Politechnika Wrocławska

Projektowanie efektywnych algorytmów

SPRAWOZDANIE – Zadanie projektowe nr 3

MARIA KRANZ

Prowadzący: dr inż. Jarosław Mierzwa

Implementacja i analiza efektywności algorytmu genetycznego (ewolucyjnego) dla problemu komiwojażera

sprawozdanie

Data oddania sprawozdania: 19.01.2024

Spis treści

I Wstęp

Wstęp teoretyczny

Asymetryczny problem komiwojażera – problem NP-trudny, polegający na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Zakładamy, że odległość z miasta A do B może być różna od odległości z miasta B do A.

Algorytm genetyczny to rodzaj heurystyki, zaprojektowany w sposób przypominający zjawisko ewolucji biologicznej. Poniżej są przedstawione jego elementy.

Funkcja przystosowania – funkcja określająca, który osobnik jest lepszy – w tej implementacji jest to osobnik z mniejszym kosztem ścieżki.

Warunek stopu – warunek który, gdy będzie spełniony, zakończy działanie algorytmu. W tej implementacji jest to ustalany przez użytkownika maksymalny czas działania algorytmu. Wybrane czasy są takie same jak w projekcie poprzednim i są to: 2min, 4min, 6min – dla plików kolejno: ftv47.xml, ftv170.xml i rbg403.xml.

Współczynnik krzyżowania – określa prawdopodobieństwo krzyżowania dwóch rodziców i utworzenia w ten sposób dwójki dzieci.

Współczynnik mutacji – określa prawdopodobieństwo mutacji genomu (ścieżki) dziecka.

Krzyżowanie OX (Order Crossover) – polega na zachowaniu części genomu rodzica i przekazanie go dziecku, reszta genomu jest uzupełniania zgodnie z kolejnością występowania genów drugiego rodzica.

Przykład graficzny:

Rodzic 1	0	1	**3**	**4**	**5**	**6**	7	8	9
Rodzic 2	2	3	**1**	**5**	**8**	**7**	0	6	4

Losujemy indeksy, na których przedziale przekopiujemy genom dzieciom: [3 – 6]

Dziecko 1			3	4	5	6			
Dziecko 2			1	5	8	7			

Na końcu uzupełniamy resztę genomu, zgodnie z kolejnością wartości w drugim rodzicu (jeżeli dana wartość już wystąpiła w genomie, to ją pomijamy).

Dziecko 1	8	7	3	4	5	6	0	2	1
Dziecko 2	4	6	1	5	8	7	9	0	3

Mutacja przez inwersję (ang. Inversion Mutation) – polega na wylosowaniu dwóch indeksów ścieżki i odwróceniu jej elementów na przedziale pomiędzy tymi indeksami.

Mutacja przez zamianę (ang. Swap Mutation) – polega na wylosowaniu dwóch indeksów i zamianie wartości miejscami na tych indeksach.

Działanie takiego algorytmu jest następujące:

1. **Wygenerowanie populacji losowej o zadanym rozmiarze** – w tym projekcie zbadano wyniki dla trzech różnych rozmiarów populacji (500, 2000, 5000). Populacja zawiera w sobie osobniki, czyli ścieżki (wraz z przystosowaniem).

Pętla programu:

Tworzymy nową pustą populacje.

2. **Elitaryzm** – zachowanie w nowej populacji najlepszych osobników ze starej populacji. W implementacji zagwarantowano przeżywalność dziesięciu najlepszych osobników.
3. **Selekcja** – jest to wybór dwóch osobników ze starej populacji, które mają szanse przekazać swoje geny potomstwu. W projekcie została zaimplementowana selekcja turniejowa, która wybiera losowych dwóch osobników, po czym wygrywa, czyli jest zwracany, silniejszy z nich (mający lepszą funkcję przystosowania). Taka selekcja turniejowa wywoływana jest w pętli dwukrotnie, żeby uzyskać dwóch rodziców.
4. **Krzyżowanie** – to uzyskanie nowego dziecka (czyli ścieżki) z wcześniej wyselekcjonowanych rodziców. Rodzice przekazują swoje geny (fragmenty ścieżek) swoim dwóm dzieciom. W tej implementacji zastosowane zostało krzyżowanie Order Crossover.
5. **Mutacja** – to losowa zmiana w genomie dzieci, w tym projekcie zaimplementowana została Swap mutation oraz Inverse mutation.

Po wykonaniu tych kroków otrzymaliśmy dwóch osobników, które umieszczamy w nowej populacji. Powtarzamy pętlę do momentu aż nie zapełnimy osobnikami żądanego rozmiaru populacji. Gdy uda się to zrobić to sprawdzamy warunek stopu (w implementacji - czas), jeżeli jest spełniony, kończymy działanie programu i zwracamy najlepszego osobnika – w przeciwnym razie tworzymy nową populację i powtarzamy pętlę.

II Opis najważniejszych klas

GeneticAlgorithm - zawiera implementacje algorytmu genetycznego

Najważniejsze metody:

```
Individual run(int stopTime, int populationSize, double mutationRate,
double crossoverRate, Mutation mutationType);
```

Rozpoczyna wykonanie algorytmu genetycznego. Metoda ta przyjmuje następujące parametry:

int stopTime – czas w sekundach (kryterium stopu),

int populationSize – rozmiar populacji,

double mutationRate – współczynnik mutacji,

double crossoverRate – współczynnik krzyżowania,

Mutation mutationType – typ mutacji: swapMutation/inversionMutation.

Poniżej listing kodu tej funkcji:

```
Individual GeneticAlgorithm::run(int stopTime, int populationSize, double
mutationRate, double crossoverRate, Mutation mutationType) {
    bestSolutionFoundInTime.clear();
    srand ( time(nullptr) );
    int bestSolutionFound = INT_MAX;

    const std::clock_t start_time = std::clock();
```

```
    //wygeneruj losowa populacje
    std::vector<Individual> population (populationSize);
    population = generateRandomPopulation(populationSize);

    while ((std::clock() - start_time) / CLOCKS_PER_SEC < stopTime) {
        std::vector<Individual> newPopulation(populationSize);
        int iteratorInPopulation = 0;

        // sortowanie wedlug przystosowania (rosnaco)
        std::sort(population.begin(), population.end(), compareIndividuals);

//jesli najnowsze rozwiązanie jest lepsze niz stare, to je zapisujemy
        if (population[0].cost < bestSolutionFound) {
            bestSolutionFound = population[0].cost;
            double currentTime = (std::clock() - (double) start_time) /
CLOCKS_PER_SEC;
            bestSolutionFoundInTime.emplace_back(currentTime, bestSolutionFound);
        }

//elitaryzm - zachowujemy 10 najlepszych wynikow z poprzedniej populacji
        for(; iteratorInPopulation < 10; iteratorInPopulation++){
            newPopulation[iteratorInPopulation] = population[iteratorInPopulation];
        }
        iteratorInPopulation--;

        while ( iteratorInPopulation != populationSize - 1) {

            //selekcja
            Individual parent1 = tournamentSelection(population);
            Individual parent2 = tournamentSelection(population);
            Individual child1 = parent1;
            Individual child2 = parent2;

            // krzyżowanie
            if (rand() < crossoverRate * RAND_MAX) {
                //krzyzowanie
                int iD1 = rand() % parent1.path.size();
                int iD2;
                do{
                    iD2= rand() % parent1.path.size();
                }while (iD2 == iD1);

                if(iD1 < iD2) OXCrossover(parent1, parent2, child1, child2, iD1,
iD2);
                else OXCrossover(parent1, parent2, child1, child2, iD2, iD1);

                //mutacja
                if (rand() < mutationRate * RAND_MAX){
                    switch (mutationType) {
                        case swapMut:
                            swapMutation(child1);
                            break;
                        case inverseMut:
                            inversionMutation(child1);
                            break;
                    }
                }
```

```
                if (rand() < mutationRate * RAND_MAX){
                    switch (mutationType) {
                        case swapMut:
                            swapMutation(child2);
                            break;
                        case inverseMut:
                            inversionMutation(child2);
                            break;
                    }
                }
            }
            newPopulation[++iteratorInPopulation] = child1;
            newPopulation[++iteratorInPopulation] = child2;
        }

        population = newPopulation;
    }

    std::sort(population.begin(), population.end(), compareIndividuals);

//jesli najnowsze rozwiązanie jest inne niz stare, to zapisujemy czas i jego
wartosc do listy
    if (population[0].cost < bestSolutionFound) {
        bestSolutionFound = population[0].cost;
        double currentTime = (std::clock() - (double) start_time) / CLOCKS_PER_SEC;
        bestSolutionFoundInTime.emplace_back(currentTime, bestSolutionFound);
    }
    return population[0];
}
```

```
std::vector<Individual> generateRandomPopulation(int populationSize);
```

Tworzy populacje początkową. Parametrem jest jej wielkość.

```
Individual tournamentSelection(std::vector<Individual>& population);
```

Zwraca osobnika, który wygrał selekcję turniejową. Parametrem jest populacja z osobnikami, spośród których będziemy losowo wybierać dwóch osobników do turnieju.

```
void OXCrossover(const Individual &parent1, const Individual &parent2,
Individual &child1, Individual &child2, startPos, endPos);
```

Wykonuje krzyżowanie Order Crossover. Jako parametry przyjmuje rodziców oraz dzieci, które zostaną zmodyfikowane, a także indeks początkowy i końcowy segmentu rodzica do skopiowania do potomka.

Poniżej listing kodu tej funkcji.

```
void GeneticAlgorithm::OXCrossover(const Individual &parent1, const Individual
&parent2, Individual &child1, Individual &child2, int startPos, int endPos) {
    child1.path.resize(parent1.path.size());
    child2.path.resize(parent2.path.size());

    //ustawianie wszystkich elementow na '-1', poniewaz w vectorze bedzie
znajdowalo sie 0 (wierzcholki sa numerowane od 0)
    for(int i = 0; i < child1.path.size(); i++){
        child1.path[i] = -1;
        child2.path[i] = -1;
    }
```

```
    // kopiowanie fragmentow rodzicow do potomkow
    for(int i = startPos; i <= endPos; i++){
        child1.path[i] = parent1.path[i];
        child2.path[i] = parent2.path[i];
    }

    // kopiowanie reszty genow do potomkow

    int iDP1 = endPos + 1;
    int iDP2 = endPos + 1;
    if(endPos+1 > child1.path.size() - 1){
        iDP1 = 0;
        iDP2 = 0;
    }
    int newPos = iDP1;
    for(int i = newPos ; i != startPos; ){
        //znajdz nastepny element do skopiowania z drugiego rodzica
        while(std::find(child1.path.begin(), child1.path.end(), parent2.path[iDP2])
!= child1.path.end()){
            iDP2++;
            if(iDP2 == parent2.path.size()) iDP2 = 0;
        }
        child1.path[i] = parent2.path[iDP2];

        while(std::find(child2.path.begin(), child2.path.end(), parent1.path[iDP1])
!= child2.path.end()){
            iDP1++;
            if(iDP1 == parent1.path.size()) iDP1 = 0;
        }
        child2.path[i] = parent1.path[iDP1];


        // inkrementacja na nastepny indeks dziecka
        if(i == parent1.path.size() - 1){
            i = 0;
        }else{
            i++;
        }
    }
    child1.cost = graph->calculateTour(child1.path);
    child2.cost = graph->calculateTour(child2.path);
}
```

```
void inversionMutation(Individual& individual);
void swapMutation(Individual& individual);
```

Funkcje wykonujące mutacje inversion lub swap. Parametrem jest osobnik podlegający mutacji.

III Pomiary błędu względnego

Wartość błędu względnego wyznaczono ze wzoru $\frac{|f_{zn}-f_{opt}|}{f_{opt}} \cdot 100\%$,

gdzie f_{zn} – wartość znaleziona, f_{opt} – wartość optymalna

Plik ftv47.xml

Algorytm został wywołany 10-krotnie dla obu typów mutacji. Dla tego pliku pomiary trwały 2 minuty (120s).

f_{opt} = 1776

Inversion Mutation

Inversion Mutation - ftv47.xml								
population size = 500			**population size = 2000**			**population size =5000**		
time[s]	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**
0,482	2048	15,32	0,768	1897	6,81	74,442	1915	7,83
65,297	1954	10,02	0,8	1936	9,01	3,335	1937	9,07
112,331	1923	8,28	1,019	1848	4,05	5,054	1981	11,54
44,697	1986	11,82	0,752	1857	4,56	4,177	1831	3,1
0,173	2076	16,89	0,753	1940	9,23	3,177	1790	0,79
97,37	1996	12,39	0,625	1859	4,67	3,945	1874	5,52
0,142	1869	5,24	0,86	2016	13,51	7,331	1821	2,53
54	2014	13,4	2,039	1800	1,35	4,538	1813	2,08
45,742	1949	9,74	89,188	1883	6,02	3,452	1926	8,45
32,497	1907	7,38	1,554	1929	8,61	5,21	1934	8,9
średnia		11,048	średnia		6,782	średnia		5,981

Tabela 1. Wyniki dziesięciu wywołań algorytmu genetycznego z Inversion Mutation dla instancji ftv47.xml

Swap Mutation

Swap Mutation - ftv47.xml								
population size = 500			**population size = 2000**			**population size =5000**		
time[s]	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**
0,256	2013	13,34	1,375	1865	5,01	1,751	1938	9,12
0,113	1962	10,47	82,181	1858	4,62	3,525	1823	2,65
0,235	2047	15,26	0,782	1841	3,66	2,565	1868	5,18
0,217	2058	15,88	22,623	1932	8,78	2,413	1951	9,85
0,266	1915	7,83	116,418	1849	4,11	2,739	1865	5,01
0,294	1933	8,84	0,703	1870	5,29	2,174	1888	6,31
0,797	1811	1,97	1,048	1988	11,94	7,002	1861	4,79
0,36	2016	13,51	0,734	1825	2,76	5,01	1889	6,36
0,125	2014	13,4	0,954	1816	2,25	4,198	1922	8,22
0,139	1908	7,43	0,812	1887	6,25	3,084	1787	0,62
średnia		10,793	średnia		5,467	średnia		5,811

Tabela 2. Wyniki dziesięciu wywołań algorytmu genetycznego ze Swap Mutation dla instancji ftv47.xml

Porównanie wyników

Przeprowadzenie doświadczenie wskazuje, że dla tej instancji lepiej radziły sobie populacje o większym rozmiarze. Populacje o rozmiarze 2000 i 5000 znajdywały rozwiązania o podobnym średnim procencie błędu względnego - koło 5.5 – 6 %.

W tym przypadku trochę lepiej zadziałała mutacja typu Swap. Średni błąd względny generowany przez działanie algorytmu z mutacją Swap był o ok. 1 punkt procentowy mniejszy niż generowany mutacją Inversion.

Średnio najlepszymi parametrami były: typ mutacji Swap oraz rozmiar populacji równy 2000.

Najlepsze rozwiązanie natomiast, znaleziono dla Swap Mutation i rozmiaru populacji równej 5000, w ostatniej próbie. Wynik jaki został uzyskany to 1787 z błędem względnym 0,62%.

Plik ftv170.xml

Algorytm został wywołany 10-krotnie dla obu typów mutacji. Dla tego pliku pomiary trwały 4 minuty (240s).

f_{opt} = 2755

Inversion Mutation

Inversion Mutation - ftv170.xml								
population size = 500			**population size = 2000**			**population size =5000**		
time[s]	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**
201,059	4621	67,73	36,442	3845	39,56	238,068	3740	35,75
24,409	5018	82,14	136,106	4209	52,78	155,612	4086	48,31
10,294	5080	84,39	67,376	3597	30,56	178,85	3417	24,03
65,672	4580	66,24	46,423	3765	36,66	227,705	4436	61,02
108,897	4794	74,01	42,606	4098	48,75	232,011	4634	68,2
19,62	4428	60,73	32,926	3785	37,39	239,79	3726	35,25
176,78	3973	44,21	39,33	3712	34,74	230,328	4714	71,11
152,802	4512	63,77	39,846	3881	40,87	239,852	4640	68,42
37,641	4641	68,46	32,023	3745	35,93	231,473	3489	26,64
193,729	4572	65,95	127,574	3872	40,54	239,708	5167	87,55
średnia		67,763	średnia		39,778	średnia		52,628

Tabela 3. Wyniki dziesięciu wywołań algorytmu genetycznego z Inversion Mutation dla instancji ftv170.xml

Swap Mutation

Swap Mutation - ftv170.xml								
population size = 500			**population size = 2000**			**population size =5000**		
time[s]	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**
171,289	4413	60,18	236,472	3906	41,78	177,53	3430	24,5
203,775	4296	55,93	36,798	3818	38,58	210,532	3609	31
127,843	4565	65,7	58,341	3911	41,96	198,692	3713	34,77
124,906	4347	57,79	63,339	4019	45,88	179,79	3651	32,52
42,863	4152	50,71	162,828	3977	44,36	226,156	3831	39,06
237,296	4325	56,99	44,823	4044	46,79	234,009	5026	82,43
193,876	4515	63,88	69,761	3936	42,87	220,478	3983	44,57
239,338	4332	57,24	98,007	3891	41,23	237,047	4570	65,88
92,105	4865	76,59	97,904	3902	41,63	174,785	3698	34,23
43,641	4608	67,26	47,388	3933	42,76	211,854	3698	34,23
średnia		61,227	średnia		42,784	średnia		42,319

Tabela 4. Wyniki dziesięciu wywołań algorytmu genetycznego ze Swap Mutation dla instancji ftv170.xml

Porównanie wyników

Dla tej instancji błąd względny wyszedł największy – najmniej 24,5%. Aby poprawić wyniki należałoby wykonać więcej testów z różnymi parametrami.

Dla testowanych parametrów, najlepszymi z nich dla rozwiązania problemu komiwojażera były średnio: wielkość populacji równa 2000 oraz metoda mutacji typu Inversion.

Najlepszy pojedynczy wynik uzyskano natomiast, dla Inversion Mutation i wielkością populacji równą 5000. Wynik był równy 3417 i miał błąd względny równy 24,03 %.

Mutacją typu Swap lepiej poradziły sobie większe instancje, czyli 2000 oraz 5000.

Dla obu typów mutacji najgorsze wyniki dawała populacja najmniejsza, czyli równa 500.

Plik rbg403.xml

Algorytm został wywołany 10-krotnie dla każdego typu sąsiedztwa. Dla tego pliku pomiary trwały 6 minut (360s).

f_{opt} = 2465

Inversion Mutation

Inversion Mutation - rbg403.xml								
population size = 500			**population size = 2000**			**population size =5000**		
time[s]	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**
355,398	2534	2,8	357,568	2864	16,19	359,822	3626	47,1
291,831	2549	3,41	357,124	2863	16,15	357,47	3475	40,97
290,661	2513	1,95	358,202	2848	15,54	358,469	3537	43,49
338,168	2515	2,03	359,591	2829	14,77	360,117	3467	40,65
348,03	2546	3,29	358,792	2964	20,24	359,27	3538	43,53
291,695	2529	2,6	358,67	2898	17,57	359,714	3634	47,42
336,221	2516	2,07	355,251	2891	17,28	357,848	3602	46,13
319,032	2528	2,56	359,383	2870	16,43	359,481	3516	42,64
285,656	2547	3,33	357,562	2849	15,58	356,912	3590	45,64
267,701	2501	1,46	355,425	2862	16,11	358,248	3626	47,1
średnia		2,55	średnia		16,586	średnia		44,467

Tabela 5. Wyniki dziesięciu wywołań algorytmu genetycznego z Inversion Mutation dla instancji rbg403.xml

Swap Mutation

Swap Mutation – rbg403.xml								
population size = 500			**population size = 2000**			**population size =5000**		
time[s]	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**	**time[s]**	**value**	**błąd wzgl [%]**
336,478	2562	3,94	357,906	2823	14,52	356,898	3494	41,74
340,724	2541	3,08	358,892	2817	14,28	359,958	3619	46,82
235,965	2528	2,56	354,182	2885	17,04	358,671	3578	45,15
335,732	2517	2,11	359,739	2846	15,46	358,97	3530	43,2
333,924	2500	1,42	356,255	2860	16,02	356,685	3493	41,7
313,676	2527	2,52	358,807	2838	15,13	359,614	3595	45,84
348,354	2536	2,88	359,348	2857	15,9	360,06	3576	45,07
331,458	2497	1,3	358,768	2852	15,7	359,832	3485	41,38
351,06	2532	2,72	359,555	2827	14,69	358,136	3526	43,04
350,293	2527	2,52	352,886	2865	16,23	357,706	3567	44,71
średnia		2,505	średnia		15,497	średnia		43,865

Tabela 6. Wyniki dziesięciu wywołań algorytmu genetycznego ze Swap Mutation dla instancji rbg403.xml

Porównanie wyników

Dla obu typów mutacji wraz ze wzrostem rozmiaru populacji, wzrastał średni błąd względny. Dlatego więc, najlepsze rozwiązania otrzymano dla rozmiaru populacji równego 500.

Średni najmniejszy błąd względny miała mutacja typu Swap i rozmiar populacji równy 500.

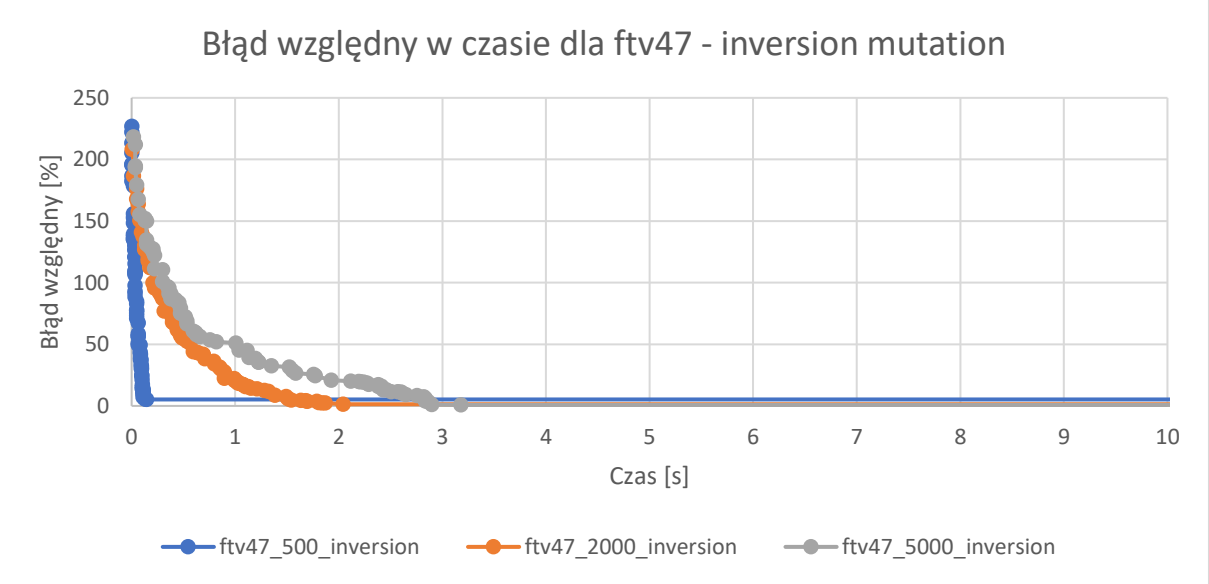
Najlepsze pojedynczy pomiar odnaleziono w siódmym wywołaniu algorytmu dla mutacji typu Swap i rozmiaru populacji równego 500.

IV Błąd w funkcji czasu

Plik ftv47.xml

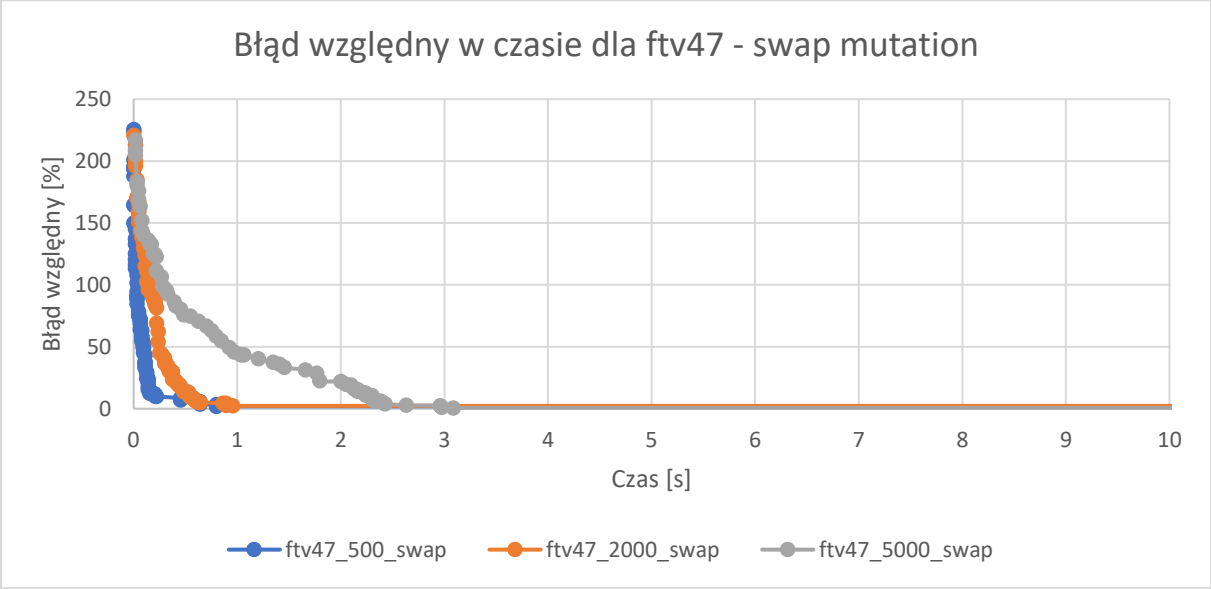
Całkowity czas trwania algorytmu = 120s. Aby zwiększyć czytelność wykresu, czas ograniczono do 10s, wyniki w dalszej części wykonywania algorytmu nie uległy zmianie.

Inversion Mutation



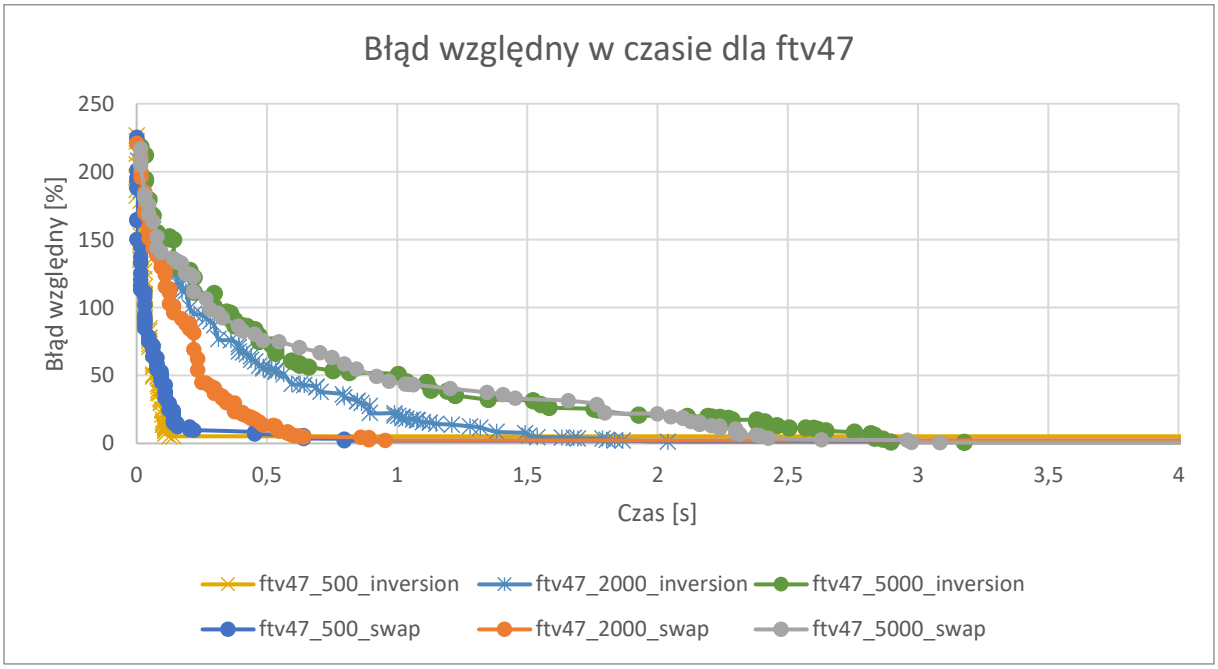
Wykres 1. Błąd względny w funkcji czasu algorytmu genetycznego z Inversion Mutation dla instancji ftv47

Swap Mutation



Wykres 2. Błąd względny w funkcji czasu algorytmu genetycznego ze Swap Mutation dla instancji ftv47.

Porównanie wyników



Wykres 3. Wyniki skumulowane pomiarów błędu w czasie algorytmu genetycznego dla ftv47.

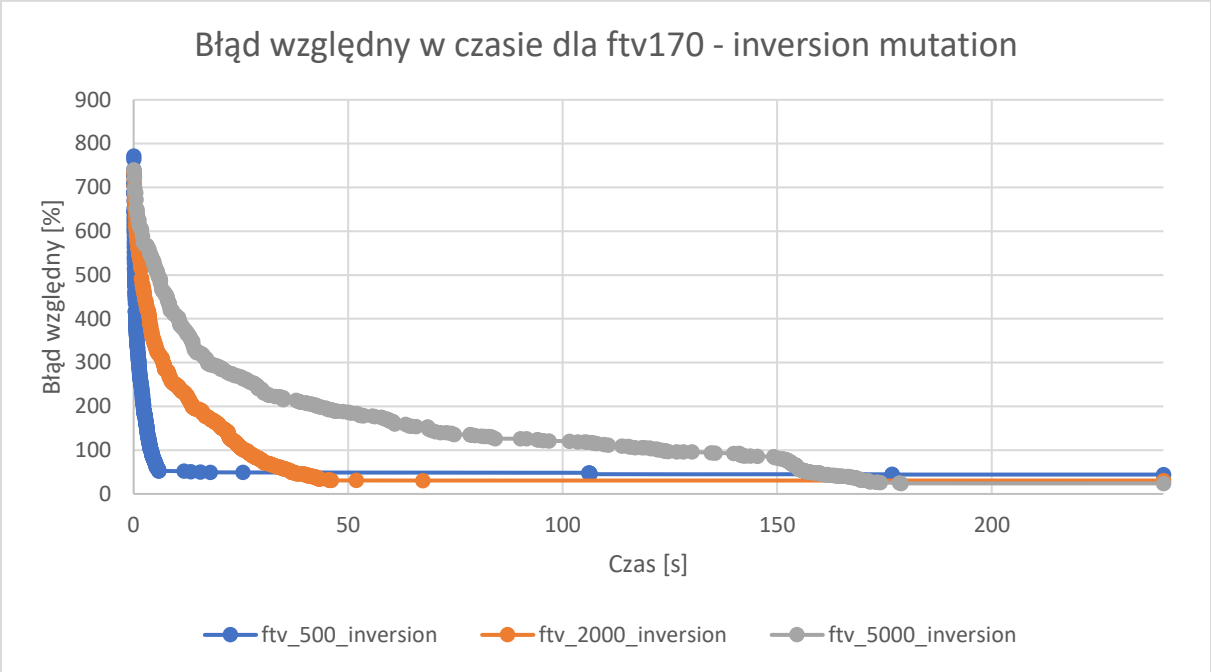
Dla obu typów mutacji, im mniejszy jest rozmiar populacji, tym szybciej algorytm wpada w minimum lokalne, po czym trudniej jest mu znaleźć lepsze wyniki.

Porównując te same rozmiary populacji największa różnica jest widoczna dla rozmiaru równego 2000, wtedy Swap znajduje lepsze wyniki znacznie szybciej niż Inversion. Dla reszty rozmiarów zmiany zachodzą w podobnym tempie.

Plik ftv170.xml

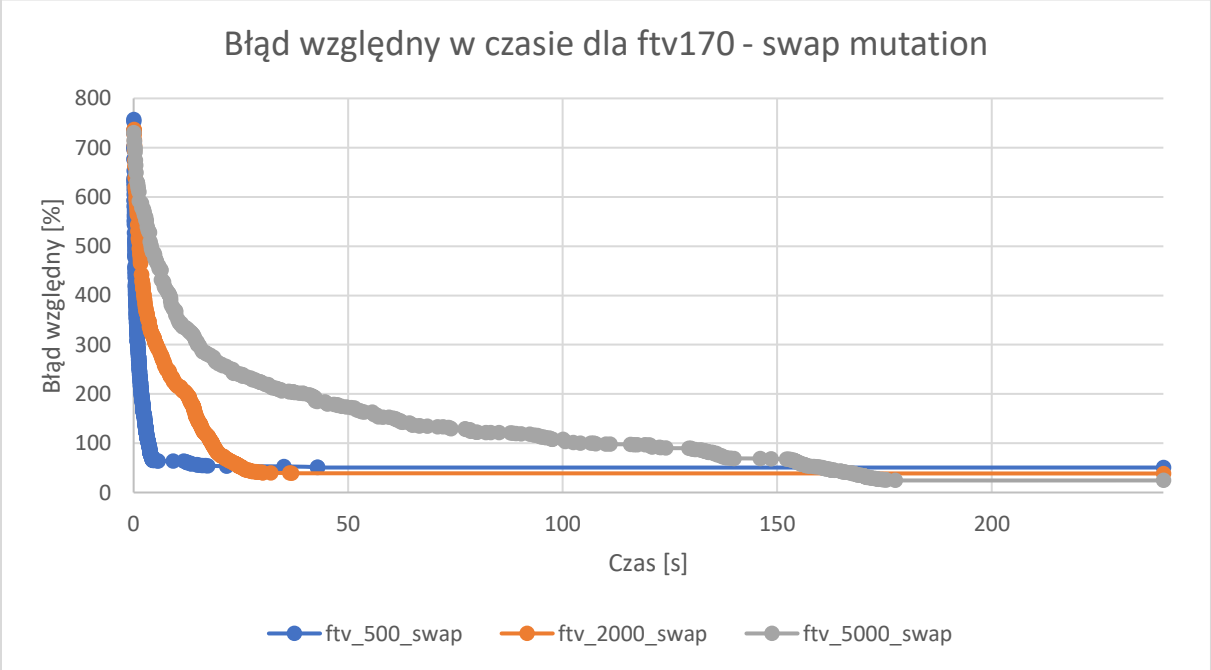
Całkowity czas trwania algorytmu = 240s.

Inversion Mutation



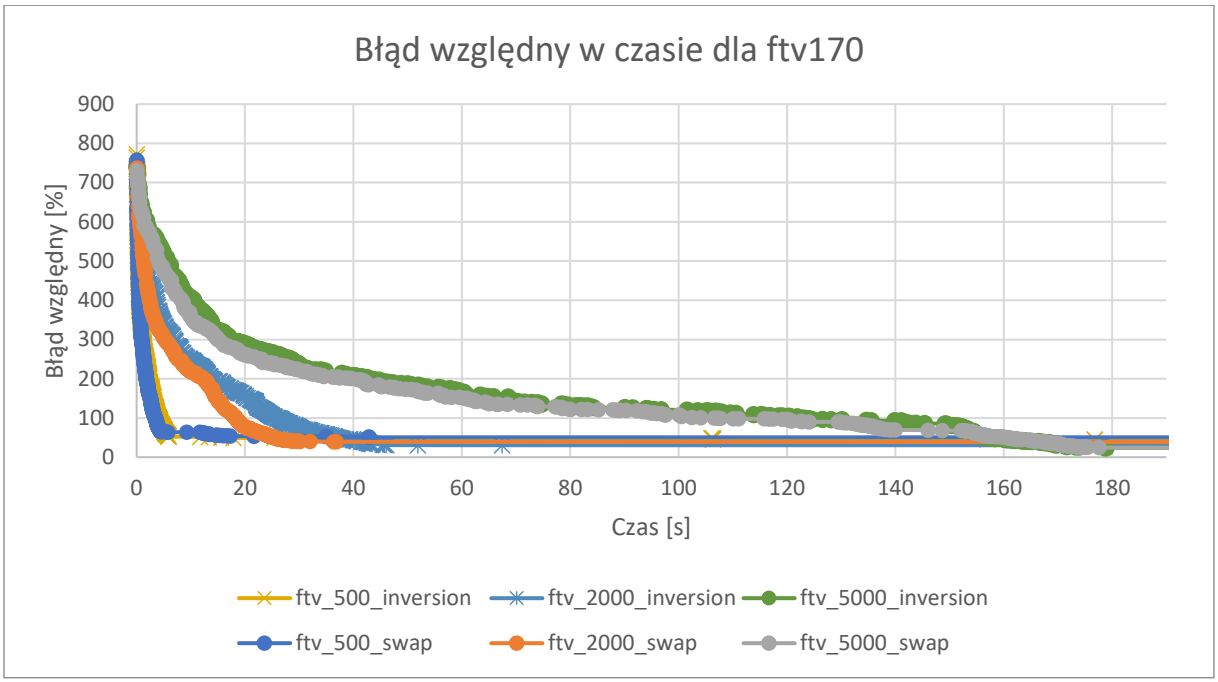
Wykres 4. Błąd względny w funkcji czasu algorytmu genetycznego z Inversion Mutation dla instancji ftv170.

Swap Mutation



Wykres 5. Błąd względny w funkcji czasu algorytmu genetycznego ze Swap Mutation dla instancji ftv170.

Porównanie wyników



Wykres 6. Wyniki skumulowane pomiarów błędu w czasie algorytmu genetycznego dla ftv170.

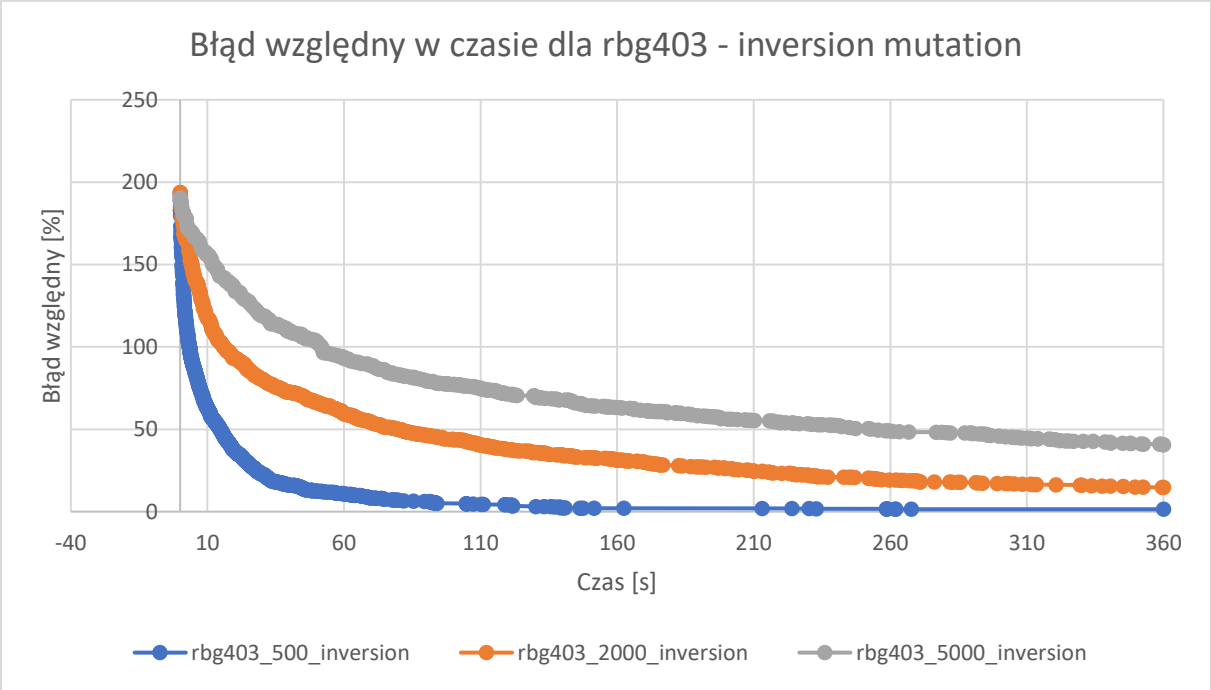
Dla obu typów mutacji, im mniejszy jest rozmiar populacji, tym szybciej algorytm wpada w minimum lokalne, po czym trudniej jest mu znaleźć lepsze wyniki.

Porównując te same rozmiary populacji największa różnica jest widoczna dla rozmiaru równego 2000, wtedy Swap znajduje lepsze wyniki znacznie szybciej niż Inversion. Dla reszty rozmiarów zmiany zachodzą w podobnym tempie.

Plik rbg403.xml

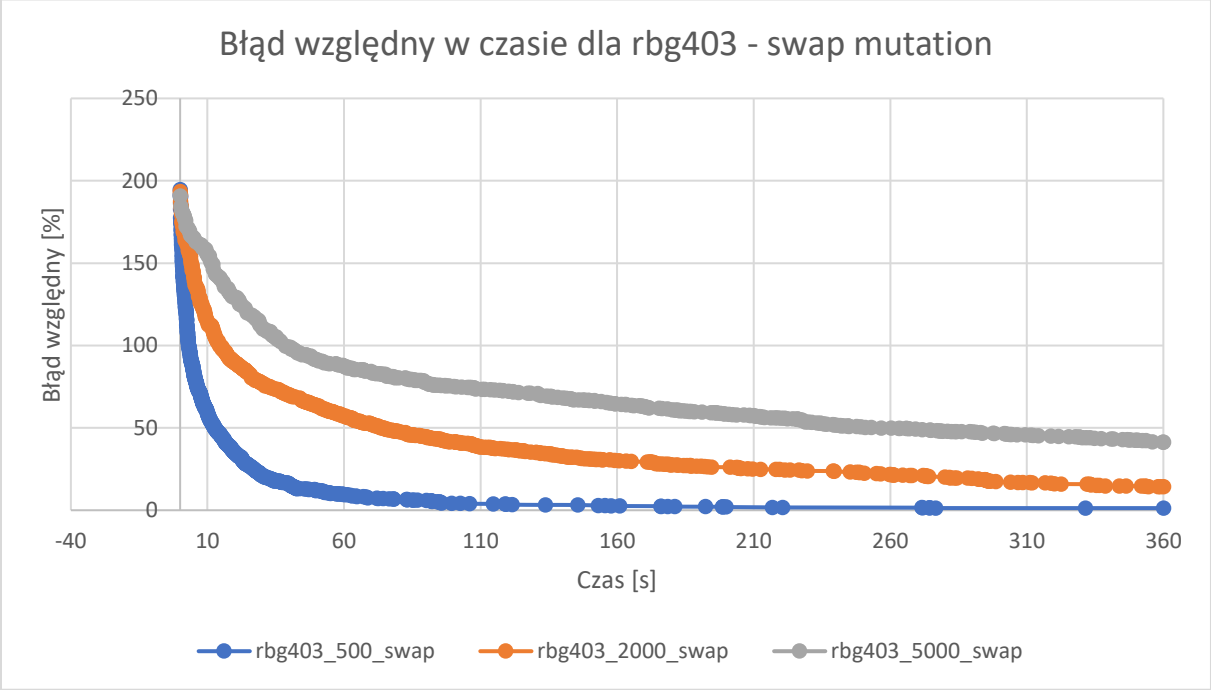
Całkowity czas trwania algorytmu = 360s.

Inversion Mutation



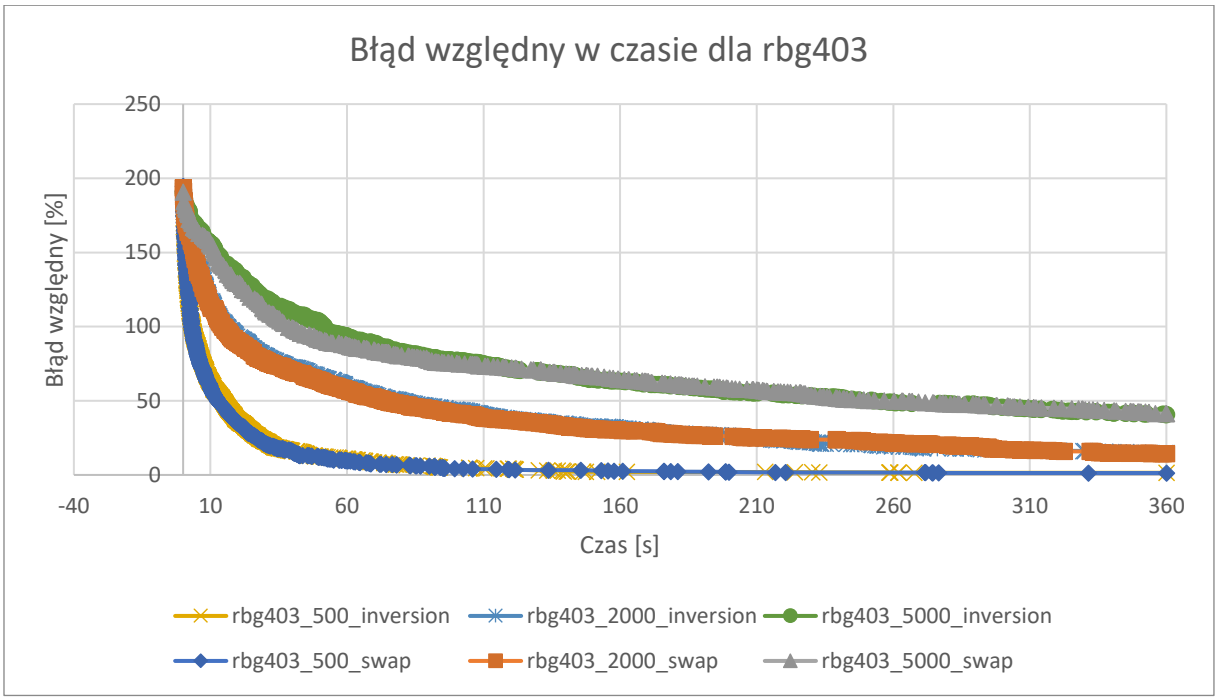
Wykres 7. Błąd względny w funkcji czasu algorytmu genetycznego z Inversion Mutation dla instancji rbg403.

Swap Mutation



Wykres 8. Błąd względny w funkcji czasu algorytmu genetycznego ze Swap Mutation dla instancji rbg403.

Porównanie wyników

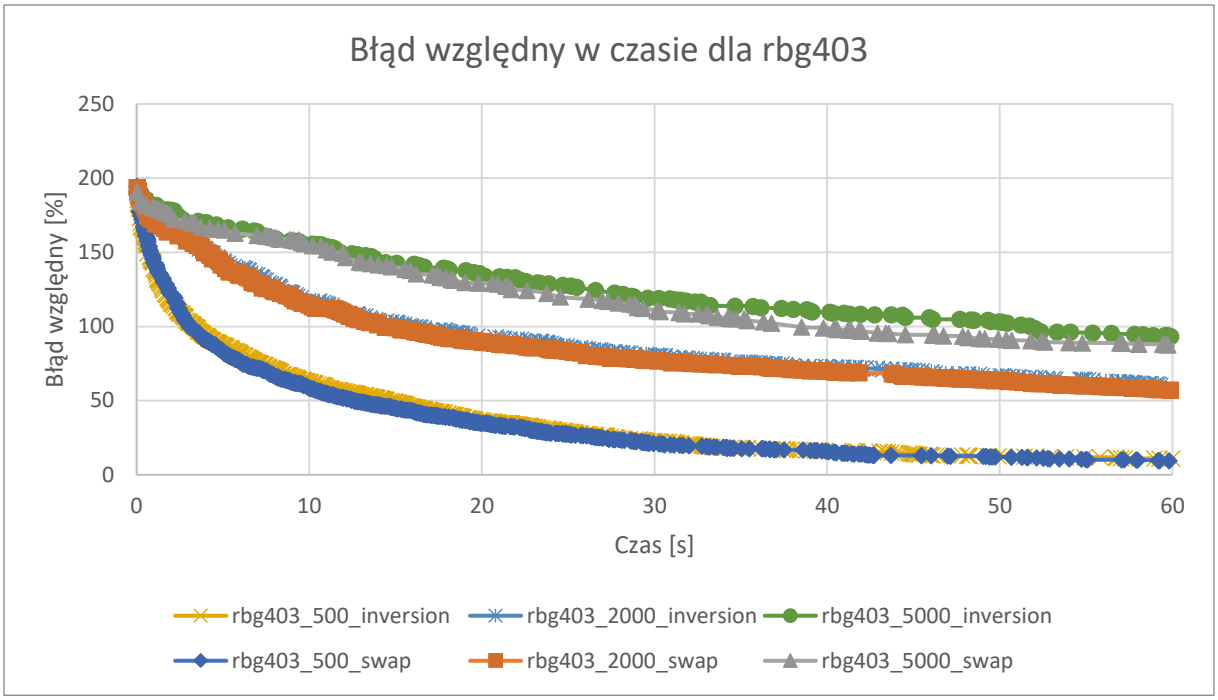


Wykres 9. Wyniki skumulowane pomiarów błędu w czasie algorytmu genetycznego dla rbg403.

Jak widać na wykresie, w ciągu działania programu, algorytm nie osiąga minimum lokalnego i generuje wciąż lepsze wyniki. Aby uzyskać lepsze wyniki należałoby zmienić kryterium stopu, czyli zwiększyć czas działania algorytmu.

Dla obu typów mutacji, im mniejszy jest rozmiar populacji, tym szybciej algorytm wpada w minimum lokalne, po czym trudniej jest mu znaleźć lepsze wyniki.

Zmiany dla obu typów mutacji w zależności od rozmiaru zachodzą porównywalnie szybko. Poniżej wykres przybliżony (pierwsze 60 sekund wykonania algorytmu).



Wykres 10. Pierwsze 60 sek. pomiaru błędu względnego w czasie działania algorytmu genetycznego dla rbg403

.

V Porównanie z algorytmem Tabu Search dla instancji ftv170

Lepsze wyniki dla instancji ftv170 osiągnięto za pomocą algorytmu Tabu Search z poprzedniego projektu. Tam błąd względny w zależności od metody generowania sąsiedztwa był na przedziale 20-30%.

Błąd względny najlepszego wyniku uzyskanego za pomocą algorytmu genetycznego wyniósł 24,03%, aczkolwiek średnie wyniki są dużo gorsze. Dla najlepszej kombinacji parametrów (ustalonych w punkcie III - Plik ftv170.xml - Porównanie wyników) uzyskano średni błąd względny na poziomie 39,778%.

Algorytm Tabu Search znajdował najlepsze rozwiązanie w o wiele szybszym czasie, który nie przekraczał 1 sek. Wpadał wtedy w lokalne minimum, z którego praktycznie nigdy już nie wychodził. Natomiast algorytm genetyczny znajdował lepsze rozwiązania przez cały czas działania. Oczywiście po wpadnięciu w minimum lokalne, znajdował je rzadziej, lecz częściej niż algorytm Tabu Search.

VI Wnioski

Dla pliku ftv47.xml i rbg403.xml algorytm potrafił znaleźć wyniki bardzo bliskie wartości optymalnej. Im dłuższy czas działania algorytmu tym lepsze wyniki znajduje, ponieważ lepiej radzi sobie z wychodzeniem z minimum lokalnego niż np. Tabu Search.

Dla rbg403 zwiększenie czasu wykonania algorytmu mogłoby znacznie wpłynąć na rezultaty, bo dla tej instancji algorytm jeszcze nie osiąga minimum lokalnego w badanym czasie.

Z instancją ftv170 algorytm genetyczny radzi sobie gorzej niż, w tym samym czasie (240sek.), za pomocą algorytmu Tabu Search.

Na zmianę wyników, poza różnymi typami mutacji, wpłynie też zmiana innych parametrów, które w tym projekcie nie zostały zbadane. Są to np. zmiana krzyżowania, zmiana współczynnika mutacji lub współczynnika krzyżowania, inny sposób selekcji, zmiana kryterium stopu.