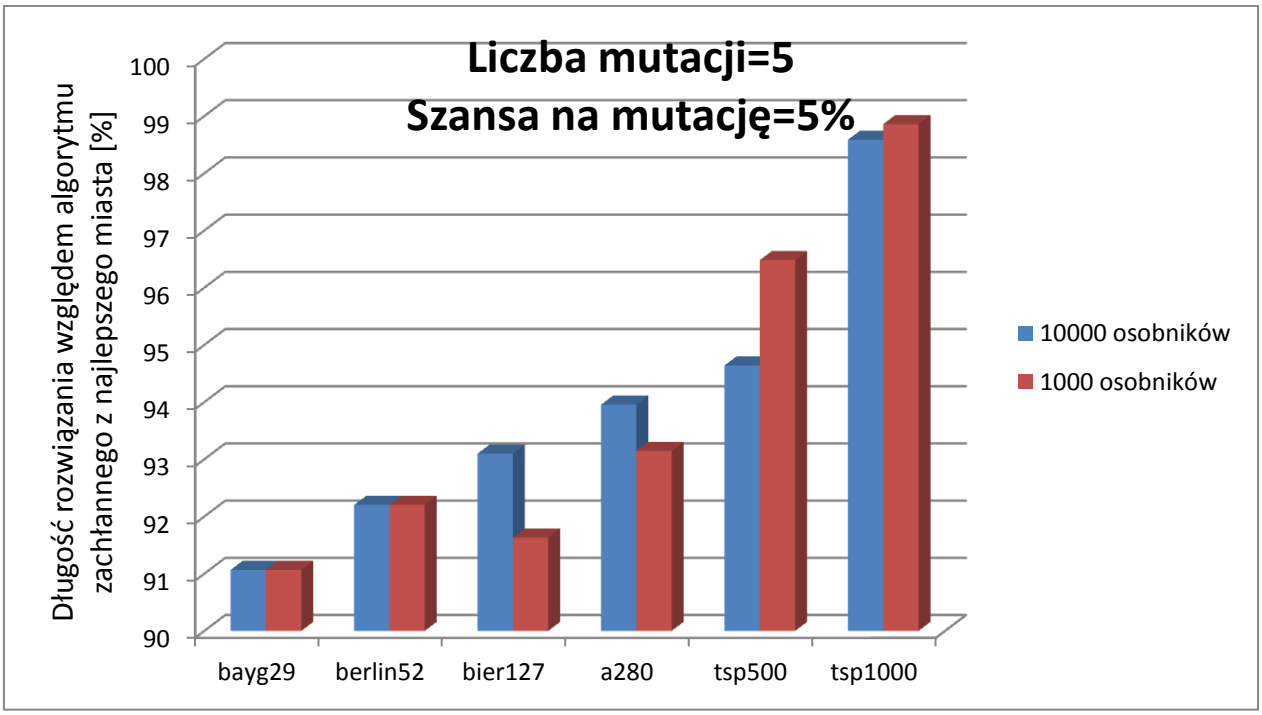

tsp500
100000
98000
96000
94000
92000
90000
88000
86000
długość trasy
0
5
10
20
szansa mutacji [%]
liczba mutacji 1
liczba mutacji 3
liczba mutacji 5
greedy z najlepszego wierzchołka


tsp1000
28500
28000
27500
27000
26500
26000
25500
długość trasy
0
5
10
20
szansa mutacji [%]
liczba mutacji 1
liczba mutacji 3
liczba mutacji 5
greedy z najlepszego wierzchołka

Porównanie wyników między 10000 osobników po 1000 iteracji i 1000 osobników po 10000 iteracjach:





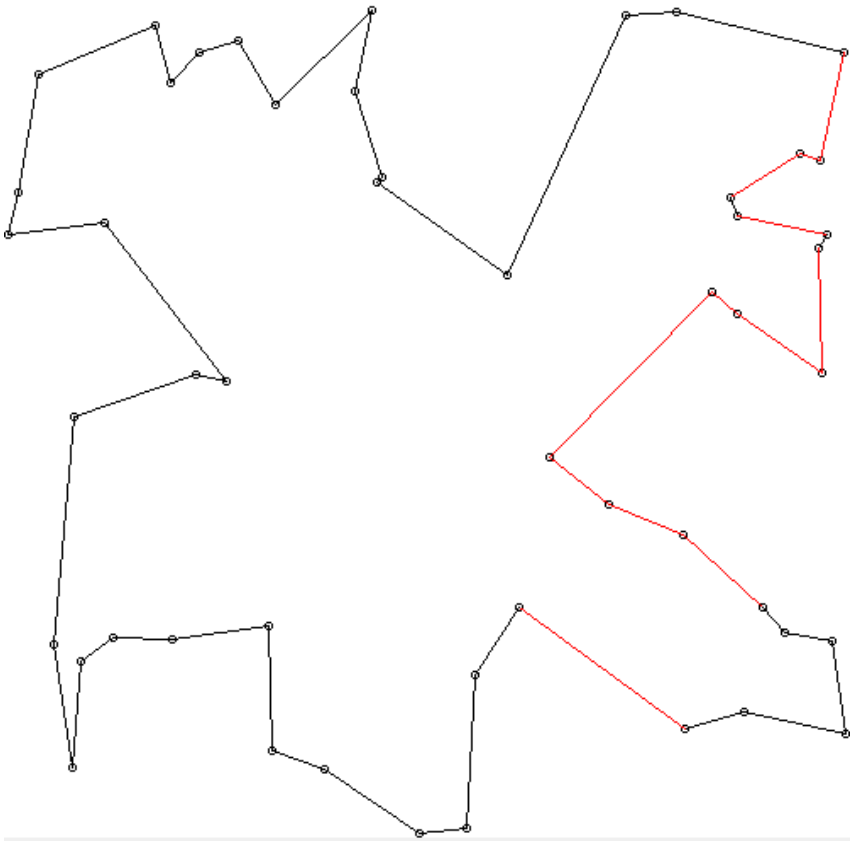

Liczba mutacji=5
Szansa na mutację=5%
Długość rozwiązania względem algorytmu zachłannego z najlepszego miasta [%]
100
99
98
97
96
95
94
93
92
91
90
bayg29
berlin52
bier127
a280
tsp500
tsp1000
10000 osobników
1000 osobników

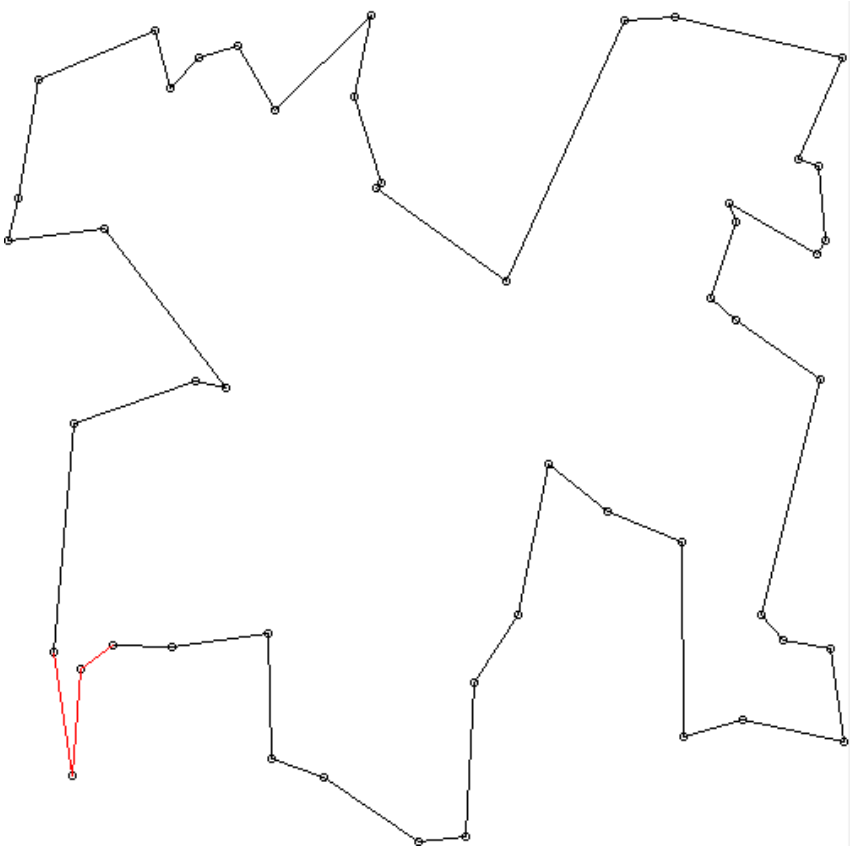
Strojenie i analiza wyników:

Porównano podejście wykorzystania 10000 osobników i 1000 osobników w populacji. Wyniki bywały porównywalne z tendencją do lepszych rezultatów przy 10000 osobnikach, jednak tym, co zaważyło na wybraniu wersji z 10000 osobników do dalszych testów była znacznie mniejsza losowość wyników przy takim podejściu. Przy 1000 osobnikach okazało się, że nawet tak mała instancja jak bayg29 może nie zwracać wartości optymalnej. Było to być może spowodowane niewystarczającą różnorodnością osobników.

Dalsze testy prowadzone były dla parametrów:
TOP_SURVIVORS = 50
BOTTOM_SURVIVORS = 0
duplicatesRemovalInterval = 2
populationSize = 10000
iterations = 1000
MAX_MUTATION = 1, 3, 5
MUTATION_PROBABILITY = 0, 5, 10, 20

Sterujemy więc częstością mutacji i liczbą przestawień podczas jej wystąpienia. Prawdopodobieństwo mutacji ustawione na 0% nie oznacza jej całkowitego wyeliminowania, ponieważ używana jest ona bezwarunkowo podczas usuwania duplikatów. Z tego powodu maksymalna długość mutacji nadal ma wpływ na działanie algorytmu, gdy prawdopodobieństwo mutacji jest ustawione na 0%. Mutowanie ma na celu zwiększenie różnorodności populacji i unikanie zjawiska zakleszczenia w lokalnym minimum. Przykładem mogą być tu rozwiązania własnowygenerowanej instancji na kolejnej stronie. Pierwszy obraz przedstawia optymalną ścieżkę o długości 11423, drugi podobną o długości 11424. Zdarzało się iż algorytm utykał w drugiej przedstawionej ścieżce, szczególnie dla nastawów z rzadką i krótką mutacją.





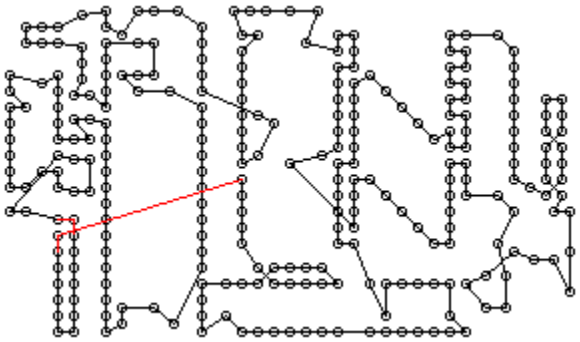
Po wynikach widać, że duża ilość mutacji zamiast zwiększać różnorodność populacji powoduje niszczenie rozwiązań. W ekstremalnych przypadkach jak długość mutacji równa 5, szansza na mutację równa 20%, algorytm genetyczny nie jest w stanie znaleźć niczego lepszego niż w pierwszej iteracji. Najlepsze wyniki zwracają nastawy: długość mutacji = 1, szansza na mutację 0%; dlugość mutacji = 1, szansa na mutację = 5%; długość mutacji = 3, szansa na mutację = 5%, a więc scenariusze, gdzie mutacja nie jest znaczącym elementem działania algorytmu.

Algorytm genetyczny zwraca jako wynik ścieżkę o długości około 90-92% długości ścieżki zwracanej przez algorytm zachłanny z najlepszego wierzchołka. Wyjątkiem jest tu instancja tsp1000, w której nie udało się zejść poniżej 94,41% wyniku algorytmu zachłannego. Prawdopodobnie wybrana metoda krzyżowania nie radzi sobie z tak dużą ilością miast.

Podsumowanie:

Ważnym elementem podczas tworzenia algorytmu genetycznego jest zapobiegnięcie utknięciu jego wyników w lokalnym minimum funkcji celu. Odpowiada za to mutacja osobników, czyli losowa zamiana kolejności odwiedzanych miast w ścieżce. Mutacja uruchamia się z pewną szansą dla każdego osobnika oprócz najlepszego przy każdej iteracji, a także co ustaloną liczbę iteracji dla wykrytych duplikatów. Przy odpowiednich nastawach mutacji pozwala to unikać sytuacji, w której algorytm stanie na rozwiązaniu, którego wynik jest trudny do stopniowej poprawy i potrzeba w dużym stopniu zmienionej ścieżki aby ruszyć dalej, jednocześnie nie niszcząc uzyskanych rozwiązań.

Pewną trudność w algorytmie genetycznym dla problemu komiwojażera jest skonstruowanie takiego krzyżowania, które będzie zwracało poprawnego osobnika z potencjałem na niższy wynik funkcji celu. Zastosowane skrzyżowanie spełnia te warunki, jednak ma problem z eliminacją skoków charakterystycznych dla algorytmu zachłannego. Przykładem może być wynik dla instancji a280 poniżej:



Ścieżka wygląda dość dobrze oprócz dwóch skoków, których algorytm nie zdołał się pozbyć od czasu wygenerowania rozwiązania zachłannego.