Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Poniedziałek 15:25*
Temat *Algorytmy ewolucyjne*	**Problem** *TSP*
Skład grupy *241385 Radosław Lis*	**Nr grupy** *-*
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *4 lutego 2020*

Spis treści

1 Opis problemu

Problem komiwojażera (ang. **TSP** - *Travelling Salesman Problem*) to jedno z najbardziej powszechnych zagadnień z dziedziny algorytmiki. W celu zobrazowania zagadnienia, należy wyobrazić sobie komiwojażera, który podróżuje między miastami w prowincji, sprzedając swoje towary. Wyrusza ze swojego domu, po czym jego trasa przebiega dokładnie jeden raz przez każde miasto w prowincji, aż na końcu wraca do domu rodzinnego. Rozwiązanie problemu to znalezienie odpowiedniej drogi i kosztów podróży (odległość, czas itp.), która maksymalnie je zminimalizuje.

Z matematycznej perspektywy wygląda to tak, że miasta są wierzchołkami grafu, a łączace je trasy to krawędzie z odpowiednimi wagami. Jest to graf pełny, ważony oraz może być skierowany - co tworzy problem *asymetryczny*. Rozwiązanie problemu komiwojażera sprowadza się do znalezienia właściwego - o najmniejszej sumie wag krawędzi - *cyklu Hamiltona*, czyli cyklu przechodzącego przez każdy wierzchołek grafu dokładnie jeden raz. Przeszukanie wszystkich cykli (czyli zastosowanie metody *Brute Force*) nie jest optymalną metodą, jako że prowadzi do wykładniczej złożoności obliczeniowej - $O(n!)$, dla której problemy o dużym n są traktowane jako nierozwiązywalne. Klasyfikuje to problem komiwojażera jako *problem NP-trudny*, czyli niedający rozwiązania w czasie wielomianowym. To powoduje konieczność skorzystania z tzw. *algorytmów heurystycznych* bądź *metaheurystycznych* (bardziej ogólnych), a konkretnie w naszym przypadku algorytmów ewolucyjnych - *algorytmu genetycznego* oraz bazujących na nim - *algorytmu memetycznego* oraz *algorytmu wyspowego*.

2 Algorytm genetyczny

Algorytm genetyczny jest to rodzaj heurystyki, należy do grupy algorytmów ewolucyjnych, gdyż jego sposób działania jest zaczerpnięty z ewolucji biologicznej. Ewolucja rozpoczyna się od utworzenia początkowej populacji, stosowanie operatorów krzyżowania (rozmnażania) i mutacji (wpływ otoczenia na osobnika, np. wirus) tak aby dojść do rozwiązania jak najbliższego optymalnemu.

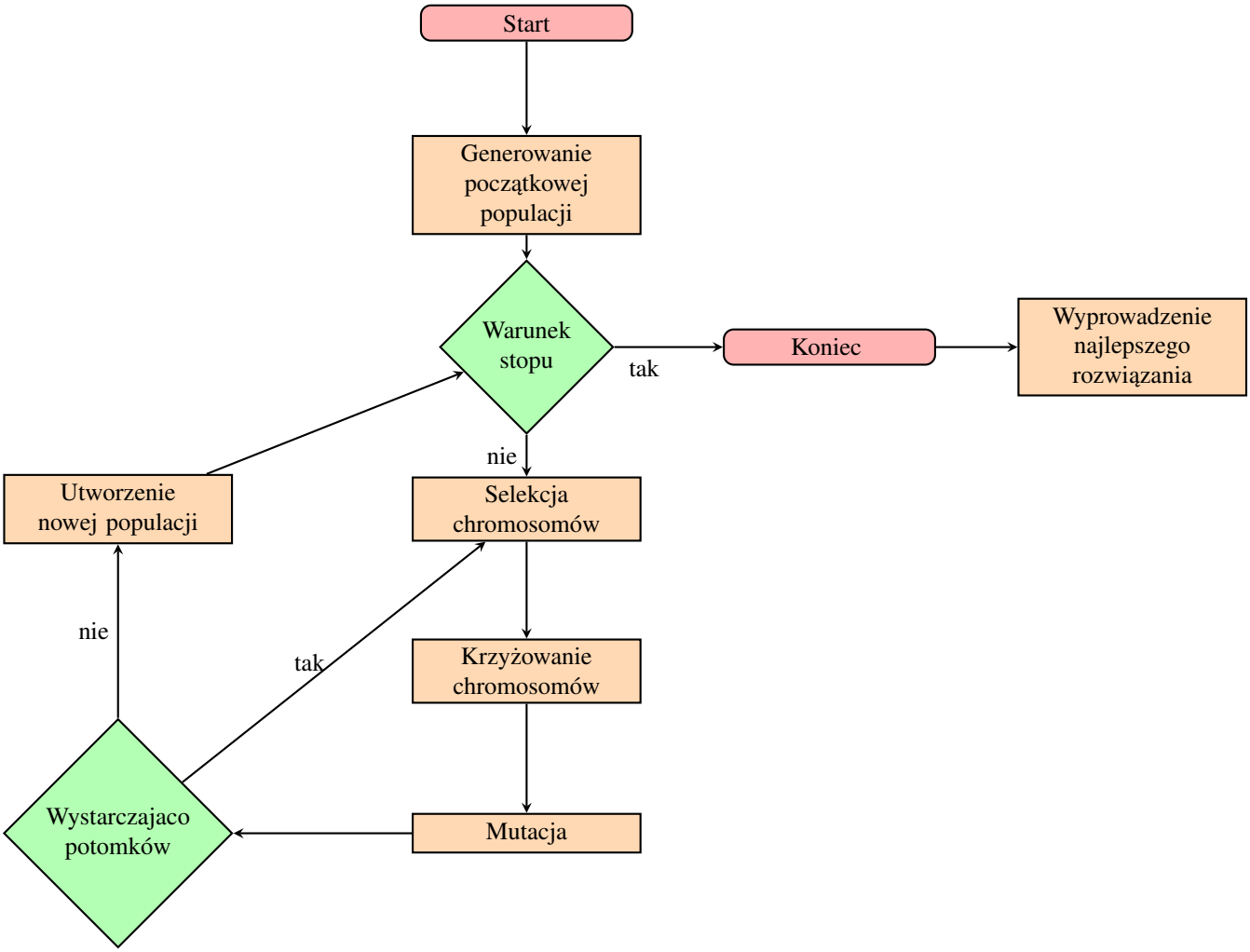
2.1 Opis i schemat działania algorytmu

Algorytm rozpoczyna się od wygenerowania wybranej liczby osobników (w ewolucji *chromosomów*), którzy są reprezentowani jako ścieżka (przykładowo formatu 0-1-3-7-...-4-6-0) z zapisanym na końcu kosztem ścieżki. Wszystkie wygenerowane osobniki stanowią obecną populację. Następnie w wyniku selekcji zostaje wybranych dwóch „rodziców", którzy za pomocą operatora krzyżowania tworzą nowych osobników, którzy po przejściu przez opcjonalny proces mutacji dodawani są do *„chwilowej populacji"*. Gdy *chwilowa populacja* osiągnie wystarczający rozmiar osobniki z niej są przenoszone do właściwej populacji zgodnie z ideą tzw. *elitaryzmu*, czyli zastrzeżenia, że w populacji zostaje dana ilość najlepszych osobników ze starej populacji. Warunkiem zakończenia może być dana liczba pokoleń (iteracji) lub określony czas - w przypadku przedstawionej implementacji jest to czas.

2.2 Generowanie początkowej populacji

Generowanie początkowej populacji odbywa się w sposób deterministyczno-probabilistyczny. Pierwszy osobnik generowany jest sposobem z algorytmu *Branch&Bound*, który metodą wgłąb dochodzi do liścia, a zarazem początkowego rozwiązania. Drugi osobnik generowany jest standardowym *algorytmem zachłannym*, połowa pozostałych do wygenerowania osobników jest natomiast generowana algorytmem losowo-zachłannym, który na początku losuje n (przekazanych jako parametr) wierzchołków, zaś wybór następnych wierzchołków odbywa się zachłannie. Następna połowa pozostałych do wygenerowania osobników generowana jest hybrydową metodą - losowanie n wierzchołków, a następnie procedura algorytmu generującego pierwszego osobnika.

Taki sposób generowania osobników sprawdził się znacznie lepiej niż generowanie wszystkich osobników losowo. W przeprowadzonych wstępnych testach algorytm genetyczny z przedstawioną wyżej procedurą zwracał końcowy wynik z ponad 2-krotnie mniejszym błędem względnym.



Rysunek 1: Schemat *Algorytmu genetycznego*

2.3 Selekcja rodziców

W literaturze można spotkać wiele rodzajów selekcji, jednak zdecydowanie najczęściej spotykane to *selekcja turniejowa*, *selekcja rankingowa* oraz *selekcja koła ruletki*. Ich wspólną cechą jest to, że faworyzują osobników lepszych, o mniejszym koszcie ścieżki. W implementacji odszedłem od standardowego tworzenia populacji rodzicielskiej, w tym przypadku jest nią stara populacja, co zapewnia szanse na rozmnożenie każdemu osobnikowi.

2.3.1 Selekcja turniejowa

Selekcja turniejowa opiera się na prostym algorytmie, który losuje k osobników (najczęściej 2) i wybiera najlepszego z nich:

Listing 1: Algorytm selekcji turniejowej

```
vector <unsigned> Genetic::tournamentSelection(vector <vector <unsigned>> pop) {
  vector <unsigned> best,ind;
  random_device randomSrc;
  default_random_engine randomGen(randomSrc());
  uniform_int_distribution<> indRand(0, pop.size() - 1);
  int k = 2;

  for (int i = 1; i <= k; i++) {
    if (i==1)
    best = pop.at(indRand(randomGen));

    else {
      ind = pop.at(indRand(randomGen));
      if (ind.at(matrixSize + 1) < best.at(matrixSize+1))
        best = ind;
    }
  }
  return best;
}
```

2.3.2 Selekcja rankingowa

Selekcja rankingowa wprowadza do implementacji dodatkowy element - tzw. funkcję zdatności (ang. *fitness value*). Idea selekcji opiera się na rozdzieleniu odpowiednich prawdopodobieństw wybrania poszczególnych osobników na podstawie rankingu, który szereguje osobników w kolejności od ścieżki o najmniejszej funkcji celu do największej funkcji celu. Jako, że *TSP* to problem minimalizacji, to należy tak zdefiniować funkcję zdatności, żeby wyższy współczynnik oznaczał lepszego osobnika. W tym celu został zastosowany wzór:

$$fitness\ value = \frac{s-r}{s\cdot(s-1)}$$

gdzie s = rozmiar populacji, r=miejsce w rankingu

Następnie zgodnie z wartością *funkcji zdatności* przyporządkowane jest właściwe prawdopodobieństwo, a na końcu dochodzi do wybrania osobnika:

Listing 2: Algorytm selekcji rankingowej

```
int Genetic::rankSelection(vector <double> fitnesses) {
  double number, totalProb = 0.0, offset = 0.0, helpRank = populationSize;
  int pick = 0;

  static_cast<double>(number = (double)rand() / (double)RAND_MAX);

  for (int i = 0; i < fitnesses.size(); i++) {
    fitnesses.at(i) = helpRank / (populationSize*(populationSize - 1));
    helpRank = helpRank - 1;
  }

  for (int i = 0; i < fitnesses.size(); i++)
    totalProb += fitnesses.at(i);

  for (int i = 0; i < fitnesses.size(); i++)
    fitnesses.at(i) = fitnesses.at(1) / totalProb;

  for (int i = 0; i < fitnesses.size(); i++) {
    offset += fitnesses.at(i);
    if (number < offset) {
      pick = i;
      break;
    }
  }
  return pick;
}
```

2.3.3 Selekcja koła ruletki

Selekcja koła ruletki również opiera się na funkcji zdatności. Różni się od *selekcji rankingowej* wyliczaniem właśnie funkcji zdatności:

$$fitness\ value = \frac{1}{k}$$
gdzie k = koszt ścieżki

Następnie zgodnie z wartością *funkcji zdatności* przyporządkowane jest właściwe prawdopodobieństwo, a na końcu dochodzi do wybrania osobnika:

Listing 3: Algorytm selekcji ruletkowej

```
int Genetic::rouletteWheelSelection(vector <double> fitnesses) {
  double number, totalFitness = 0.0, offset = 0.0;
  int pick = 0;

  static_cast<double>(number = (double)rand() / (double)RAND_MAX);

  for (int i = 0; i < fitnesses.size(); i++)
    totalFitness += fitnesses.at(i);

  for (int i = 0; i < fitnesses.size(); i++)
    fitnesses.at(i) = fitnesses.at(i) / totalFitness;

  for (int i = 0; i < fitnesses.size(); i++) {
    offset += fitnesses.at(i);
    if (number < offset) {
      pick = i;
      break;
    }
  }
  return pick;
}
```

2.3.4 Porównanie metod selekcji

Tablica S1: Porównanie błędów względnych średnich wyników dla różnych metod selekcji w algorytmie genetycznym

Instancja	metody selekcji		
	TS[1]	**RS**[2]	**RWS**[3]
ftv44.atsp	**2.74%**	4.38%	6.54%
ft53.atsp	**6.51%**	8.19%	10.43%
ftv55.atsp	**5.56%**	6.76%	11.69%
ftv64.atsp	5.68%	**4.41%**	9.34%
ftv47.atsp	**4.84%**	5.80%	8.72%

[1]Tournament Selection (selekcja turniejowa) [1].
[2]Rank Selection (selekcja rankingowa) [10].
[3]Roulette Wheel Selection (selekcja koła ruletki) [10].

Dla każdej z instancji problemu został przetestowany każdy z rodzajów selekcji, testy objęły 50 uruchomień programu - każde po 10 sekund. Zdecydowanie najlepiej spisała się *selekcja turniejowa*, dla której błędy względne końcowych wyników były najmniejsze.

2.4 Operatory krzyżowania

Zastosowanie operatorów krzyżowania to jedna z najważniejszych części algorytmu genetycznego. Właściwie dobrane metody generujące nowych osobników mogą znacznie zwiększyć efektywność i osiągi algorytmu. Na potrzeby programu zostało zaimplementowanych 7 różnych operatorów, które następnie zostały w różnych konfiguracjach przetestowane w wybranych instancjach. Wyniki 60 testów po 10 sekund dla wszystkich operatorów dla każdej instancji zostały przedstawione w Tablicy S2:

Tablica S2: Porównanie błędów względnych średnich wyników dla różnych operatorów krzyżowania w algorytmie genetycznym

Instancja	operatory						
	PMX [1]	**OX** [2]	**CX** [3]	**CX2** [4]	**TPX** [5]	**SCX** [6]	**ESCX** [7]
ftv44.atsp	6.53%	**2.67%**	5.61%	8.89%	4.97%	2.78%	5.25%
ft53.atsp	9.13%	**6.98%**	9.66%	12.59%	8.90%	11.44%	11.53%
ftv55.atsp	7.19%	**6.07%**	8.20%	15.36%	6.82%	8.62%	8.14%
ftv64.atsp	5.57%	5.38%	7.33%	11.46%	6.22%	5.32%	**4.19%**
ftv47.atsp	8.32%	5.29%	8.26 %	12.92%	7.77%	5.59%	**4.62%**

[1]Partially Mapped Crossover [2].
[2]Order Crossover [2].
[3]Cycle Crossover [2].
[4]Cycle Crossover 2 [2].
[5]Two-Point Crossover [3].
[6]Sequential Constructive Crossover [6].
[7]Enhanced Sequential Constructive Crossover [6].

Przy każdym z zaimplementowanych operatorów widnieje przypis, który odsyła do linku pod którym szczegółowo są opisane działania poszczególnych operatorów, dlatego w sekcji tej zostaną przedstawione tylko dwa - *OX* i *ESCX* - które dawały w testach najlepsze rezultaty.

2.4.1 Order Crossover Operator

Mając rodziców o przykładowych ścieżkach losujemy dwa indeksy i oraz j (zakładając, że $i < j$), pomiędzy którymi tworzy się tzw. *sekcja dopasowania*. W operatorze niemożliwe jest zaburzenie prawidłowości ścieżki poprzez zmianę pozycji zer (pierwszy i jednocześnie ostatni wierzchołek), lecz zostały zachowane w celu ukazania właściwej struktury ścieżki:

rodzic 1 →	0	4	8	7	6	1	3	5	2	0
rodzic 2 →	0	1	5	8	7	2	3	6	4	0

Materiał genetyczny z sekcji dopasowania zostaje przekopiowany do odpowiadających rodzicom potomków:

potomek 1 →	0		**6**	**1**	**3**		0
potomek 2 →	0		**7**	**2**	**3**		0

Następnie z obu rodziców tworzona jest nowa sekwencja wierzchołków, począwszy od pierwszego wierzchołka po sekcji dopasowania w każdym rodzicu, a po dojściu do końca ścieżki następuje powrót na początek i do sekwencji dokładane są kolejne wierzchołki aż do dojścia do początku sekcji dopasowania:

sekwencja powstała z 1 rodzica	5	$\rightarrow$	2	$\rightarrow$	4	$\rightarrow$	8	$\rightarrow$	7	$\rightarrow$	6	$\rightarrow$	1	$\rightarrow$	3
sekwencja powstała z 2 rodzica	6	$\rightarrow$	4	$\rightarrow$	1	$\rightarrow$	5	$\rightarrow$	8	$\rightarrow$	7	$\rightarrow$	2	$\rightarrow$	3

Kolejnym krokiem jest wstawienie sekwencji - zapobiegając kolizji, czyli wizytom odwiedzonych już wierzchołków - *na krzyż*, czyli sekwencja z 1 rodzica do potomka 2 i sekwencja z 2 rodzica do potomka 1. Oto przykład kolizji gdyby nie skorzystano z weryfikacji obecności wierzchołka w ścieżce:

potomek 1 $\rightarrow$	0	**6**	**1**	**3**	6	4	0
potomek 2 $\rightarrow$	0	**7**	**2**	**3**	5	2	0

Prawidłowym krokiem dla potomka 1 w tej sytuacji jest pominięcie pierwszej w sekwencji 6, wpisanie 4, pominięcie 1 i wpisanie dopiero 5. Zaś dla potomka drugiego właściwym będzie wpisanie 5, ominięcie 2 i następnie wpisanie 4. Analogicznie sytuacja ma się do zapełnienia początkowych miejsc w ścieżce (oczywiście jeśli takowe są dostępne, tylko jeśli $i \neq 1$). Końcowe struktury potomków prezentują się następująco:

potomek 1 $\rightarrow$	0	**8**	**7**	**2**	6	1	3	**4**	**5**	0
potomek 2 $\rightarrow$	0	**8**	**6**	**1**	7	2	3	**5**	**4**	0

2.4.2 Enhanced Sequential Crossover Operator

Mając przykładową instancję problemu komiwojażera oraz dwóch wyselekcjonowanych rodziców:

	0	1	2	3	4	5
0	∞	81	50	18	75	39
1	81	∞	76	21	37	26
2	50	76	∞	24	14	58
3	18	21	24	∞	19	58
4	75	37	14	19	∞	31
5	39	26	58	58	31	∞

rodzic 1 $\rightarrow$	0	4	5	1	3	2	0
rodzic 2 $\rightarrow$	0	1	5	3	2	4	0

Zaczynając od wierzchołka startowego rozpatrujemy dwa następne wierzchołki w obojgu rodziców, tj. wierzchołek 4 dla rodzica 1 i wierzchołek 1 dla rodzica 2. Zgodnie z następującym kryterium wybierany jest kolejny wierzchołek w potomku:

$$\min(c_{n_3 n_1} + c_{n_1 t}, c_{n_3 n_2} + c_{n_2 k})$$

gdzie:

n_3 - poprzedni wierzchołek, n_1 - pierwszy rozpatrywany wierzchołek, n_2 - drugi rozpatrywany wierzchołek

c_{ij} - koszt przejścia z wierzchołka i do j

$c_{n_1 t}$ = min $\{c_{n_1 n_j} : n_j$ - nieodwiedzony w potomku wierzchołek $\}$
$c_{n_2 k}$ = min $\{c_{n_2 n_j} : n_j$ - nieodwiedzony w potomku wierzchołek $\}$

Czyli dla $n_3 = 0$, $n_1 = 4$ i $n_2 = 1$:

$$\min(c_{04} + c_{4t}, c_{01} + c_{1k}) = \min(75 + 14, 81 + 21) = \min(89, 102)$$

Rywalizację wygrał wierzchołek 4 i on zostaje wpisany jako następujący po 0 w potomku:

potomek 1 → 0 4 0

Następnie trzeba znaleźć wierzchołki następujące po 4 w obu rodzicach - są to wierzchołek 5 w rodzicu 1 i wierzchołek 0 w rodzicu 2. Jednak wierzchołek 0 jest już odwiedzony, więc w takiej sytuacji wybierany jest pierwszy nieodwiedzony wierzchołek ze zbioru {0, 1, 2, 3, 4, 5}, jakim jest wierzchołek 1. Czyli dla $n_3 = 4$, $n_1 = 5$ i $n_2 = 1$:

$$\min(c_{45} + c_{5t}, c_{41} + c_{1k}) = \min(31 + 26, 37 + 21) = \min(57, 58)$$

Rywalizację wygrał wierzchołek 5 i on zostaje wpisany jako następujący po 4 w potomku:

potomek 1 → 0 4 5 0

Procedura odbywa się analogicznie aż do wypełnienia całej ścieżki, jeśli $n_1 = n_2$ to wierzchołek automatycznie jest dopisywany do ścieżki.

2.5 Wielkość populacji

Kolejnym z parametrów charakteryzujących algorytm genetyczny jest wielkość populacji. Im populacja jest większa, tym mniej iteracji się wykona w danym czasie, lecz jest większa szansa, na trafienie na lepszego osobnika. Kluczem jest przetestowanie różnych wielkości, tak aby wybrać dla danej instancji tę optymalną. W celu znalezienia właściwej zostało wykonanych 30 testów po 10 sekund dla danych wielkości dla każdej instancji, a błędy względne średnich wyników zostały zamieszczone w Tablicy S3:

Tablica S3: Porównanie błędów względnych średnich wyników dla różnych wielkości populacji w algorytmie genetycznym

Instancja	wielkości populacji						
	10	**50**	**100**	**150**	**200**	**250**	**300**
ftv44.atsp	6.89%	5.41%	2.38%	**1.67%**	2.90%	3.69%	3.38%
ft53.atsp	14.81%	9.42%	7.08%	**5.59%**	6.68%	7.38%	9.96%
ftv55.atsp	10.92%	7.22%	6.52%	6.71%	**6.24%**	9.07%	11.16%
ftv64.atsp	9.54%	7.45%	**4.80%**	6.08%	9.37%	15.02%	16.57%
ftv47.atsp	10.50%	7.70%	6.15%	**4.97%**	5.37%	6.22%	6.54%

Jak widać najlepiej sprawdziły się wielkości ze zbioru {100, 150, 200}, dając najmniejsze błędy względne ze wszystkich testowanych wielkości.

2.6 Mutacja

2.6.1 Eksploracja i eksploatacja

W literaturze często spotyka się stwierdzenia, że w algorytmie genetycznym za kwestię *eksplotacji* odpowiada operator krzyżowania, a za kwestię *eksploracji* odpowiada właśnie mutacja. Nie jest to do końca poprawne, ponieważ tylko odpowiednie dobranie i połączenie operatorów selekcji, krzyżowania i mutacji odpowiada za właściwe zbalansowanie tych dwóch kluczowych pojęć związanych z algorytmem genetycznym. *Eksploracja* polega na „badaniu nieznanych terenów", czyli przeszukiwaniu nowych ścieżek, znacznie różniących się od poprzedniej, zaś *eksploatacja* polega na przeszukiwaniu sąsiedztwa. Kluczem jest właściwe ich zbalansowanie, gdyż jeśli *eksploracja* będzie miała za dużą rolę to możliwe jest nie dojście w danym sąsiedztwie do optimów lokalnych, zaś w odwrotnym przypadku istnieje możliwość utknięcie w tychże optimach.

2.6.2 Typ mutacji

W przedstawionej implementacji mutacja zapewnia eksploatację sąsiedztwa w trzech typach, które zostały wykorzystane również w poprzednich implementacjach algorytmów przeszukiwania lokalnego - *Tabu Search* oraz *Simulated Annealing*:

- swap
- insert
- reverse

2.6.3 Prawdopodobieństwo mutacji

Ważnym parametrem dotyczącym mutacji jest jej prawdodobieństwo, z którym zachodzi dla każdego nowego potomka. Zostało przeprowadzonych 30 testów po 10 sekund dla prawdopodobieństw od *0.00* do *0.40* ze skokiem co *0.05* dla wybranych instancji. Wyniki zostały zamieszczone poniżej:

Tablica S4: Porównanie błędów względnych średnich wyników dla różnych wartości prawdopodobieństwa mutacji w algorytmie genetycznym

Instancja	wartości								
	0.00	**0.05**	**0.10**	**0.15**	**0.20**	**0.25**	**0.30**	**0.35**	**0.40**
ftv44.atsp	4.62%	3.14%	3.16%	3.02%	3.26%	2.65%	**2.58%**	3.05%	3.75%
ft53.atsp	7.07%	6.81%	6.70%	**6.41%**	6.89%	7.38%	6.55%	7.91%	8.48%
ftv55.atsp	**5.00%**	6.07%	**5.00%**	5.54%	6.74%	5.81%	6.58%	7.25%	6.50%
ftv64.atsp	5.15%	**4.73%**	5.59%	5.74%	5.10%	6.37%	6.83%	5.42%	7.40%
ftv47.atsp	6.31%	6.61%	5.67%	**4.93%**	5.77%	6.14%	6.15%	7.06%	6.40%

Zdecydowanie najlepiej spisały się prawdopodobieństwa z zakresu *0.00-0.15*, jednak same różnice w błędach względnych są niewielkie i mogą wynikać z wykonania zbyt małej ilości testów.

2.7 Elitaryzm

Gdyby za każdym razem zastępować starą populację tą nowo utworzoną mogłoby dojść do sytuacji utraty jakiegoś super osobnika, który mógł reprezentować nawet globalne optimum. Dlatego ważnym jest skorzystanie z tzw. *elitaryzmu*, którego działanie pozwala na zachowanie danej liczby najlepszych osobników ze starej populacji. Dla wielkości populacji równej 150 zostało przeprowadzonych po 30 testów dla wybranych wielkości elitaryzmu dla wszystkich instancji:

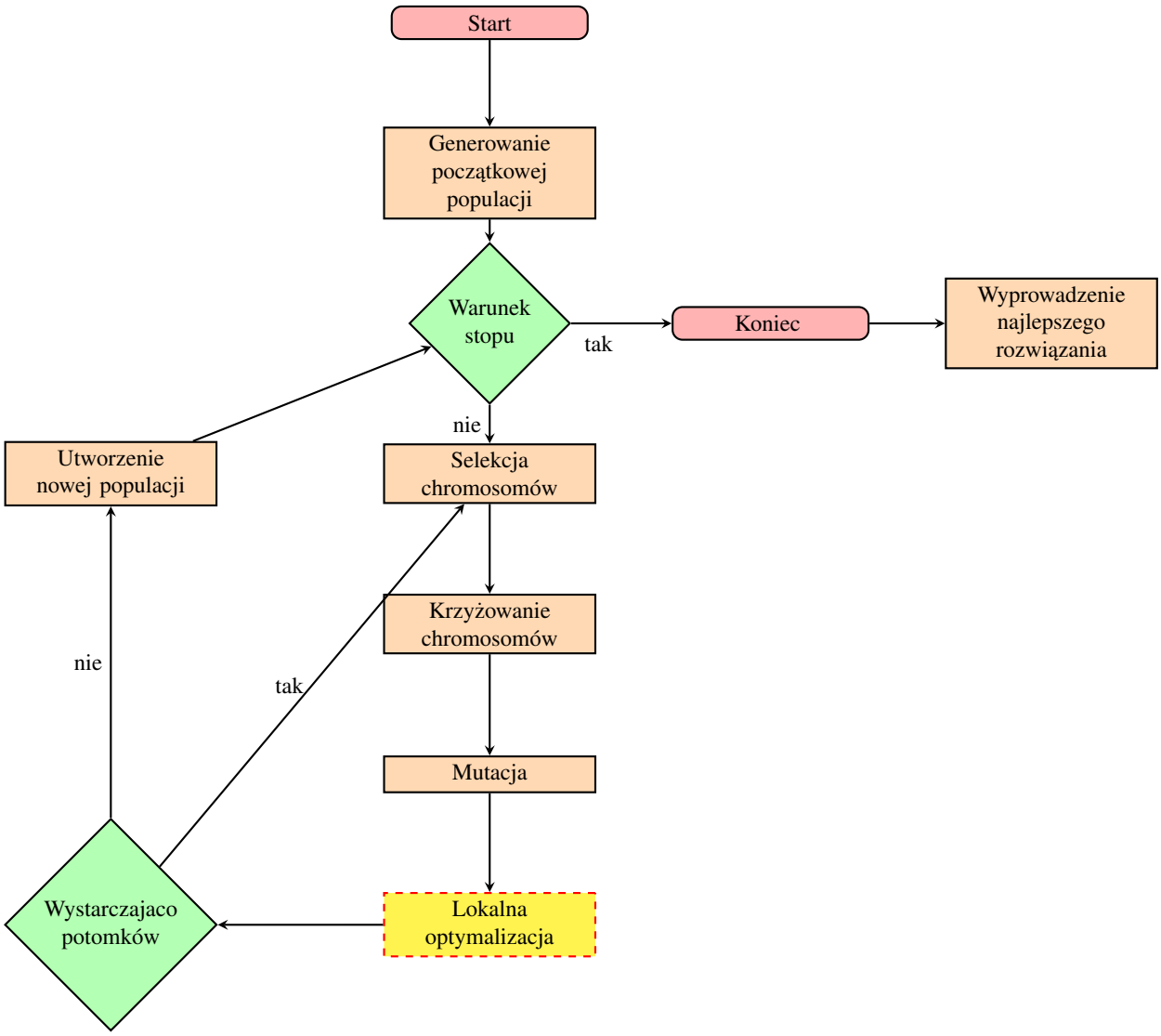
Tablica S5: Porównanie błędów względnych średnich wyników dla różnych wielkości elitaryzmu w algorytmie genetycznym

Instancja	wartości							
	0	**1**	**5**	**10**	**15**	**25**	**35**	**45**
ftv44.atsp	39.00%	5.82%	**3.03%**	3.19%	4.77%	3.96%	3.89%	3.30%
ft53.atsp	57.51%	10.68%	**5.21%**	7.36%	8.52%	8.48%	7.53%	7.08%
ftv55.atsp	91.47%	12.28%	**5.44%**	7.06%	6.74%	6.58%	6.19%	7.13%
ftv64.atsp	92.26%	11.08%	6.16%	**4.29%**	4.79%	4.90%	4.89%	4.64%
ftv47.atsp	57.49%	10.36%	5.54%	**5.19%**	6.55%	5.75%	5.79%	5.48%

Widać, że najlepiej spisuje się pozostawienie 5 lub 10 osobników ze starej populacji oraz, że całkowite wymienienie populacji (0) powoduje olbrzymie błędy względne końcowych wyników.

3 Algorytm memetyczny

Algorytm memetyczny to delikatna modyfikacja algorytmu genetycznego. Poprzez dołączenie metody *lokalnej optymalizacji*, czyli wyboru najlepszego sąsiedztwa w zakresie osobinika, można uzyskać znaczną poprawę osiągów algorytmu. Jak widać na Rysunku 2 (strona niżej) schemat algorytmu memetycznego różni się jedynie dodatkowym blokiem występującym po mutacji. Dla najlepszych znalezionych do tej pory parametrów dla danych instancji zostało przeprowadzonych po 100 testów dla każdego z algorytmów (w *MA* została wykorzystana lokalna optymalizacja poprzez typ sąsiedztwa *insert*):



Rysunek 2: Schemat *Algorytmu memetycznego*

3.1 Porównanie algorytmu genetycznego i memetycznego

W Tablicy S6 z następnej strony widać znaczne ulepszenie jakie dał algorytm memetyczny, mimo utworzenia znacznie mniejszej ilości pokoleń w danym czasie, co wynika z zapotrzebowań czasowych na lokalną optymalizację i przeszukanie całego sąsiedztwa tak aby wybrać najlepsze.

Tablica S6: Porównanie algorytmu genetycznego (GA) i algorytmu memetycznego (MA)

Instancja	Optimum	GA				MA				Pokolenia
		avg	RE	best	RE	avg	RE	best	RE	[GA / MA]
ftv44.atsp	1613	1809.5	12.18%	1730	7.25%	1655.2	2.62%	1623	0.62%	4311 / 451
ft53.atsp	6905	7912.5	14.59%	7185	4.06%	7396.6	7.12%	7070	2.39%	3809 / 255
ftv55.atsp	1608	1852.7	15.22%	1693	5.29%	1688.9	5.03%	1608	0.00%	3719 / 247
ftv64.atsp	1839	2210.3	14.75%	1998	8.65%	1945.2	5.77%	1880	2.23%	2867 / 148
ftv47.atsp	1776	2057.0	15.82%	1889	6.36%	1877.7	5.72%	1803	1.52%	4083 / 386

avg - średnia z pomiarów, best - najlepszy z pomiarów, RE - błąd względny

Wykorzystano typy sąsiedztwa dokładnie te same co w mutacji, a zarazem w wcześniej zaimplementowanych algorytmach przeszukiwania lokalnego, co pozwoliło na wykorzystanie napisanych już metod, które zwracają najlepszego sąsiada w danym typie sąsiedztwa (za pomocą metody szybkiego liczenia ruchu), a następnie wykonują właściwy ruch i aktualizują koszt ścieżki:

Listing 4: Lokalna optymalizacja

```
void Genetic::memeticImprovement(vector <unsigned>& ind) {
  int bestBalance, bestI = 0, bestJ = 0;

  if (memeticType == 0) {
    bestBalance = getBestNeighborhoodSwap(bestI, bestJ, ind);
    swapVector(bestI, bestJ, ind);
  }

  if (memeticType == 1) {
    bestBalance = getBestNeighborhoodInsert(bestI, bestJ, ind);
    insertVector(bestI, bestJ, ind);
  }

  if (memeticType == 2) {
    bestBalance = getBestNeighborhoodReverse(bestI, bestJ, ind);
    reverseVector(bestI, bestJ, ind);
  }

  ind.at(matrixSize + 1) += bestBalance;
}
```

Ze względu na podobieństwo mutacji i lokalnej optymalizacji zostały razem przetestowane w rożnych 9-u konfiguracjach (3x3), a wyniki zostały zamieszczone w poniższej tablicy:

Tablica S7: Porównanie różnych typów sąsiedztw

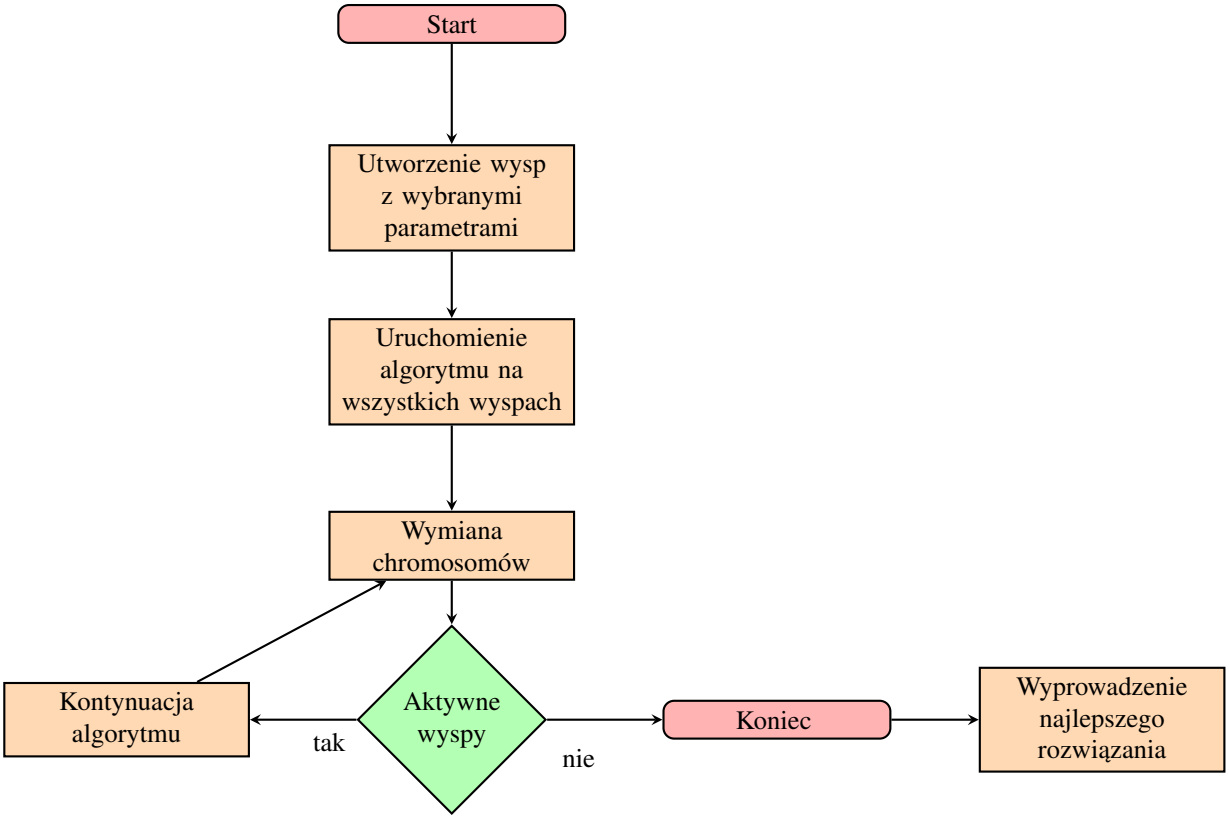
Instancja	Konfiguracja [mutacja/lokalna optymalizacja]								
	ins/ins	**ins/rev**	**ins/swap**	**rev/ins**	**rev/rev**	**rev/swap**	**swap/ins**	**swap/rev**	**swap/swap**
ftv44.atsp	2.82%	6.39%	8.12%	**2.74%**	5.20%	9.20%	2.90%	7.76%	10.08%
ft53.atsp	**5.70%**	8.92%	11.88%	5.92%	8.91%	12.75%	6.13%	10.16%	12.40%
ftv55.atsp	**4.82%**	8.90%	9.08%	5.77%	7.44%	10.11%	5.76%	10.15%	10.24%
ftv64.atsp	**5.01%**	10.46%	10.65%	5.72%	9.83%	11.98%	5.04%	11.49%	11.25%
ftv47.atsp	**4.53%**	7.84%	9.34%	4.91%	6.87%	10.26%	4.58%	9.31%	11.05%

ins - sąsiedztwo typu insert rev - sąsiedztwo typu reverse swap - sąsiedztwo typu swap

Widać, że najlepiej sprawdziło się sąsiedztwo *insert*, które polega jedynie na zastąpieniu 3-ch wybranych krawędzi 3-ma nowymi, co prowadzi do bardzo szczegółowej eksploatacji danego sąsiedztwa.

4 Algorytm wyspowy

Algorytm wyspowy to wielowątkowa implementacja algorytmu memetycznego. Ideą algorytmu wyspowego (ang. *Island Model Genetic Algorithm* - IMGA) jest utworzenie populacji na kilku wyspach, które co daną liczbę pokoleń wymieniają się w ścisle określony sposób osobnikami. Jest to dodatkowy sposób na eksplorację nowych rozwiązań.



Rysunek 3: Schemat *Algorytmu wyspowego*

4.1 Wymiana osobników

Kluczową kwestią w algorytmie wyspowym jest komunikacja między wyspami i wymiana osobników. W przedstawionej implementacji każda wyspa - co 5 pokoleń - „przyjmuje do siebie" po jednym najlepszym osobniku z pozostałych wysp. Jest to kolejny czynnik pogłębiający eksplorację przestrzeni rozwiązań.

4.2 Synchronizacja wysp

Co nie zostało wykonane, a wydaje się konieczne z perspektywy programu to zrealizowanie ochrony sekcji krytycznej, gdyż wektor zawierający najlepszych obecnie osobników na danych wyspach może być nadpisany np. na wyspie X w momencie wymiany osobników pomiędzy wyspą X a Y, co może doprowadzić do uzyskania *rozwiązań niedopuszczalnych*. Jednakże w testach nie zostały wykryte takowe rozwiązania i ze względu na konieczność zamknięcia projektu synchronizacja nie została finalnie zrealizowana.

5 Testy

5.1 Algorytm wyspowy

Do głównych testów został wybrany algorytm wyspowy, ze względu na teoretyczną przewagę nad alogrytmem memetycznym, który w testach okazał się znacznie lepszy od zwykłego algorytmu genetycznego. Testy zostały wykonane na sześciu różnych instancjach - do dotychczas testowanych została dołączona większa instancja - *ftv170.atsp*.

5.1.1 Parametry

Spośród wszystkich przeprowadzonych do tej pory testów dla każdego parametru, zostały wybrane takie, które zapewnią optymalność i jeszcze lepszą eksplorację przestrzeni rozwiązań. Utworzone zostały 4 wyspy, z których - w zakresie jednej instancji - każda różni się od pozostałych przynajmniej dwoma parametrami.

Tablica S8: Parametry dla poszczególnych wysp dla wybranych instancji

Instancja	Wyspy	Parametry							
		1	2	3	4	5	6	7	8
ftv44.atsp	1 wyspa	150	3	0.15	OX	TS	insert	5	insert
	2 wyspa	150	3	0.30	SCX	TS	insert	5	insert
	3 wyspa	100	4	0.25	OX	RS	reverse	10	insert
	4 wyspa	200	4	0.00	ESCX	TS	reverse	25	insert
ft53.atsp	1 wyspa	150	3	0.15	OX	TS	insert	5	insert
	2 wyspa	150	3	0.05	SCX	TS	insert	5	insert
	3 wyspa	100	4	0.25	OX	RS	reverse	5	insert
	4 wyspa	200	4	0.20	ESCX	TS	insert	10	insert
ftv55.atsp	1 wyspa	200	2	0.00	OX	TS	reverse	5	insert
	2 wyspa	150	2	0.10	PMX	TS	insert	5	insert
	3 wyspa	100	4	0.15	OX	RS	reverse	5	insert
	4 wyspa	200	1	0.10	ESCX	RS	insert	25	insert
ftv64.atsp	1 wyspa	100	2	0.05	ESCX	TS	insert	5	insert
	2 wyspa	150	2	0.10	PMX	TS	reverse	5	insert
	3 wyspa	100	4	0.15	OX	RS	reverse	10	insert
	4 wyspa	150	1	0.20	SCX	RS	insert	15	insert
ftv47.atsp	1 wyspa	150	2	0.15	ESCX	TS	insert	5	insert
	2 wyspa	200	2	0.10	ESCX	TS	reverse	5	insert
	3 wyspa	100	4	0.15	OX	TS	insert	10	insert
	4 wyspa	150	1	0.00	SCX	RS	insert	10	insert
ftv170.atsp	1 wyspa	150	2	0.15	ESCX	TS	insert	5	insert
	2 wyspa	200	2	0.10	OX	TS	insert	5	insert
	3 wyspa	150	3	0.10	SCX	TS	insert	10	insert
	4 wyspa	200	4	0.05	SCX	RS	insert	10	insert

Opis parametrów:

1 - wielkość populacji
2 - liczba losowych wierzchołków przy generowaniu początkowej populacji
3 - prawdopodobieństwo mutacji
4 - operator krzyżowania
5 - typ selekcji
6 - typ sąsiedztwa mutacji
7 - elitaryzm (ilość najlepszych osobników do przetrwania ze starej populacji)
8 - typ sąsiedztwa lokalnej optymalizacji

5.1.2 Wyniki

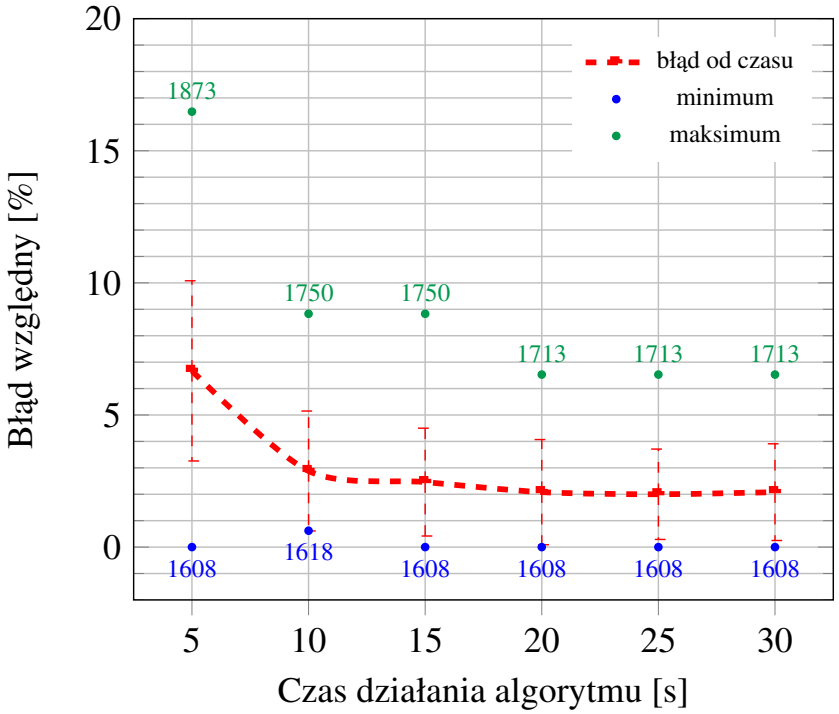
Tablica S9: Wyniki działania algorytmu wyspowego dla danych instancji

Instancja	Optimum	Czas [s]	Rezultaty					
			avg	best	worst	avgErr	bestErr	worstErr
ftv44.atsp	1613	5	1646.3	1613	1705	2.06%	0.00%	5.70%
		10	1645.7	1623	1704	2.02%	0.62%	5.64%
		15	1644.0	1613	1690	1.92%	0.00%	4.77%
		20	1639.1	1613	1668	1.62%	0.00%	3.41%
		25	1641.0	1613	1683	1.73%	0.00%	4.34%
		30	1637.4	1613	1673	1.51%	0.00%	3.72%
ft53.atsp	6905	5	7351.1	6976	7734	6.46%	1.03%	12.01%
		10	7238.0	6909	7601	4.82%	0.06%	10.08%
		15	7207.0	6976	7643	4.37%	1.03%	10.69%
		20	7196.0	6966	7549	4.21%	0.88%	9.33%
		25	7209.8	6905	7591	4.41%	0.00%	9.93%
		30	7205.3	6976	7541	4.35%	1.03%	9.21%
ftv55.atsp	1608	5	1715.2	1608	1873	6.67%	0.00%	16.48%
		10	1654.3	1618	1750	2.88%	0.62%	8.83%
		15	1647.6	1608	1750	2.46%	0.00%	8.83%
		20	1641.5	1608	1713	2.08%	0.00%	6.53%
		25	1640.1	1608	1713	2.00%	0.00%	6.53%
		30	1641.5	1608	1713	2.08%	0.00%	6.53%
ftv64.atsp	1839	5	1936.6	1864	2016	5.31%	1.36%	9.62%
		10	1873.0	1839	1921	1.85%	0.00%	4.46%
		15	1875.2	1839	1964	1.97%	0.00%	6.80%
		20	1869.4	1839	1936	1.65%	0.00%	5.27%
		25	1873.7	1839	1928	1.88%	0.00%	4.84%
		30	1868.6	1842	1915	1.61%	0.16%	4.13%
ftv47.atsp	1776	5	1832.6	1777	1881	3.18%	0.06%	5.91%
		10	1825.3	1776	1868	2.78%	0.00%	5.18%
		15	1830.2	1776	1881	3.05%	0.00%	5.91%
		20	1826.1	1776	1881	2.82%	0.00%	5.91%
		25	1825.1	1776	1864	2.76%	0.00%	4.95%
		30	1819.9	1776	1881	2.47%	0.00%	5.91%
ftv170.atsp	2755	5	3347.0	3266	3445	21.49%	18.55%	25.05%
		10	3208.5	3130	3321	16.44%	13.61%	20.54%
		15	3189.7	3134	3315	15.76%	13.76%	20.33%
		20	3172.3	3149	3298	15.16%	14.30%	19.71%
		25	3168.6	3045	3270	14.99%	10.53%	18.69%
		30	3169.7	3097	3357	15.03%	12.41%	21.85%

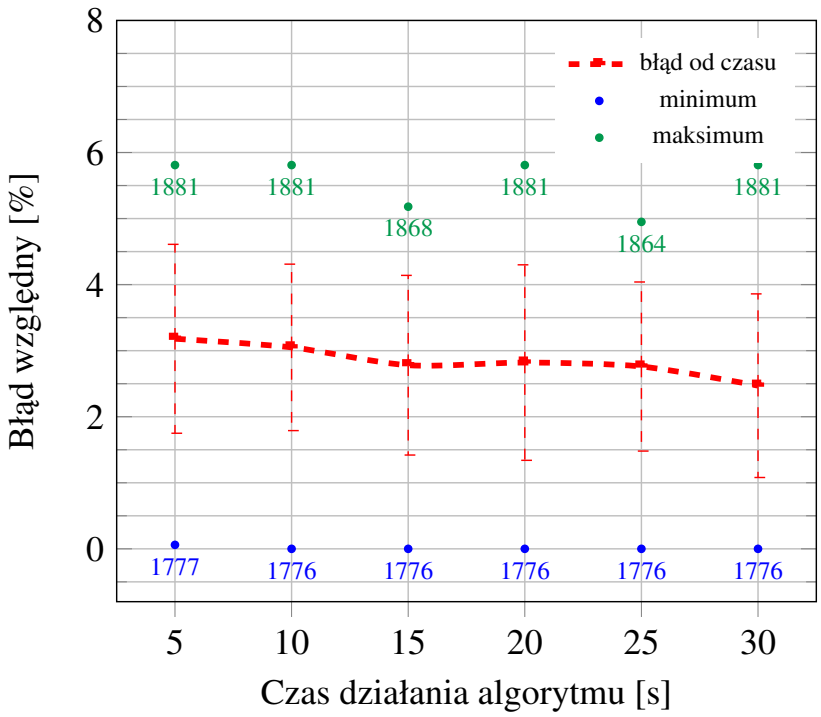
avg - średnia wyników — avgErr - błąd względny średniej wyniku
best - najlepszy wynik — avgErr - błąd względny najlepszego wyniku
worst - najgorszy wynik — avgErr - błąd względny najgorszego wyniku

5.1.3 Wykresy

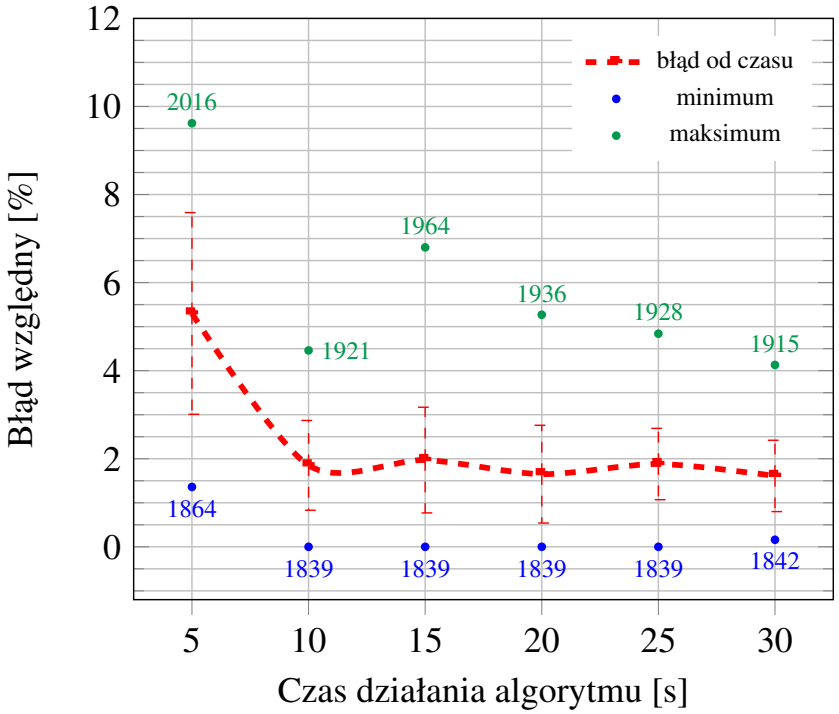
Uzyskane wyniki zostały przeniesione na wykresy zależności błędu względnego od czasu.



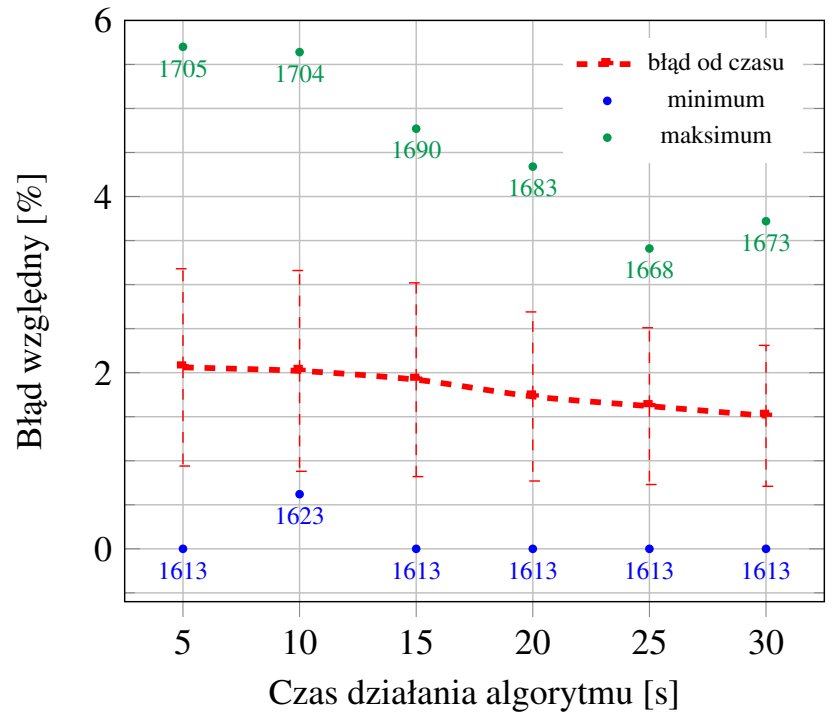
Rysunek 4: Wyniki działania *algorytmu wyspowego* dla instancji *ftv55.atsp*



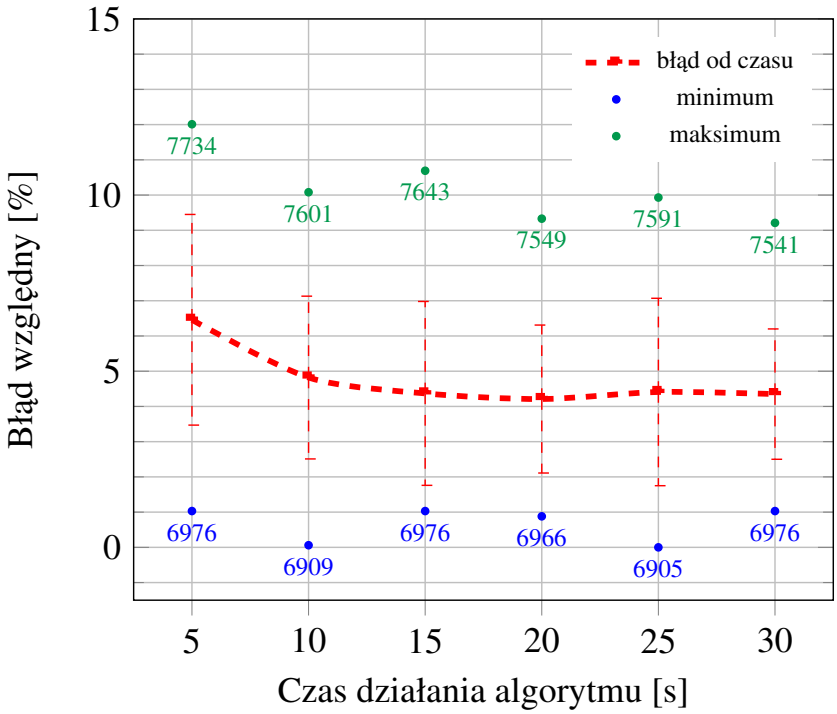
Rysunek 5: Wyniki działania *algorytmu wyspowego* dla instancji *ftv47.atsp*



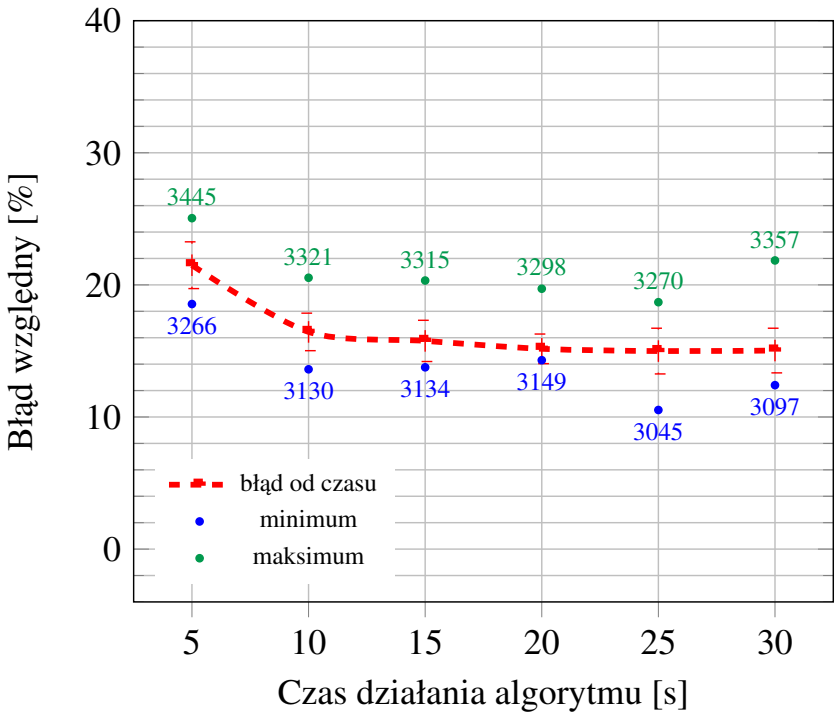
Rysunek 6: Wyniki działania *algorytmu wyspowego* dla instancji *ftv64.atsp*



Rysunek 7: Wyniki działania *algorytmu wyspowego* dla instancji *ftv44.atsp*



Rysunek 8: Wyniki działania *algorytmu wyspowego* dla instancji *ft53.atsp*



Rysunek 9: Wyniki działania *algorytmu wyspowego* dla instancji *ftv170.atsp*

5.2 Porównanie algorytmów

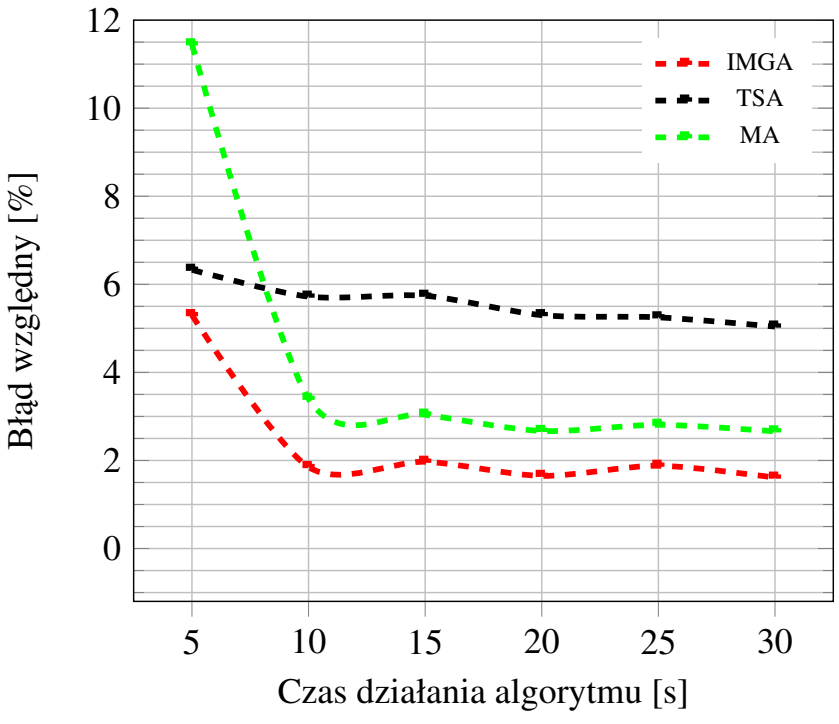
W celu uzyskania porównania między algorytmami lokalnego przeszukiwania zostały wykonane testy dla wybranych macierzy. Testy TSA i MA objęły 50 prób dla każdego okresu czasu dla znalezionych najlepszych parametrów, zaś testy IMGA zostały zaczerpnięte z Tablicy S9.

Tablica S10: Porównanie algorytmów TSA, MA i IMGA

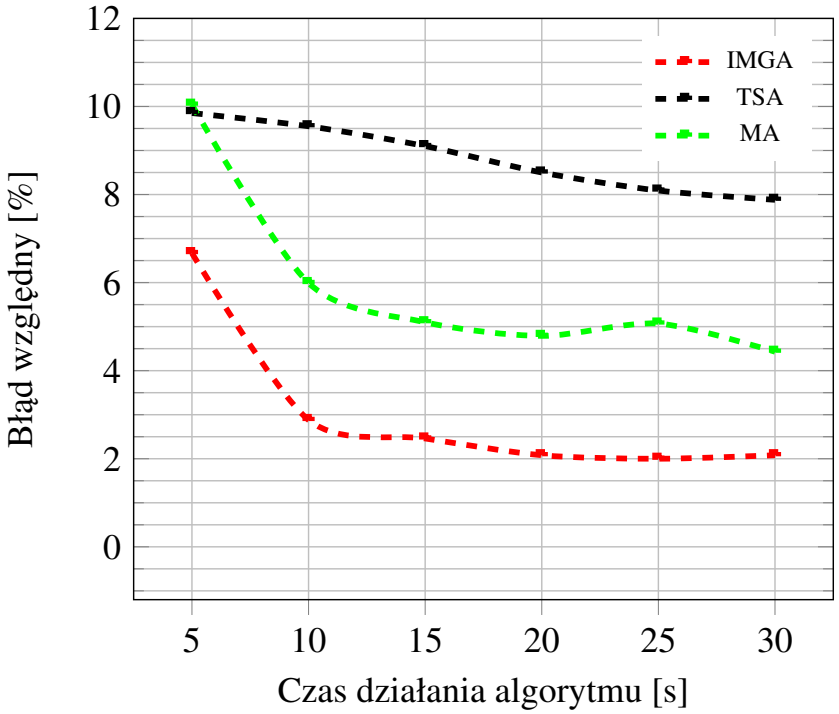
Instancja	Optimum	Algorytm	błąd względny od czasu						Najlepszy wynik
			5	10	15	20	25	30	
ftv44.atsp	1613	TSA	4.69%	2.16%	0.87%	0.87%	0.58%	0.56%	1615
		MA	4.58%	4.13%	2.56%	2.40%	2.14%	2.20%	1613
		IMGA	2.06%	2.02%	1.92%	1.73%	1.62%	1.51%	1613
ft53.atsp	6905	TSA	12.95%	11.57%	10.27%	10.28%	9.28%	8.72%	7136
		MA	7.19%	5.48%	4.96%	5.11%	5.24%	5.18%	6973
		IMGA	6.46%	4.82%	4.37%	4.21%	4.41%	4.35%	6905
ftv55.atsp	1608	TSA	9.85%	9.55%	9.10%	8.50%	8.09%	7.88%	1684
		MA	10.04%	5.99%	5.10%	4.79%	5.07%	4.43%	1608
		IMGA	6.67%	2.88%	2.46%	2.08%	2.00%	2.08%	1608
ftv64.atsp	1839	TSA	6.33%	5.72%	5.74%	5.30%	5.25%	5.04%	1875
		MA	11.45%	3.40%	3.04%	2.67%	2.81%	2.66%	1839
		IMGA	5.30%	1.85%	1.97%	1.65%	1.88%	1.61%	1839
ftv47.atsp	1776	TSA	8.23%	6.05%	5.87%	5.47%	5.05%	4.68%	1806
		MA	4.53%	4.23%	4.08%	3.83%	4.05%	4.06%	1776
		IMGA	3.18%	3.05%	2.78%	2.82%	2.76%	2.47%	1776
ftv170.atsp	2775	TSA	20.47%	20.47%	20.47%	20.47%	18.98%	18.98%	3278
		MA	22.39%	18.52%	17.28%	16.54%	16.58%	16.45%	3133
		IMGA	21.49%	16.44%	15.76%	15.16%	14.99%	15.03%	3088

TSA - algorytm tabu search (ang. *Tabu Search Algorithm*)
MA - algorytm memetyczny (ang. *Memetic Algorithm*)
IMGA - algorytm wyspowy (ang. *Island Model Genetic Algorithm*)

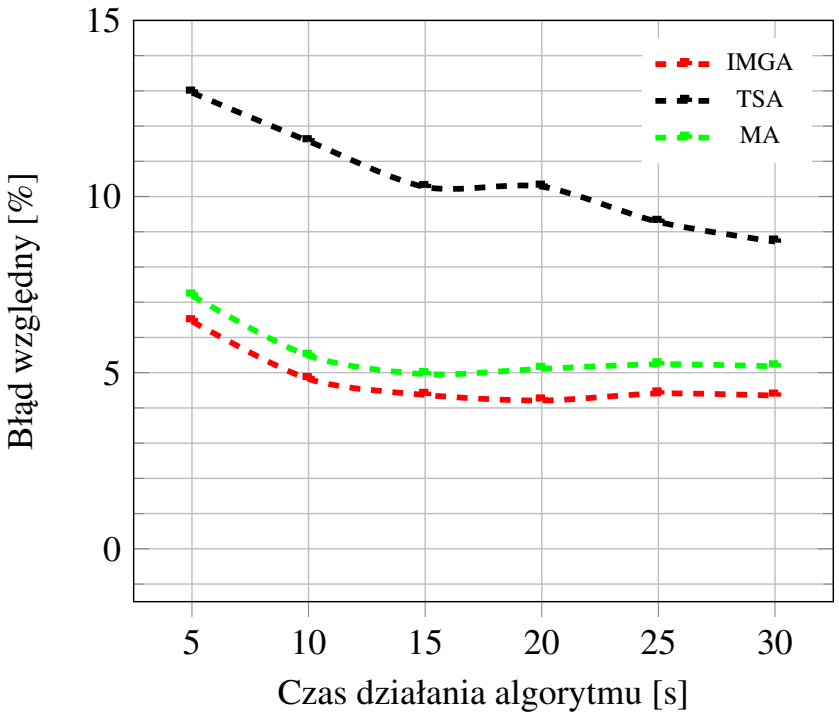
5.2.1 Wykresy



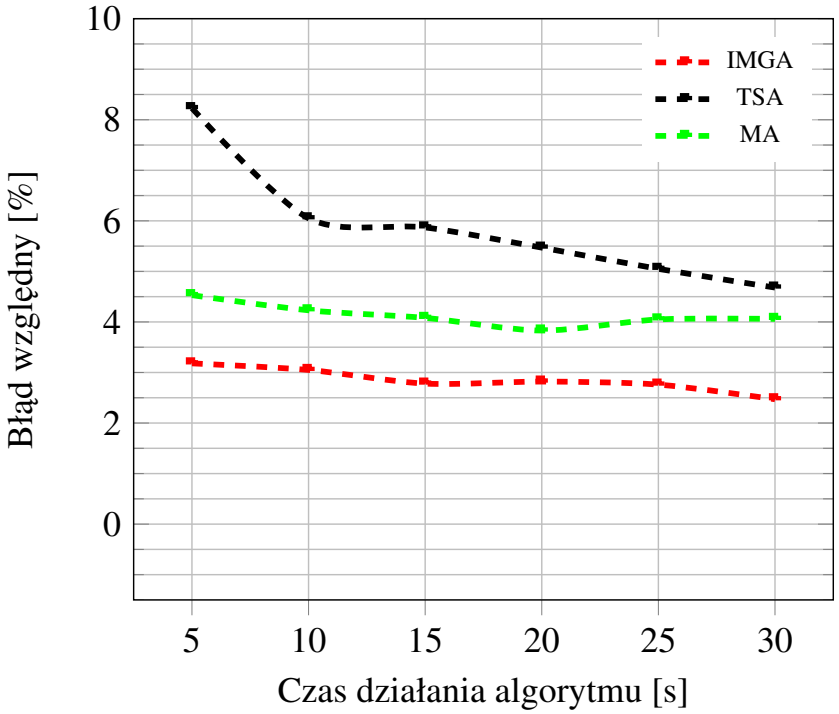
Rysunek 10: Porównanie wyników działania algorytmów dla instancji *ftv64.atsp*



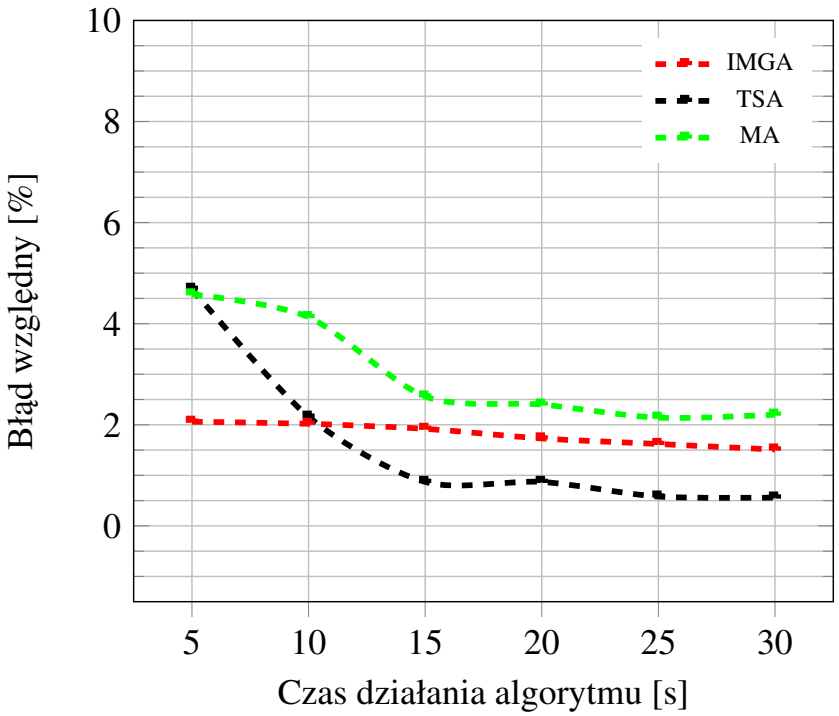
Rysunek 11: Porównanie wyników działania algorytmów dla instancji *ftv55.atsp*



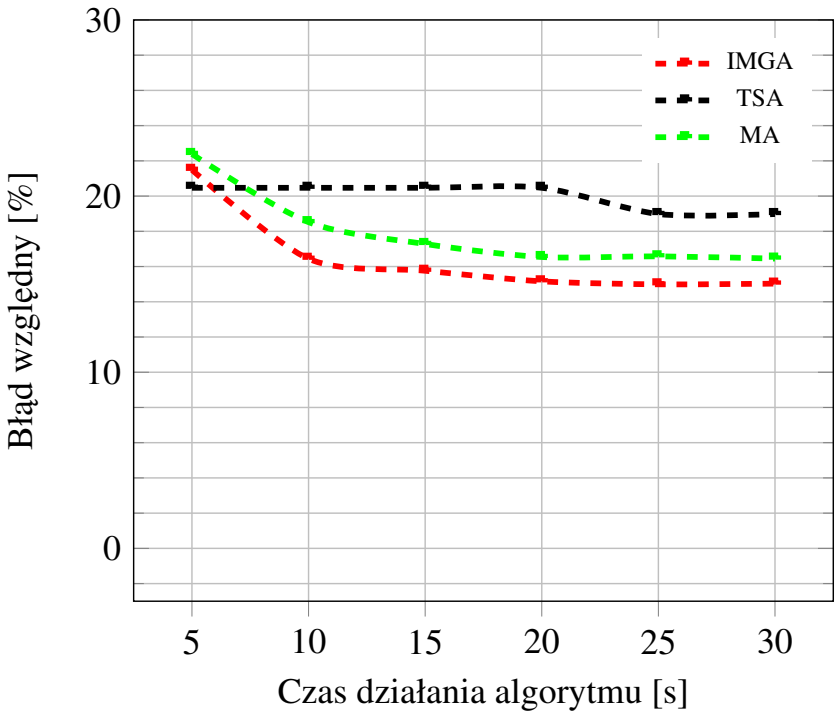
Rysunek 12: Porównanie wyników działania algorytmów dla instancji *ft53.atsp*



Rysunek 13: Porównanie wyników działania algorytmów dla instancji *ftv47.atsp*



Rysunek 14: Porównanie wyników działania algorytmów dla instancji *ftv44.atsp*



Rysunek 15: Porównanie wyników działania algorytmów dla instancji *ftv170.atsp*

6 Podsumowanie realizacji etapu i wnioski

Po dokonaniu implementacji oraz wykonaniu dużej ilości testów, mając już pewne pojęcie na temat specyfiki zaimplementowanych algorytmów można dokonać ich porównania oraz indywidualnej oceny.

Początkowa implementacja przebiegła zgodnie ze standardowym schematem algorytmu genetycznego, lecz po jej ukończeniu i krótkiej fazie testów, które dały niezadowalające osiągi, zgodnie ze schematem 2 został stworzony algorytm memetyczny wykorzystujący przeszukiwanie sąsiedztwa i wybór najlepszego sąsiada. Na podstawie tabeli S6 widać, że algorytm memetyczny - mimo wykonania 10 razy mniejszej ilości pokoleń - daje znacznie lepsze rezultaty średnie oraz jest w stanie znaleźć lepsze rozwiązania od najlepszych rozwiązań uzyskanych za pomocą algorytmu genetycznego. Na przykładzie instancji *ftv55.atsp* daje poprawę średniego błędu względnego z *15.22%* na *5.03%* oraz pozwala uzyskać globalne optimum, czyli 1608, a najlepsze rozwiązanie jakie zwrócił algorytm genetyczny to 1693.

Po zrealizowaniu algorytmu memetycznego został on użyty w wielowątkowej implementacji jaką jest algorytm wyspowy, która zapewnia komunikację między wątkami, w których w każdym z osobna działa algorytm memetyczny, tworząc tzw. *wyspę*. Wyspy komunikując się ze sobą - co polega na cyklicznej wymianie najlepszych osobników z populacji - pozwalają na teoretyczne ulepszenie algorytmu, intensyfikując eksplorację przestrzeni rozwiązań.

Analizując wykresy błędów względnych - jakie dał algorytm wyspowy - od czasu, rzuca się w oczy nieznaczny spadek błędów średnich wyników oraz tendencja do znajdowania lepszych rozwiązań wraz ze wzrostem czasu działania algorytmu. Gwałtowny spadek między 5-a a 10-a sekundami na wykresach 4 i 6 - instancje *ftv55.atsp* oraz *ftv64.atsp* - są spowodowane niewytworzeniem jeszcze całej początkowej populacji ze względu na zapotrzebowanie czasowego algorytmu *branch&bound'owego*, dlatego przy testach instancji o wiele większej - *ftv170.atsp* - zostały użyte jedynie algorytm zachłanny oraz losowo-zachłanny i patrząc na wykres 9 nie widać już tak znacznego skoku. Patrząc na wyniki z tabeli S9 widać, że zostało znalezione globalne optimum aż dla 5 instancji - nie zwróciły go jedynie testy dla instancji *ftv170.atsp*.

W celu lepszego zbadania efektywności zaimplementowanych algorytmów wykonane zostało zestawienie między algorytmem przeszukiwania z zakazami (ang. *Tabu Search Algorithm*) oraz algorytmem wyspowym i jego jednowątkową wersją - algorytmem memetycznym. W tabeli S10 porównującej osiągane błędy względne przy danym czasie, widać że *Tabu Search* radzi sobie niemal w każdym przypadku gorzej niż zestawione z nim algorytmy ewolucyjne. Jedynie dla instancji *ftv44.atsp* daje niższe średnie błędy względne, lecz nie jest w stanie - w odróżnieniu od konkurentów - znaleźć globalnego optimum jakim jest 1613. Porównując ze sobą algorytm memetyczny i wyspowy widać stałą tendencję do uzyskiwania mniejszych średnich błędów względnych oraz „lepszych najlepszych" wyników przez ten drugi.

Patrząc przez przekrój całej implementacji warto zwrócić uwagę na cenność różnorodności zaimplementowanych operatorów krzyżowania, szczególnie w kwestii algorytmu wyspowego, głównie przez wykorzystywanie różnych operatorów na poszczególnych wyspach. Bardzo ważne okazało się testowanie wszelkich parametrów dla każdej z wybranych instancji, co pozwoliło na dopasowanie najlepszych do algorytmu memetycznego oraz kilku najlepszych do algorytmu wyspowego. Warto zaznaczyć kluczowy wpływ wykorzystanego elitaryzmu w tabeli S5, którego zastosowanie zapobiegło utracie potencjalnego super osobnika ze starej populacji. Nie sposób również nie wspomnieć o nieprzewidywanej, olbrzymiej poprawie osiągów, jaką dało proste dołączenie elementu lokalnej optymalizacji do algorytmu genetycznego i tym samym utworzenie algorytmu memetycznego.

Wszystkie te elementy zestawione ze sobą dały znakomite - nieprzewidziane przed rozpoczęciem implementacji - wyniki, czyli znalezienie 5 globalnych optimów w 6 testowanych instancjach, porównując do 1 globalnego optimum dla tych samych instancji uzyskanego na etapie 2 za pomocą algorytmu *Tabu Search*.

Bibliografia

[1] Tournament Selection (GA). https://www.geeksforgeeks.org/tournament-selection-ga/.

[2] Abid Hussain, Yousaf Shad Muhammad, M. Nauman Sajid, Ijaz Hussain, Alaa Mohamd Shoukry, Showkat Gani. Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. https://new.hindawi.com/journals/cin/2017/7430125/, 2017.

[3] Avik Dutta. Crossover in Genetic Algorithm. https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/.

[4] Robert B.Heckendorn Darrell Whitley, Soraya Rana. The Island Model Genetic Algorithm. http://neo.lcc.uma.es/Articles/WRH98.pdf, November 1998.

[5] Adrien Goeffon Frederic Lardeux. A Dynamic Island-Based Genetic Algorithms Framework. http://info.univ-angers.fr/~goeffon/Publi/SEAL10.pdf.

[6] Hachemi Bennaceur, Entesar Alanzi. Genetic Algorithm For The Travelling Salesman Problem using Enhanced Sequential Constructive Crossover Operator. https://www.researchgate.net/publication/329643742_Genetic_Algorithm_For_The_Travelling_Salesman_Problem_using_Enhanced_Sequential_Constructive_Crossover_Operator, January 2017.

[7] Robert Kruse. *Data structures and program design*. Prentice-Hall, 1984.

[8] Radosław Lis. Listing kodu. https://github.com/radosz99/PEA. [Online; accessed 27-Januray-2020].

[9] John Geraghty Noraini Mohd Razali. Genetic Algorithm Performance with Different Selection Strategies in Solving TSP. http://www.iaeng.org/publication/WCE2011/WCE2011_pp1134-1139.pdf, July 2011.

[10] Setu Kumar Basak. How to perform Roulette wheel and Rank based selection in a genetic algorithm? https://medium.com/@setu677/how-to-perform-roulette-wheel-and-rank-based-selection-in-a-genetic-algorithm-d0829a37a189, July 2018.