Politechnika Wrocławska

Projektowanie efektywnych algorytmów

Prowadzący: Dr inż. Jarosław Mierzwa

Zadanie 3

Temat: Implementacja i analiza efektywności algorytmu genetycznego (ewolucyjnego) dla wybranego problemu optymalizacji

Termin zajęć: PT 15:15 – 16:55

Autor sprawozdania: Karol Pastewski 252798

1. Wstęp teoretyczny

Rozpatrywanym problemem jest problem komiwojażera (*ang. travelling salesman problem, TSP*) w wersji asymetrycznej. Problem ten jest zagadnieniem optymalizacyjnym, polegającym na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Innymi słowami musimy znaleźć najkrótszą trasę przez wszystkie punkty, a następnie wrócić do początkowego punktu. W asymetrycznym TSP odległości między dwoma punktami mogą być różne.

1.1. Algorytm genetyczny

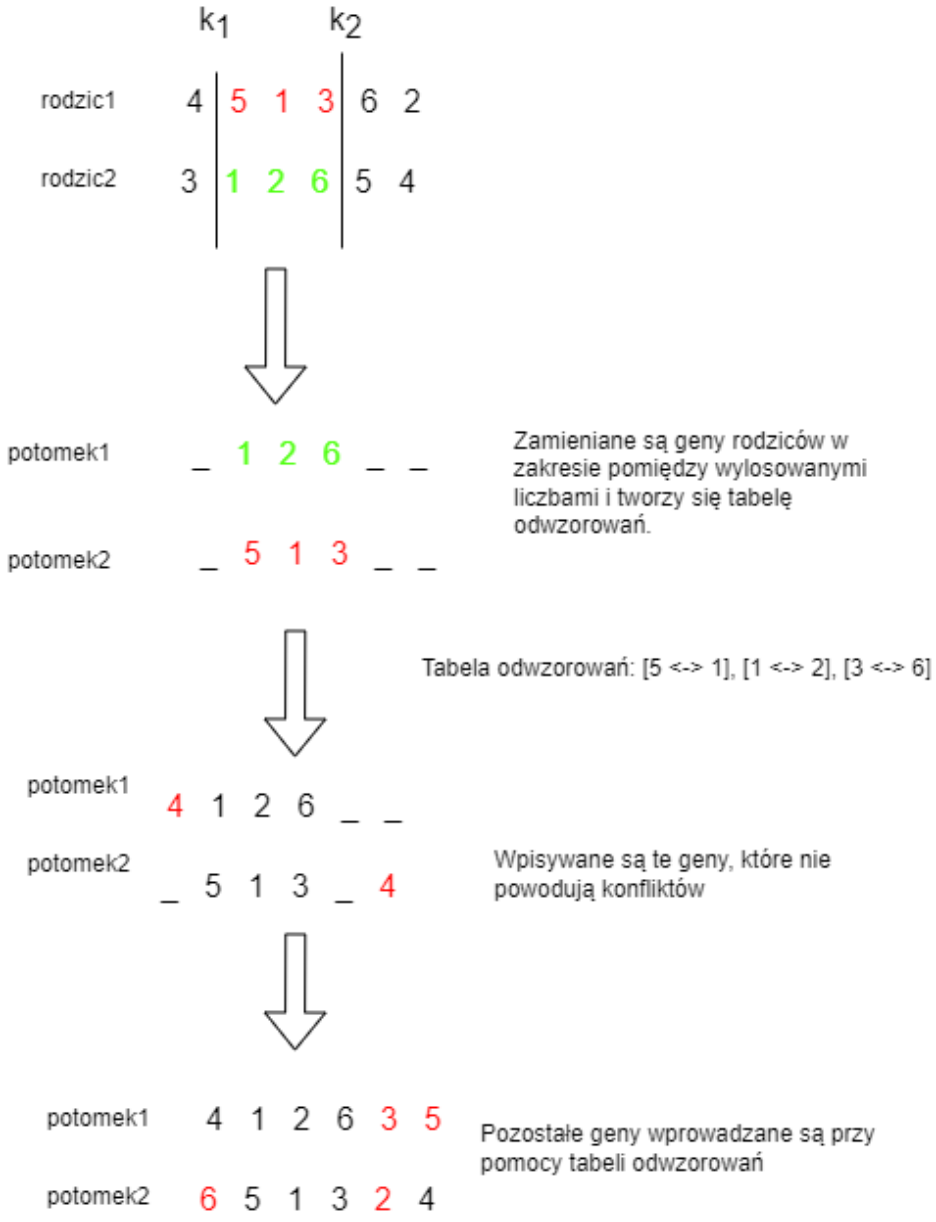
Algorytm genetyczny jest metaheurystyką inspirowaną procesami naturalnej selekcji występującej w przyrodzie. W tym algorytmie populacja osobników ewoluuje w kierunku optymalnego rozwiązania. Każdy osobnik jest opisany zestawem właściwości nazywanym chromosomem. Osobniki w kolejnych iteracjach algorytmu zostają poddane selekcji (wybranie osobników do krzyżowania), krzyżowaniu (wymiana materiału genetycznego pomiędzy parami osobników i stworzenie nowych) oraz mutacji (zmiana losowego genu (cechy osobnika)). Algorytm zatrzymuje się, gdy dojdzie do warunku stopu.

1.2. Metoda krzyżowania PMX

Krzyżowanie polega na wymianie materiału genetycznego pomiędzy losowo wybranymi parami osobników. W wyniku tej operacji powstają nowe osobniki, która powinny być lepiej przystosowane od swoich rodziców. Proces ten może zajść z prawdopodobieństwem p_c.

PMX (ang. Partially Mapped Crossover) – metoda krzyżowania, w której losowo znajdowane są dwa punkty k1 i k2 i w tym zakresie utworzonym przez te punkty zamienia się geny pomiędzy rodzicami. Pojedyncze zamiany genów następnie tworzą tabelę odwzorowań, która będzie później potrzebna. Kopiowane są te geny z rodzica, które nie powodują konfliktów, a te, które powodują, są wstawiane zgodnie z tabelą odwzorowań.

Przykład:

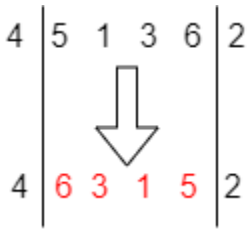


1.3. Metody mutacji

Mutacja polega na zamianie wartości losowo wybranego genu osobnika. Celem mutacji jest wprowadzenie zmienności chromosomów. Zachodzi z prawdopodobieństwem p_m.

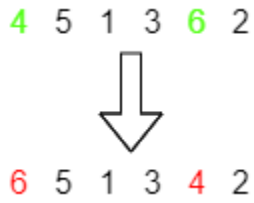
1.3.1. Inversion

Polega na losowym wybraniu podciągu miast i zamienienia ich kolejności. Przykład:



1.3.2. Transposition

Polega na losowym wybraniu dwóch miast i zamienieniu ich miejscami. Przykład:



1.4. Metoda selekcji turniejowa

Selekcja polega na wyborze z bieżącej populacji osobników tych, których podda się operacji krzyżowania oraz mutacji. Wybór odbywa się przy pomocy wybranej metody selekcji.

Wybrana metoda to selekcja turniejowa. Polega ona na tworzeniu turniejów z uczestnikami wybranymi z populacji. Z turnieju wybierany jest najlepszy osobnik, który trafia do puli osobników do krzyżowania i mutacji. W tej implementacji wielkość osobników zgłoszonych do turnieju wynosi 2% populacji. Liczba wykonywanych turniejów jest równa liczbie osobników w populacji. Wszystkie wybrane do turnieju osobniki są zwracane później do populacji (nawet ten wybrany), więc mamy do czynienia z selekcją turniejową ze zwracaniem.

2. Opis najważniejszych klas w projekcie

```
55  /*
56      Funkcja tworząca pierwszą populację.
57      Początkowa populacji jest wielkości populationSize, a osobniki są losowe.
58  */
59  void GeneticAlgorithm::setStartPopulation(std::vector<pathInfo>& population) {
60      auto rng = std::default_random_engine{ rd() };
61
62      std::vector<int> base;
63      base.reserve(N);
64
65      for (int i = 0; i <= N - 1; i++) {
66          base.push_back(i);
67      }
68
69      for (int i = 0; i < populationSize; i++) {
70          pathInfo newIndividual;
71          newIndividual.path = base;
72          std::shuffle(std::begin(newIndividual.path),
73                       std::end(newIndividual.path),
74                       rng);
75          newIndividual.cost = calcPathCost(newIndividual.path);
76          population.push_back(newIndividual);
77      }
78  }
```

Rysunek 1 Kod tworzący początkową populację wraz z opisem tej funkcji

Funkcja tworzy początkową populację losowych osobników. Pętla, która wykonuje się w linii 69 dla każdego osobnika kopiuje bazową ścieżkę (czyli ścieżka z miastami uporządkowanymi rosnąco), a następnie używa funkcji *shuffle*, która przetasowuje miasta w ścieżce. Dzięki temu, że w linii 60 domyślny generator liczb pseudolosowych jest inicjalizowany z *random_device* (*rd* oznacza właśnie *random_device*), to każda ścieżka będzie inna.

```
113 std::vector<GeneticAlgorithm::pathInfo>
114     GeneticAlgorithm::tournament(std::vector<pathInfo> population) {
115
116     auto rng = std::default_random_engine{ rd() };
117
118     std::vector<pathInfo> matingPool;
119     matingPool.reserve(population.size());
120
121     std::vector<pathInfo> tournamentPool;
122     int tournamentSize = 2 * population.size() / 100;
123     tournamentPool.reserve(tournamentSize);
124
125     for (int i = 0; i < population.size(); i++) {
126         tournamentPool.clear();
127         std::sample(population.begin(),
128                     population.end(),
129                     std::back_inserter(tournamentPool),
130                     tournamentSize,
131                     rng);
132         int minId = minPathId(tournamentPool);
133         matingPool.push_back(tournamentPool[minId]);
134     }
135
136     return matingPool;
137 }
```

Rysunek 2 Kod wykonujący metodę selekcji turniejową

Idea selekcji turniejowej została opisana w punkcie 1.4. Bardzo ważna w tej funkcji jest funkcja *sample* (linia 129), która umożliwia losowe wybranie próbki osobników z populacji i wstawienie ich do puli turniejowej.

```
165 std::tuple<GeneticAlgorithm::pathInfo, GeneticAlgorithm::pathInfo>
166     GeneticAlgorithm::crossoverPMX(pathInfo parent1, pathInfo parent2) {
167
168     auto rng = std::default_random_engine{ rd() };
169     std::uniform_int_distribution<int> distrubution(0, N - 1);
170     auto random = bind(distrubution, rng);
171
172     int k1, k2;
173     k1 = random();
174     do {
175         k2 = random();
176     } while (k1 == k2);
177     if (k2 < k1) {
178         int tmp = k1;
179         k1 = k2;
180         k2 = tmp;
181     }
182
183     pathInfo child1, child2;
184     child1.path.resize(N, INT_MAX);
185     child2.path.resize(N, INT_MAX);
186
187     std::map<int, int> map1{}, map2{};
188     std::map<int, int> child1Count{}, child2Count{};
```

Rysunek 3 Pierwsza część kodu dla metody krzyżowania PMX

Dla tej funkcji należy utworzyć generator liczb losowych z przedziału <0, N - 1>, który będzie potrzebny do znalezienia k1 i k2 (sprawdzane jest też czy k1 jest mniejsze od k2). child1 i child2 to są nasze osobniki potomne i w nich ustawiamy wartość każdego miasta w ścieżce na INT_MAX (linie 184 i 185). map1 i map2 są strukturami map, które magazynują klucze oraz wartości do tych kluczy. Są one potrzebne, bo służą za tabelę odwzorowań (dwie mapy, bo jedna jest odwrotnością drugiej). child1Count i child2Count przechowywują klucze odpowiadające miastom w potomku, a wartość tego klucza będzie zawsze 1. Idea jest taka, że podczas wpisywania miasta do potomka sprawdzane jest, czy w mapie potomka istnieje już dany klucz, jeżeli tak to należy te miasta wprowadzić za pomocą tabeli odwzorowań.

```
189  for (int i = k1; i ⩽ k2; i++) {
190      child1.path[i] = parent2.path[i];
191      child2.path[i] = parent1.path[i];
192      map1.insert({ parent1.path[i], parent2.path[i] });
193      map2.insert({ parent2.path[i], parent1.path[i] });
194      child1Count.insert({ child1.path[i], 1 });
195      child2Count.insert({ child2.path[i], 1 });
196  }
```

Rysunek 4 Druga część metody krzyżowania PMX

W tym fragmencie kopiujemy podciąg miast z rodziców do potomków na krzyż (szczegółowy opis w punkcie 1.2), wypełniamy tabelę odwzorowań oraz mapy oznaczające czy miasto jest już wpisane do danego potomka.

```
198  for (int i = 0; i < N; i++) {
199      if (i ⩾ k1 && i ⩽ k2) continue;
200
201      auto p1 = std::find(child1.path.begin(),
202                          child1.path.end(),
203                          parent1.path[i]);
204      if (p1 == child1.path.end()) {
205          child1.path[i] = parent1.path[i];
206          auto result = child1Count.insert({ child1.path[i], 1 });
207      }
208
209      auto p2 = std::find(child2.path.begin(),
210                          child2.path.end(),
211                          parent2.path[i]);
212      if (p2 == child2.path.end()) {
213          child2.path[i] = parent2.path[i];
214          auto result = child2Count.insert({ child2.path[i], 1 });
215      }
216  }
```

Rysunek 5 Trzecia część krzyżowania PMX

W tej części sprawdzane jest, czy kopiowane z rodzica miasto nie powoduje konfliktów (linie 201 – 204 i 212 - 215). Jeżeli nie, to możemy przekopiować miasto i oznaczyć dane miasto w mapie potomka, że zostało wybrane. Powtarza to jest dla każdego miejsca w ścieżce w wyłączeniem podciągu od *k1* do *k2*.

```
218  for (int i = 0; i < N; i++) {
219      if (i ≥ k1 && i ≤ k2) continue;
220
221      if (child1.path[i] == INT_MAX) {
222          auto search = map1.find(parent1.path[i]);
223          if (search ≠ map1.end()) {
224              child1.path[i] = search→second;
225              auto result = child1Count.insert({ child1.path[i], 1 });
226              while (result.second == false) {
227                  search = map1.find(child1.path[i]);
228                  child1.path[i] = search→second;
229                  result = child1Count.insert({ child1.path[i], 1 });
230              }
231          } else {
232              search = map2.find(parent1.path[i]);
233              child1.path[i] = search→second;
234              auto result = child1Count.insert({ child1.path[i], 1 });
235              while (result.second == false) {
236                  search = map2.find(child1.path[i]);
237                  child1.path[i] = search→second;
238                  result = child1Count.insert({ child1.path[i], 1 });
239              }
240          }
241      }
```

Rysunek 6 Czwarta część krzyżowania PMX

Dla wciąż pustych miejsc w ścieżce (pustych, czyli mających wartość *INT_MAX*) sprawdzamy, czy klucz oznaczający miasto istnieje w map1 albo w map2, dlatego, że po mapie można przeszukiwać tylko po kluczach i klucze muszą być unikalne. Tabela odwzorowań jest obustronna, dlatego dwie mapy (które są jednostronne) oznaczają jedną tabele odwzorowań. Jeżeli wybrało się już mapę, w której istnieje klucz miasta, to wstawiana jest wartość klucza (czyli miasta) do potomka. W momencie, gdy próbuje się dodać nowy klucz do *child1Count* i ta operacja zakończy się niepowodzeniem (linie 226 i 235), do tak długo, aż ta operacja się nie wykona, to znajdowany jest kolejny klucz (teraz ten klucz jest wartością poprzedniego klucza, który jest wprowadzanym miastem), wprowadzana jest wartość do potomka i próbuje się dodać nowy klucz do *child1Count*. Te same operacje wykonywane są dla drugiego potomka.

```
281 void GeneticAlgorithm::mutationInversion(pathInfo& individual) {
282     auto rng = std::default_random_engine{ rd() };
283     std::uniform_int_distribution<int> distrubution(0, N - 1);
284     auto random = bind(distrubution, rng);
285 
286     int k1, k2;
287     k1 = random();
288     do {
289         k2 = random();
290     } while (k1 == k2);
291     if (k2 < k1) {
292         int tmp = k1;
293         k1 = k2;
294         k2 = tmp;
295     }
296 
297     for (int i = 0; i <= (k2 - k1 + 1) / 2; i++) {
298         int tmp = individual.path[k1 + i];
299         individual.path[k1 + i] = individual.path[k2 - i];
300         individual.path[k2 - i] = tmp;
301     }
302 
303     individual.cost = calcPathCost(individual.path);
304 }
```

Rysunek 7 Kod dla metody mutacji Inversion

Funkcja ta znajduje dwa miasta, a następnie zamienia podciąg pomiędzy tymi miastami.

```cpp
void GeneticAlgorithm::mutationTransposition(pathInfo& individual) {
    auto rng = std::default_random_engine{ rd() };
    std::uniform_int_distribution<int> distrubution(0, N - 1);
    auto random = bind(distrubution, rng);

    int k1, k2;
    k1 = random();
    do {
        k2 = random();
    } while (k1 == k2);
    if (k2 < k1) {
        int tmp = k1;
        k1 = k2;
        k2 = tmp;
    }

    int tmp = individual.path[k1];
    individual.path[k1] = individual.path[k2];
    individual.path[k2] = tmp;

    individual.cost = calcPathCost(individual.path);
}
```

Rysunek 8 Kod dla metody mutacji Transposition

Funkcja znajduje dwa miasta i zamienia jest miejscami w ścieżce.

```
399  start = read_QPC();
400  while ((read_QPC() - start) / frequency < stop) {
401
402      population = tournament(population);
403
404      do {
405          int index = rng() % population.size();
406          parent1 = population[index];
407          population.erase(population.begin() + index);
408          index = rng() % population.size();
409          parent2 = population[index];
410          population.erase(population.begin() + index);
411
412          if (randomReal() ⩽ crossoverVar) {
413              std::tie(child1, child2) = crossoverPMX(parent1, parent2);
414          } else {
415              child1 = parent1;
416              child2 = parent2;
417          }
418          if (randomReal() ⩽ mutationVar) {
419              mutationMethod == 1 ? mutationInversion(child1) : mutationTransposition(child1);
420          }
421          if (randomReal() ⩽ mutationVar) {
422              mutationMethod == 1 ? mutationInversion(child2) : mutationTransposition(child2);
423          }
424          newGeneration.push_back(child1);
425          newGeneration.push_back(child2);
426      } while (population.size() > 0);
427      population = newGeneration;
428      oldPopulationSize = population.size();
429      newGeneration.clear();
430  }
```

Rysunek 9 Kod głównej pętli programu

Główna pętla programu kończy się po, określonym w kryterium stopu, czasie. Początkowo populacja przechodzi selekcję turniejową. Następnie dopóki w populacji znajduje się co najmniej jeden osobnik, to losowane są osobniki, które przejdą krzyżowanie (to czy krzyżowanie zaistnieje jest zależne od współczynnika krzyżowania, jeżeli krzyżowania nie odbędzie się, to rodzice kopiowani są do nowej generacji). Później oba potomne osobniki (albo skopiowani rodzice) przechodzą proces mutacji (podobnie jak przy krzyżowaniu, mutacja jest zależna od współczynnika mutacji). Kiedy już nie zostało już żadnych osobników do krzyżowania oraz mutacji, to nowa generacja zastępuje starą i pętla się powtarza.

3. Plan eksperymentu

Każdy eksperyment z odpowiednimi parametrami został powtórzony 10 razy.

Początkowo dla 3 wielkości populacji (100, 500, 1000), trzech kryteriach stopu (5, 15, 30 sekund – podobne kryteria zostały także wybrane w poprzednim sprawozdaniu dla Tabu Search) i dwóch metod mutacji zrobiono testy z parametrach:

- współczynnik mutacji – 0.01
- współczynnik krzyżowania – 0.8

Następnie po wybraniu najoptymalniejszej wielkości populacji przetestowano wpływ współczynnika mutacji na wyniki (dla trzech wartości: 0.02, 0.05, 0.1). Współczynnik krzyżowania zostaje 0.8.

Optymalne wartości dla rozpatrywanych plików:

- ftv47.atsp – 1776
- ftv170.atsp – 2755
- rbg403.atsp – 2465

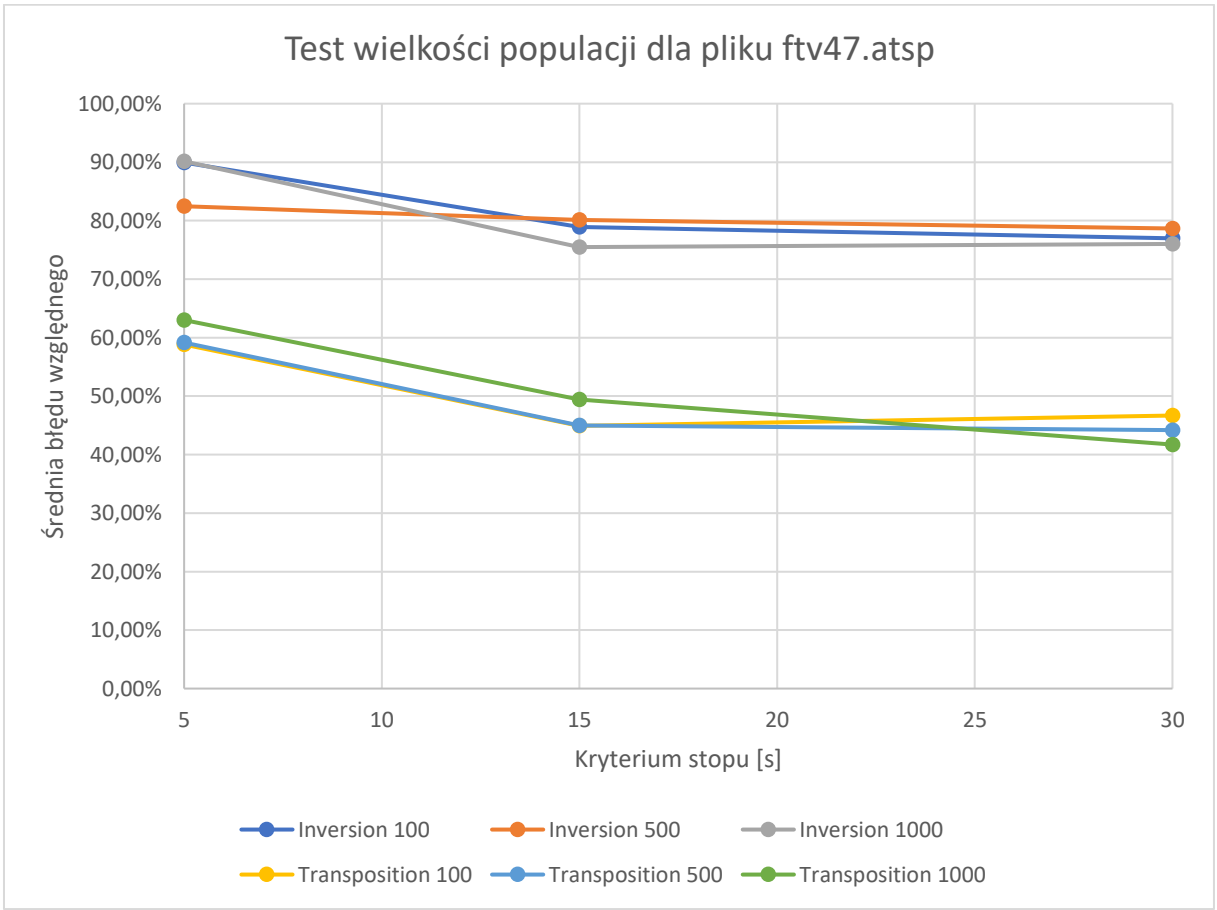
4. Wyniki eksperymentów

4.1. Znalezienie optymalnej populacji

Plik: ftv47.atsp

Metod mutacji	Wielkość populacji	Kryterium stopu [s]	Średni koszt	Średni błąd
Inversion	100	5	3373,2	89,93%
		15	3177,5	78,91%
		30	3143,2	76,98%
	500	5	3240,5	82,46%
		15	3199,4	80,15%
		30	3173	78,66%
	1000	5	3377	90,15%
		15	3116,5	75,48%
		30	3125,9	76,01%
Transposition	100	5	2820,6	58,82%
		15	2573,7	44,92%
		30	2605,5	46,71%
	500	5	2826,6	59,16%
		15	2575,2	45,00%
		30	2560,8	44,19%
	1000	5	2895	63,01%
		15	2654	49,44%
		30	2516,9	41,72%

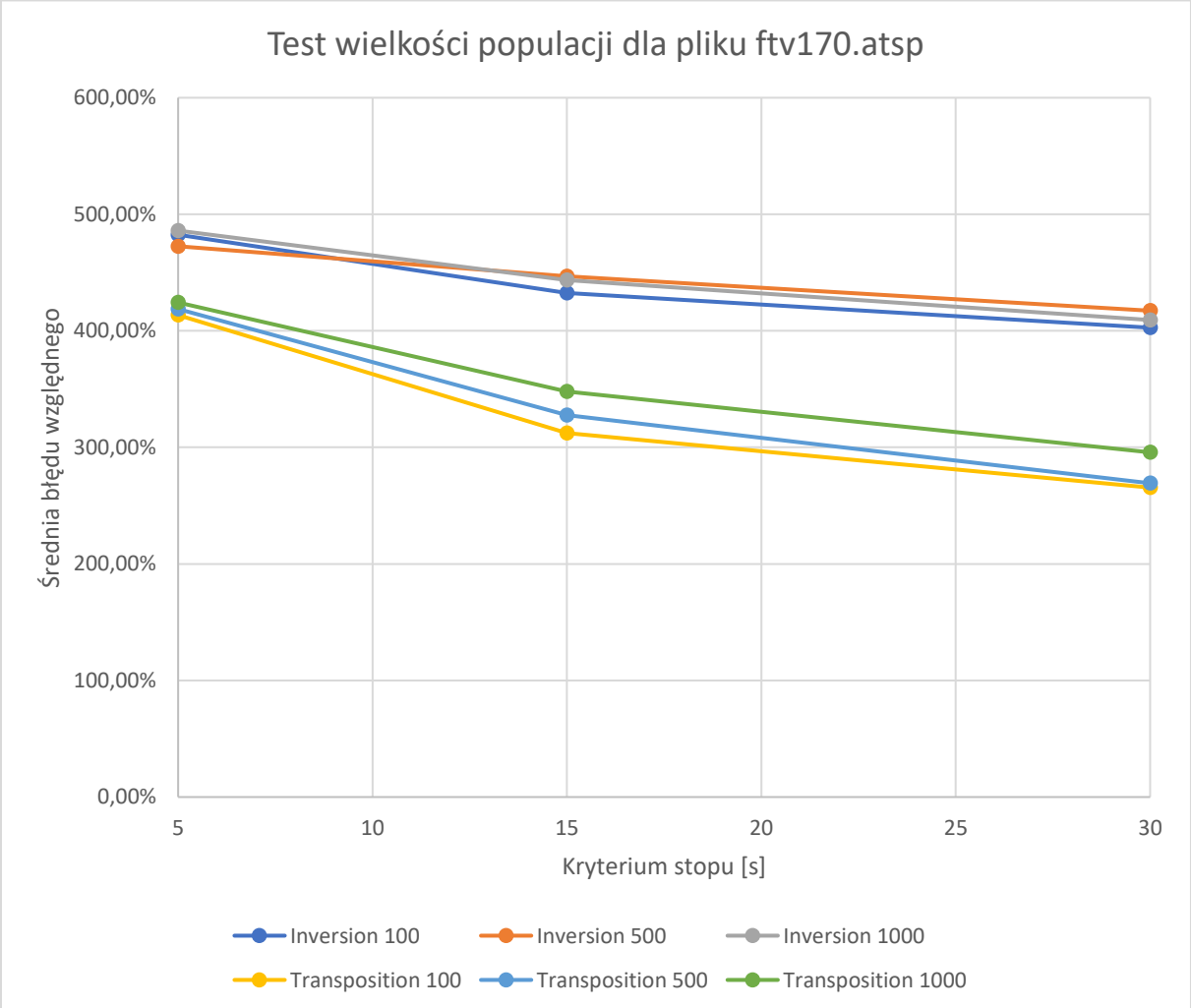
Wykres:



Plik: ftv170.atsp

Metod mutacji	Wielkość populacji	Kryterium stopu [s]	Średni koszt	Średni błąd
Inversion	100	5	16045,8	482,42%
		15	14667,8	432,41%
		30	13848,8	402,68%
	500	5	15774	472,56%
		15	15067,9	446,93%
		30	14250,8	417,27%
	1000	5	16143,9	485,99%
		15	14973,8	443,51%
		30	14030,4	409,27%
Transposition	100	5	14147,5	413,52%
		15	11356,4	312,21%
		30	10069,3	265,49%
	500	5	14290,3	418,70%
		15	11782	327,66%
		30	10172,5	269,24%
	1000	5	14445	424,32%
		15	12343,3	348,03%
		30	10903	295,75%

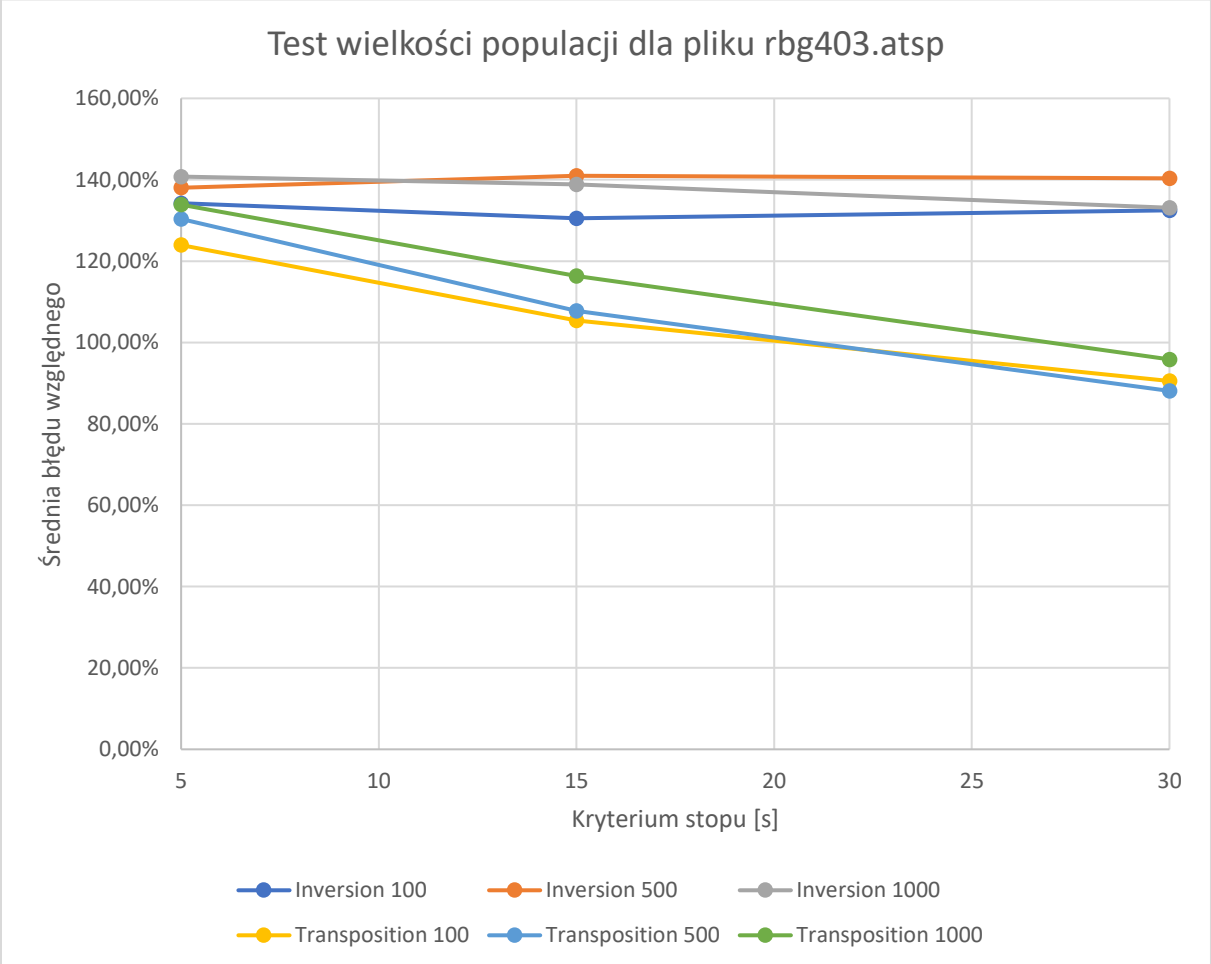
Wykres:



Plik: rbg403.atsp

Metod mutacji	Wielkość populacji	Kryterium stopu [s]	Średni koszt	Średni błąd
Inversion	100	5	5773,8	134,23%
		15	5682,4	130,52%
		30	5730,7	132,48%
	500	5	5867,5	138,03%
		15	5940,1	140,98%
		30	5925,1	140,37%
	1000	5	5935,1	140,77%
		15	5887,1	138,83%
		30	5746,3	133,12%
Transposition	100	5	5520,2	123,94%
		15	5063,2	105,40%
		30	4696,7	90,54%
	500	5	5678,2	130,35%
		15	5121,7	107,78%
		30	4636,6	88,10%
	1000	5	5764,9	133,87%
		15	5332,6	116,33%
		30	4827,6	95,85%

Wykres:



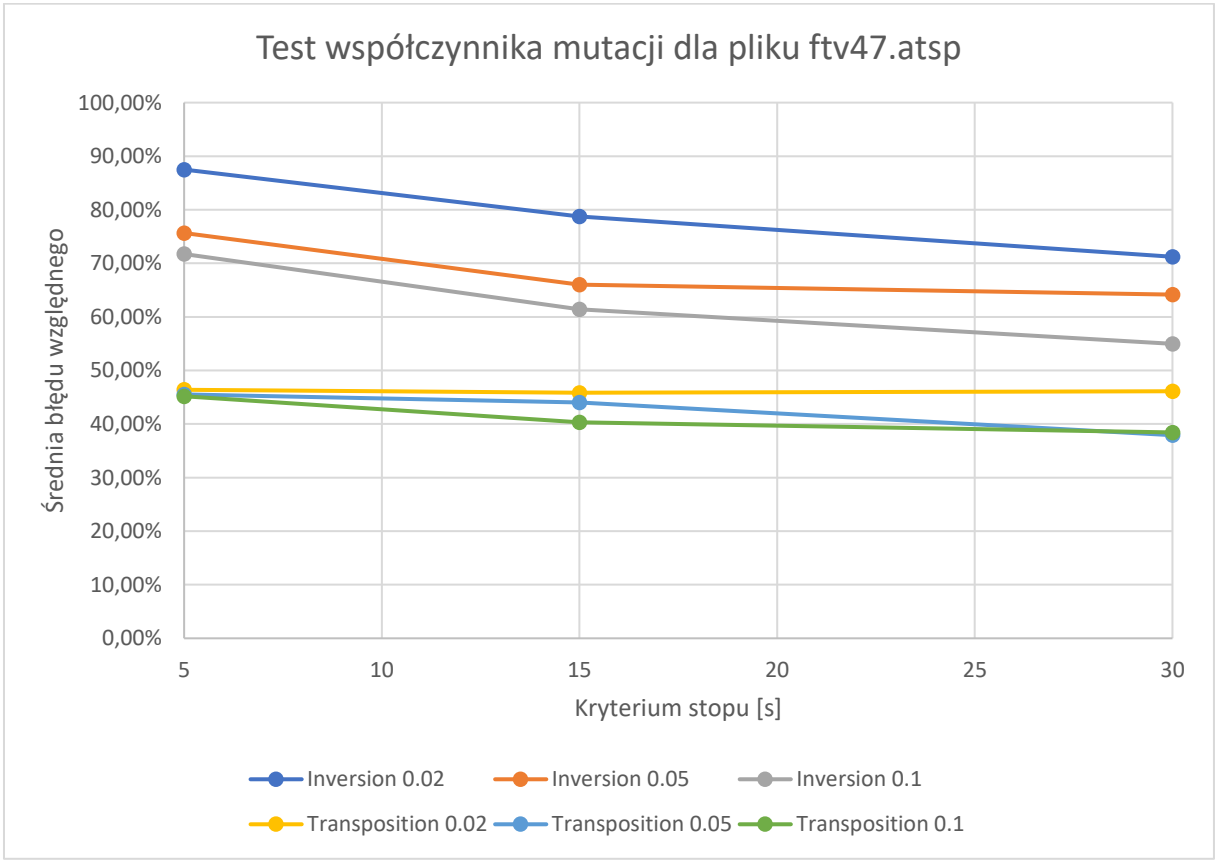
4.2.Testowanie wpływu współczynnika mutacji

Wielkość populacji dla tego testowana będzie wynosiła 100 (najlepsza z poprzedniego testowania).

Plik: ftv47.atsp

Metod mutacji	Wielkość populacji	Kryterium stopu [s]	Średni koszt	Średni błąd
Inversion	0.02	5	3330	87,50%
		15	3174,6	78,75%
		30	3041	71,23%
	0.05	5	3119,7	75,66%
		15	2948,5	66,02%
		30	2915,5	64,16%
	0.1	5	3050	71,73%
		15	2867,1	61,44%
		30	2752,3	54,97%
Transposition	0.02	5	2599,9	46,39%
		15	2590,2	45,84%
		30	2594,7	46,10%
	0.05	5	2584,6	45,53%
		15	2557,5	44,00%
		30	2449,4	37,92%
	0.1	5	2578,4	45,18%
		15	2491,9	40,31%
		30	2458,6	38,43%

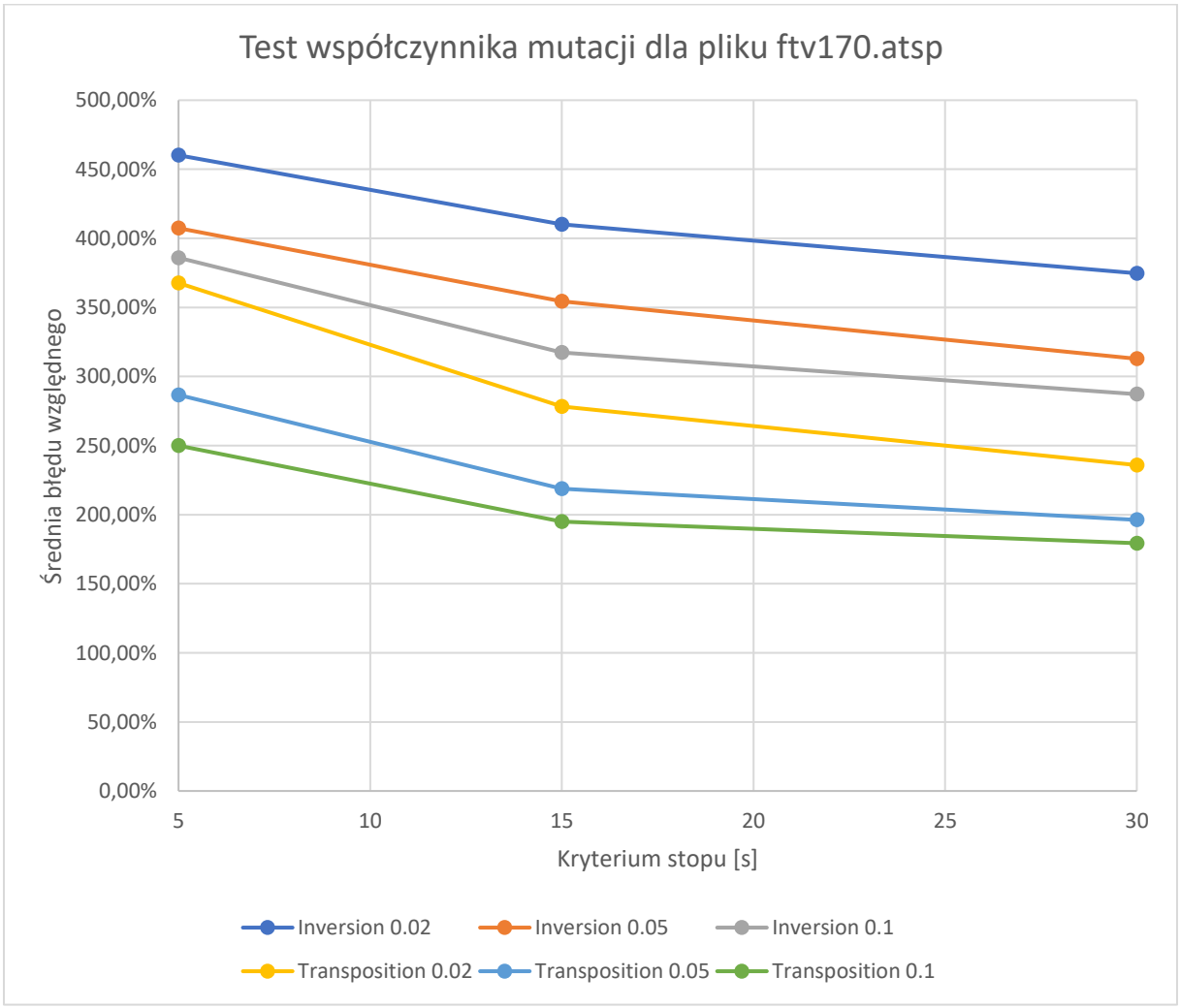
Wykres:



Plik: ftv170.atsp

Metod mutacji	Wielkość populacji	Kryterium stopu [s]	Średni koszt	Średni błąd
Inversion	0.02	5	15432,2	460,15%
		15	14056,5	410,22%
		30	13078	374,70%
	0.05	5	13978,4	407,38%
		15	12516,7	354,33%
		30	11373,6	312,83%
	0.1	5	13388,2	385,96%
		15	11501,1	317,46%
		30	10666,8	287,18%
Transposition	0.02	5	12884,5	367,68%
		15	10423,2	278,34%
		30	9253,1	235,87%
	0.05	5	10652,8	286,67%
		15	8781,1	218,73%
		30	8160,8	196,22%
	0.1	5	9641,5	249,96%
		15	8127,3	195,00%
		30	7695,7	179,34%

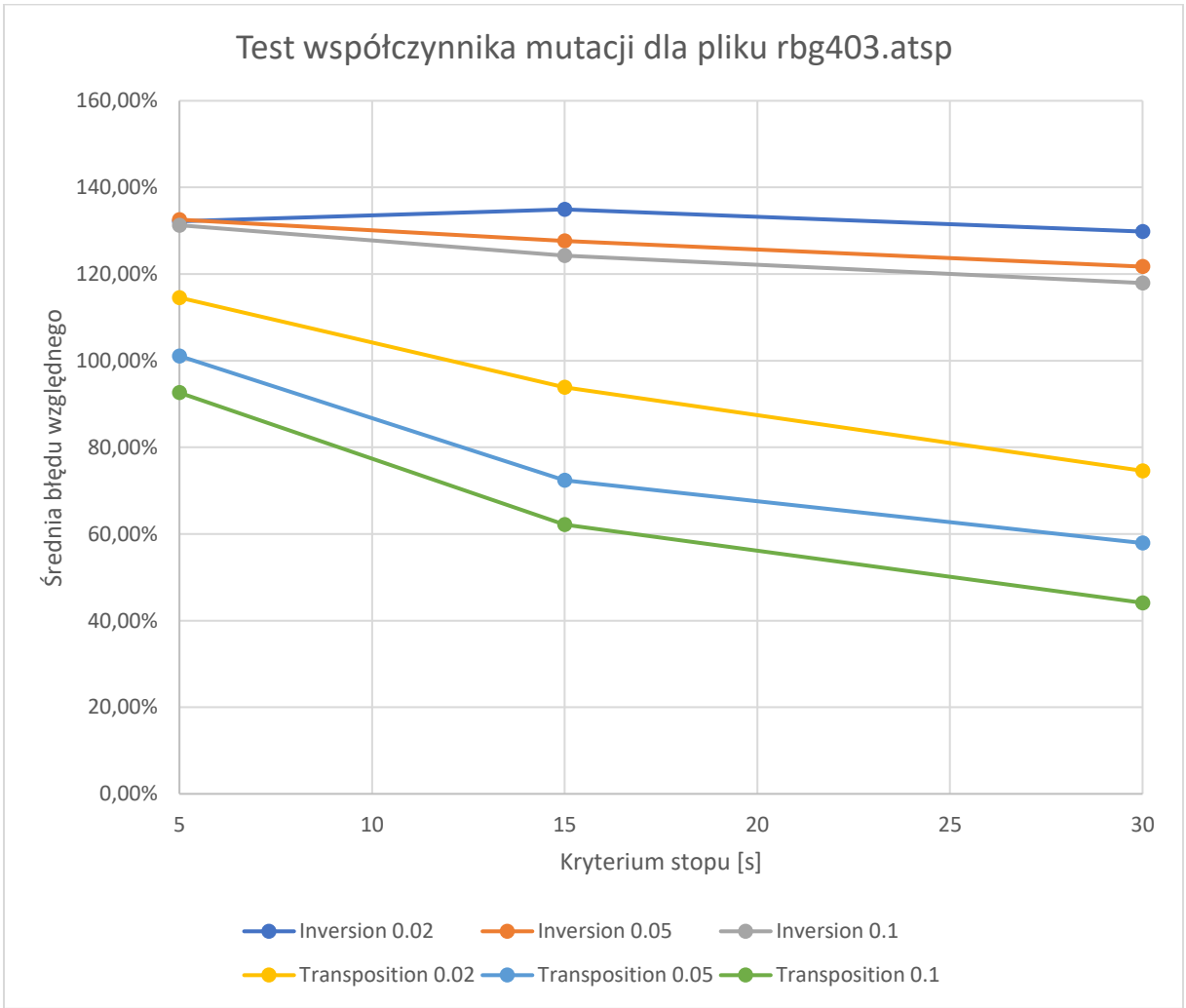
Wykres:



Plik: rbg403.atsp

Metod mutacji	Wielkość populacji	Kryterium stopu [s]	Średni koszt	Średni błąd
Inversion	0.02	5	5723	132,17%
		15	5790,8	134,92%
		30	5664,8	129,81%
	0.05	5	5732,7	132,56%
		15	5611,4	127,64%
		30	5465,3	121,72%
	0.1	5	5700,9	131,27%
		15	5527,9	124,26%
		30	5371,9	117,93%
Transposition	0.02	5	5288,9	114,56%
		15	4778,2	93,84%
		30	4303,4	74,58%
	0.05	5	4956,9	101,09%
		15	4249,6	72,40%
		30	3892,2	57,90%
	0.1	5	4747,8	92,61%
		15	3996,9	62,15%
		30	3552,2	44,11%

Wykres:



4.3. Porównanie najlepszych wyników Tabu Search i algorytmu genetycznego

Tabu Search:

ftv47.atsp:

- Koszt: 1968
- Ścieżka: 14 16 45 39 21 20 37 38 0 25 40 47 26 1 11 18 13 34 23 12 32 7 31 24 4 29 30 5 6 10 8 9 33 3 27 28 2 43 42 22 41 19 44 15 17 46 36 35 14

ftv170.atsp:

- Koszt: 3475
- Ścieżka: 25 150 160 151 152 142 141 134 131 113 164 127 126 125 124 121 120 122 123 162 102 103 117 118 119 129 128 130 135 138 139 140 6 7 8 9 10 76 74 75 11 12 18 19 20 37 22 23 26 27 28 29 30 31 33 34 156 40 39 38 35 49 170 73 77 1 2 0 81 80 79 82 78 72 71 60 50 51 52 53 43 55 54 58 59 61 68 67 167 70 87 85 86 83 84 69 66 63 64 56 57 62 65 88 153 154 89 90 91 94 96 97 99 98 95 92 93 166 108 107 106 105 165 163 100 101 104 110 109 114 115 116 136 137 147 148 149 161 14 13 21 32 158 36 157 41 155 42 45 44 46 47 48 168 3 4 5 133 169 111 112 132 143 144 146 145 159 16 17 24 15 25

rbg403.atsp

- Koszt: 2592
- Ścieżka: 20 23 14 62 13 205 204 24 36 32 274 46 33 376 68 38 270 138 18 402 287 83 43 34 295 50 56 394 225 58 8 6 64 3 2 61 107 386 47 112 128 137 65 397 260 176 286 273 272 28 232 374 126 10 27 11 310 29 77 31 52 40 39 78 226 147 79 69 245 81 22 21 82 249 130 172 26 182 152 76 160 15 333 267 281 221 67 66 60 44 88 96 94 90 72 57 164 25 352 92 51 49 37 247 256 120 392 351 293 150 213 41 208 312 35 364 303 71 239 70 349 115 114 353 263 99 389 318 97 206 30 131 355 168 242 117 180 359 307 143 59 340 129 103 209 278 122 119 214 192 189 104 9 384 383 203 146 144 105 391 154 111 369 139 4 264 328 317 108 255 102 265 275 210 110 326 258 207 45 363 289 162 48 358 327 170 284 290 125 74 216 200 198 155 323 305 215 309 252 191 187 234 228 224 178 183 17 12 217 285 282 357 322 370 121 132 243 298 291 315 161 148 325 134 306 319 135 280 279 136 308 211 98 254 223 344 345 173 354 342 266 141 116 218 238 229 393 149 347 401 400 385 300 236 227 381 220 197 171 63 297 86 240 219 19 337 181 346 338 331 378 398 396 156 390 159 7 388 360 194 324 212 169 241 330 343 85 248 348 329 294 387 84 257 42 177 253 158 1 93 145 193 87 283 277 341 124 269 166 199 91 367 75 262 261 301 195 201 118 174 250 202 190 100 186 53 73 366 321 379 109 375 371 372 356 127 268 16 246 95 271 350 304 5 222 185 231 288 251 314 235 113 395 332 296 233 380 362 244 80 230 382 320 184 151 101 336 292 153 368 142 276 259 188 299 302 89 311 55 106 165 373 140 334 196 0 335 163 365 133 361 175 179 316 157 339 237 123 399 377 167 54 313 20

Algorytm genetyczny:

ftv47.atsp

- Koszt: 2236
- Ścieżka: 24 4 3 33 27 8 11 10 6 29 30 31 5 32 7 23 12 13 34 35 14 15 16 18 17 46 36 45 39 20 0 25 37 38 47 19 44 21 40 26 1 9 2 43 22 41 42 28 22

ftv170.atsp

- Koszt: 7070
- Ścieżka: 12 19 39 40 45 168 72 78 85 86 93 94 122 121 124 130 133 134 6 9 8 14 15 17 18 11 10 75 20 23 160 151 37 50 53 46 69 66 63 56 55 54 58 62 61 65 57 64 42 48 49 170 60 89 88 153 83 84 71 2 4 5 109 106 105 98 97 165 113 126 145 144 140 108 166 107 104 135 128 131 154 90 51 59 68 67 167 70 87 92 91 163 99 123 162 120 102 117 103 95 96 100 101 52 43 44 34 30 28 29 41 155 156 47 73 7 159 16 21 22 25 149 148 129 146 152 13 32 31 35 36 110 114 164 127 125 138 139 169 0 81 80 79 82 74 158 24 150 112 132 111 3 143 147 137 136 115 116 118 119 1 77 38 157 33 27 26 161 142 141 76 12

rbg403.atsp

- Koszt: 3474
- Ścieżka: 165 232 261 175 276 49 296 99 201 368 372 69 245 248 106 359 78 382 249 13 144 190 326 102 237 182 280 316 256 17 392 131 355 141 21 42 18 195 44 304 394 14 225 108 351 320 128 228 93 34 295 110 136 272 356 322 357 274 72 266 129 120 3 47 113 317 133 331 323 168 176 288 205 224 179 292 243 210 222 250 279 217 116 153 48 332 267 107 385 67 387 311 39 132 204 335 396 54 345 60 98 23 216 339 123 164 287 281 220 197 15 105 19 318 383 104 145 29 85 194 324 230 90 196 189 388 240 188 178 275 31 344 308 35 312 260 299 87 264 226 154 206 150 84 395 174 185 251 358 257 238 229 353 367 75 10 180 376 202 297 283 170 109 301 255 247 100 379 125 74 193 103 163 244 80 239 79 96 114 401 354 200 268 16 147 58 124 269 235 121 291 362 209 278 122 302 25 55 50 56 5 300 338 236 330 89 148 325 32 112 2 83 366 340 142 199 334 68 333 241 328 370 391 258 336 321 227 212 111 369 263 166 203 173 158 1 207 282 46 65 242 70 384 246 95 94 183 157 161 24 36 262 181 346 81 22 82 208 64 223 309 88 91 8 6 130 135 315 160 365 86 143 37 152 284 329 294 286 259 374 41 364 303 377 177 92 347 341 186 277 352 211 380 289 97 313 7 172 26 343 233 360 33 187 171 399 198 285 0 140 11 371 327 252 191 77 138 213 28 397 118 66 215 115 126 271 350 234 231 219 156 390 373 30 375 167 290 402 155 378 400 265 63 57 127 192 162 151 62 298 12 254 134 306 319 361 139 4 337 169 27 393 159 363 398 221 389 38 61 348 71 137 293 314 59 76 273 386 270 43 9 146 381 307 310 214 53 305 342 52 119 73 218 349 149 51 20 184 45 101 40 253 117 165

5. Wnioski

W eksperymentach ze znalezieniem najoptymalniejszej wielkości populacji wynika, że wszystkie 3 przyjęte wielkości populacji dawały mocno zbliżone rezultaty, pomimo tego, że różnica pomiędzy wszystkimi 3 wielkościami była względnie duża. Do dalszego testowania wybrano wielkość populacji 100, bo przez większość czasu w większości plikach dawała najmniejszy błąd względny. Należy jednak zaznaczyć, że możliwa byłaby zmiana wyników, gdyby kryterium stopu zostałoby ustawione na wyższe (im większy jest plik, tym bardziej jest to prawdopodobne).

Zmiana współczynnika mutacji na większą przyniosła znaczącą poprawę w minimalizacji błędu względnego. Największa testowana wartość, czyli 0.1 dla każdego pliku oraz obu zaimplementowanych metod mutacji dawała najlepsze wyniki.

Dwie zaprezentowane metody mutacji znacząco zmieniają wyniki testów. Pierwsza metoda, czyli Inversion wypada gorzej we wszystkich eksperymentach od metody Transpositon. Należy także wskazać, że pierwsza metoda w niektórych eksperymentach (np. testy wielkości populacji dla plików ftv47.atsp i rbg403.atsp) wraz z zwiększaniem kryterium stopu, metoda ta nie poprawia swoich wyników. Można więc stwierdzić, że metoda Inversion, nawet z większą ilością czasu, nie będzie optymalniejsza od metody Transposition.

Porównanie Tabu Search z algorytmem genetycznym wypada na korzyść tego pierwszego. Oba algorytmy były testowane z maksymalnym kryterium stopu ustawionym na 30 s, więc czas przydzielony obu algorytmom jest podobny. Algorytm genetyczny najbliżej wyniku Tabu Search jest przy najmniejszym pliku, ale dla na przykład średniego algorytm genetyczny znajduje nawet dwukrotnie gorszy wynik.

6. Źródła

- https://en.wikipedia.org/wiki/Genetic_algorithm
- https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)
- https://en.wikipedia.org/wiki/Tournament_selection
- https://wpmedia.wolfram.com/uploads/sites/13/2018/02/03-5-5.pdf
- http://www.imio.polsl.pl/Dopobrania/Cw%20MH%2007%20(TSP).pdf
- http://aragorn.pb.bialystok.pl/~wkwedlo/EA5.pdf
- Wykłady dr inż. Tomasza Kapłona z Projektowania efektywnych algorytmów