Politechnika Wrocławska

Projektowanie efektywnych algorytmów – projekt nr 3

Autor:

Łukasz Gawron,
nr indeksu 264475

Termin oddania projektu:

22.01.2024

Termin zajęć:

Poniedziałek godz. 15:15

Prowadzący:

Dr. Inż. Jarosław Mierzwa

Spis treści

1. Wstęp

Algorytm genetyczny to rodzaj algorytmu heurystycznego inspirowany metodą doboru naturalnego. Opiera się on na takich operatorach jak selekcja, krzyżowanie i mutacja, inspirowanych biologią. W algorytmie genetycznym populacja potencjalnych rozwiązań (zwanych osobnikami) problemu optymalizacyjnego ewoluuje w kierunku uzyskania lepszych rozwiązań. Każde potencjalne rozwiązanie ma zestaw właściwości (chromosom), które może mutować. W przypadku problemu ATSP chromosom może być reprezentowany przez ścieżkę, natomiast ocena sprawności poszczególnego osobnika będzie polegać na obliczeniu kosztu tej ścieżki (Im mniejszy koszt, tym większa sprawność danego osobnika).

Klasyczny algorytm genetyczny składa się z następujących kroków:

1. Inicjacja początkowej populacji chromosomów
2. Ocena sprawności osobników początkowej populacji
3. Powtarzanie poniższych kroków w pętli aż do osiągnięcia kryterium zatrzymania
    a. Selekcja osobników w zależności od ich sprawności
    b. Zastosowanie operatorów genetycznych (krzyżowania oraz mutacji)
    c. Ocena sprawności osobników z nowej populacji
    d. Utworzenie nowej populacji potomnej (populacja potomna zastępuje obecną populację)
4. Jeśli spełnione zostało kryterium przerwania, wyodrębnione zostaje najlepsze rozwiązanie.

Algorytm genetyczny może przyjmować dodatkowo różne strategie. Jedną z nich jest strategia elitarna, w przypadku której część osobników o najlepszym przystosowaniu jest chroniona – najlepsze osobniki są zawsze włączane do reprodukcji.

Część najlepszych osobników może także przechodzić do następnej populacji (mechanizm sukcesji) niezmienionymi, tzn. bez zastosowania operatorów krzyżowania czy mutacji. Jest to strategia z częściową wymianą populacji (ang. steady-state).

Celem projektu była implementacja oraz zbadanie efektywności algorytmu genetycznego dla problemu Komiwojażera – zagadnieniu z teorii grafów, które polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Badany wariant problemu to ATSP (asymetryczny problem komiwojażera) – co oznacza, że waga ścieżki z wierzchołka A do wierzchołka B jest różna od tej z wierzchołka B do wierzchołka A. Główną trudnością problemu jest duża liczba danych do analizy - problem Komiwojażera jest problemem klasy NP (o wielomianowym czasie rozwiązywania).

2. Omówienie elementów implementacji algorytmu

Algorytm genetyczny został zaimplementowany w sposób zakładający stałą wielkość populacji w celu zachowania większej stabilności procesu ewolucyjnego oraz uproszczeniu zarządzania populacją.

2.1 Inicjalizacja populacji startowej

Początkowa populacja inicjalizowana jest poprzez generowanie losowych osobników (o losowych ścieżkach). Wykorzystano do tego metody z STL:

```
std::vector<int> path(matrixSize);
std::iota(std::begin(path), std::end(path), 0);
std::shuffle(path.begin(), path.end(), gen);
```

2.2 Ocena przystosowania

Osobniki reprezentowane są przez obiekt GASubject, przechowujący dane o chromosomie (śnieżce) oraz o koszcie ścieżki. Obiekt posiada także funkcje pomocnicze np. do wyznaczania kosztu na podstawie macierzy sąsiedztwa dla danego problemu załadowanej z pliku.

Wyznaczanie przystosowania poszczególnych osobników zostało uproszczone do obliczania kosztu cyklu Hamiltona. Im niższy koszt ścieżki, tym osobnik jest lepiej przystosowany.

2.3 Metody krzyżowania

Zaimplementowano dwie metody krzyżowania, z możliwością wyboru metody w menu programu: Order crossover oraz Edge crossover.

2.3.1 Order Crossover

Metoda ta dobrze spisuje się dla problemów takich jak TSP, gdzie ważna jest kolejność wierzchołków (genów) w chromosomie osobnika. Główną ideą tej metody jest próba zachowania względnej kolejności miast w cyklu.

Poniżej przedstawiono przykład działania:

Niech dane będą dwie sekwencje chromosomu (ścieżki) rodziców:

$$R1 = [0,1,2,3,4,5,6,7]$$

$$R2 = [7,5,6,0,3,4,1,2]$$

Utworzenie potomka rozpoczynamy od wylosowania dwóch indeksów odpowiadających elementom ścieżki. Niech $I_1 = 2$ oraz $I_2 = 5$. Następnie tworzymy nową pustą tablicę o tym samym rozmiarze, co rodzice, po czym kopiujemy fragment ścieżki między wybranymi indeksami z pierwszego rodzica do potomka.

$P = [N, N, 2,3,4,5, N, N]$, gdzie N oznacza – niezdefiniowany

Następnie inicjujemy zmienną aktualnego indeksu, $I_{current}$ równą $I_2 + 1 = 5 + 1$. Przyglądamy się teraz drugiemu z rodziców i przechodzimy na pozycję aktualnego indeksu $I_{current} = 6$.

$$R2 = [7,5,6,0,3,4,1,2]$$

Rozpoczynamy uzupełnianie potomka P od elementu, na który wskazuje aktualny indeks. Kopiujemy wierzchołki w porządku ich występowania, pomijając te wierzchołki już skopiowane z rodzica $R1$ w poprzednim kroku. Będą to kolejno wierzchołki:

- 1, dodajemy
- 2, pomijamy
- 7, dodajemy
- 5, pomijamy
- 6, dodajemy
- 0, dodajemy
- 3, pomijamy
- 4, pomijamy – przeglądanie drugiego rodzica kończymy dopiero, gdy osiągniemy aktualny indeks $I_{current} = I_2$ z pierwszego kroku.

Ostatecznie, wyznaczony potomek będzie wyglądał następująco:

$$P = [6,0,2,3,4,5,1,7]$$

Ze względu na fakt, iż po przekroczeniu indeksu ostatniego elementu tablic, należy przejść do pierwszego elementu tablic i kontynuować przeglądanie. W tym celu przydatna okazało się zastosowanie na indeksie operatora $modulo(size - 1)$, czyli w naszym wypadku modulo 8.

2.3.2 Edge Crossover

Zaimplementowana została wersja edge-3 (Whitley, 2000).

W pierwszej części algorytmu należy zbudować tablicę krawędzi. W tablicy tej dla każdego wierzchołka wyliczane są wierzchołki sąsiednie (możliwe do osiągnięcia z danego wierzchołka u obydwu rodziców). Jeżeli wyliczony wierzchołek powtarza się (sąsiaduje u obu rodziców), należy taki wierzchołek oznaczyć.

Niech indeksowanie wierzchołków w macierzy sąsiedztwa grafu rozpoczyna się od 0, wtedy indeksy tworzonej dla algorytmu tablicy krawędzi odpowiadać będą konkretnym wierzchołkom grafu.

Niech dane będą dwie sekwencje chromosomu (ścieżki) rodziców:

$$R1 = [0,1,2,3,4,5,6]$$

$$R2 = [5,6,0,3,4,1,2]$$

Tablicę krawędzi inicjujemy poprzez zainicjalizowanie pustej tablicy list, o wymierzę równym wielkości macierzy sąsiedztwa grafu (ilości wierzchołków):

```
std::vector<std::list<std::pair<int, bool>>> edgeTab(matrixSize);
```

Dla danego wierzchołka przechowywana będzie lista par, gdzie pierwszy element pary to liczba 4 bajtowa, a drugi element pary to bool (domyślnie ustawiony na ‘false’), który wskazywał będzie, czy dany wierzchołek powtórzył się u obu rodziców (zmiana wartości na ‘true’).

Tablica krawędzi dla wyżej wymienionego przykładu wyglądałaby następująco:

Indeks (element)	Krawędzie
[0]	<6, true>, <1, false>, <3, false>
[1]	<0, false>, <2, true>, <4, false>
[2]	<1, true>, <3, false>, <5, false>
[3]	<2, false>, <4, true>, <0, false>
[4]	<3, true>, <5, false>, <1, false>
[5]	<4, true>, <6, false>, <2, false>
[6]	<5, true>, <0, true>

Następnie, przystępujemy do tworzenia potomka przy wykorzystaniu owej tablicy krawędzi. Inicjujemy pustą tablicę (`std::vector<int>offspring`) o wymiarze równym wymiarze osobników rodzicielskich (ilości wierzchołków grafu). Uzupełnianie tablicy następuje według algorytmu:

Niech aktualny wierzchołek będzie oznaczony literą v.

1. v=losowy wierzchołek

Następnie powtarzaj, dopóki tablica potomka nie zostanie zapełniona (offspring.size() != matrixSize)

2. Usuń z tablicy krawędzi wszystkie odniesienia do v.
3. Zbadaj listę krawędzi dla v. Jeżeli to możliwe przejdź do
    a) wierzchołka połączonego krawędzią wspólną (występuje w parze z wartością true).
    b) wierzchołka, połączonego krawędzią nie wspólną, przy czym spośród kilku możliwości wybierz taki wierzchołek, którego lista krawędzi jest najkrótsza (w przypadku kilku list o tej samej, długości wybierz jeden z wierzchołków losowo).
4. Jeżeli lista krawędzi v jest pusta spróbuj wykonać pkt 3. dla drugiego końca ciągu).
5. Jeżeli pkt. 4 się nie udał, dodaj do listy nowy wierzchołek losowo.
6. Jeżeli nie skonstruowałeś jeszcze całej permutacji v=dodany wierzchołek i idź do punktu 2.

Należy zwrócić uwagę na fakt, iż jedynie w przypadku dotarcia punktu 5. Dodawana jest krawędź nieobecna u żadnego z rodziców, co czyni tę metodę krzyżowania lepszą od wcześniej omawianego Order Crossover.

Dla przedstawionego powyżej przykładu, poszczególne kroki będą wyglądały następująco:

Zbiór wierzchołków do wyboru	Wybrany element		Wynik
Wszystkie wierzchołki	0	Losowy wierzchołek	[0]
1, 3, 6	6	Wspólna krawędź	[0, 6]
5	5	Jedyny element	[0, 6, 5]
4, 2	4	Wspólna krawędź	[0, 6, 5, 4]
3, 1	3	Wspólna krawędź	[0, 6, 5, 4, 3]
2	2	Jedyny element	[0, 6, 5, 4, 3, 2]
1	1	Jedyny element	[0, 6, 5, 4, 3, 2, 1]

2.4 Selekcja turniejowa

Zaimplementowano metodę selekcji turniejowej, odbywającej się według następnych kroków:

1. Zdefiniowana jest wielkość przedziału – ilość elementów (osobników), jakie będą brały udział w pojedynczym turnieju. Parametr ten został dobrany eksperymentalnie i wynosi $n = \frac{\sqrt{size}}{2}$, gdzie $size$ oznacza wielkość populacji.
2. Ponieważ populacja przechowywana jest w tablicy dynamicznej o ustalonym na sztywno rozmiarze, losuje się indeks od 0 do ostatniego elementu tablicy – w przedziale 0 do $size - 1$.
3. Rozpoczynamy iterowanie po elementach populacji z przedziału $< I_r, I_r + n >$, gdzie I_r to wylosowany przez nas indeks.
4. Wybieramy najlepszego osobnika z tego przedziału (osobnika z najmniejszym kosztem – najlepszym przystosowaniem).

Następnie powtarzamy kroki od 2 do 4 w celu wyłonienia drugiego rodzica do krzyżowania.

5. Losujemy indeks struktury, w której przechowywana jest obecna populacja (w przedziale 0 do $size - 1$), w taki sposób, aby przedziały nie nachodziły na siebie
6. Rozpoczynamy iterowanie po elementach populacji z przedziału $< I_{r2}, I_{r2} + n >$, gdzie I_{r2} to wylosowany przez nas indeks.
7. Wybieramy najlepszego osobnika z tego przedziału.

W ten sposób wyłonione zostały 2 osobniki z populacji, które następnie przeznaczone będą do krzyżowania.

```
GeneticAlgorithm::tournamentSelection(std::uniform_int_distribution<> &distInt,
std::mt19937 &device) {
    int randomIndex = distInt(device);
    int secondRandomIndex;
    do {
        secondRandomIndex = distInt(device);
    } while (inRange(randomIndex, randomIndex + tournamentParticipants,
secondRandomIndex) ||
             inRange(randomIndex, randomIndex + tournamentParticipants,
secondRandomIndex + tournamentParticipants));

    GASubject a = currentPopulation[randomIndex++%(populationSize - 1)];
    int limit = randomIndex + tournamentParticipants;
    for (; randomIndex < limit; ++randomIndex) {
        int i = randomIndex % (populationSize - 1);
        if (currentPopulation[i].pathCost <= a.pathCost) {
            a = currentPopulation[i];
        }
    }

    GASubject b =currentPopulation[secondRandomIndex++%(populationSize-1)];
    int limit2 = secondRandomIndex + tournamentParticipants;
    for (; secondRandomIndex < limit2; ++secondRandomIndex) {
        int i = secondRandomIndex % (populationSize - 1);
        if (currentPopulation[i].pathCost <= b.pathCost) {
            b = currentPopulation[i];
        }
    }

    if (a.pathCost > b.pathCost) {
        return std::make_pair(b, a);
    }
    return std::make_pair(a, b);
}
```

2.5 Operator mutacji

Zastosowany został prosty operator mutacji polegający na zamianie 2 losowych wierzchołków w ścieżce (chromosomie osobnika).

1. Losowany jest indeks pierwszego wierzchołka (z przedziału od 0 do długości ścieżki-1). Następnie losuje się drugi indeks, tak aby nie były sobie równe.
2. Za pomocą funkcji z biblioteki STL – swap dokonuje się zamiana elementów na wylosowanych wcześniej pozycjach:

```
int v1 = RandomDataGenerator::generateRandomIntInRange(0, matrixSize - 1);
int v2;
do {
     v2 = RandomDataGenerator::generateRandomIntInRange(0, matrixSize - 1);
   } while (v2 == v1);
    std::swap(subject.path[v1], subject.path[v2]);
```

2.6 Tworzenie nowej populacji (populacji potomnej)

Implementacja algorytmu genetycznego uwzględnia elitarność. Jeśli współczynnik krzyżowania ustawiono na 0.8 (liczebność puli osobników do krzyżowania wyniesie 80% całkowitego rozmiaru populacji), pozostałą część nowej populacji będzie uzupełniona przez elitę, czyli 0.2 (20%) najlepszych osobników obecnej populacji zostaje bezpośrednio skopiowana do następnej populacji. 10% potomnych osobników powstałych w wyniku krzyżowania będzie poddanych mutacji.

3. Opis najważniejszych klas programu

3.1 AppController

Klasa ta, jak sama nazwa mówi, łączy infrastrukturę programu. Odpowiada za wywołanie widoku konsolowego (menu), po czym wywołuje wybrane przez użytkownika akcje tj. wywołanie algorytmu, wczytanie grafu z pliku, zmiana parametrów wejściowych algorytmu itd. W polu klasy znajdują się wskaźniki do klas zawierających algorytm genetyczny (GeneticAlgorithm) oraz klasy obudowującej macierz grafu (ATSPMatrix). Klasa wywołuje również statyczne metody klas pomocniczych – m.in. do odczytywania oraz zapisywania danych w plikach oraz pomiaru czasu.

3.2 DataFileUtility

Klasa posiada statyczne metody służące wczytaniu danych z plików, m.in. do wczytania pliku zawierającego ścieżkę. Znajdują się w niej także metody do zapisu danych do pliku: m.in. wynikowej ścieżki po uruchomieniu algorytmu oraz plików zawierających dane zbierane podczas testów algorytmów (wyniki, temperatura, znaczniku czasu znalezienia lepszego rozwiązania, poszczególne rozwiązania).

3.3 ATSPMatrix

Klasa opakowująca alokowaną dynamicznie tablicę dwuwymiarową stanowiąca macierz sąsiedztwa wczytywanych przez program grafów (z plików). Klasa ta udostępnia klasom zawierającym logikę algorytmów wskaźnik do macierzy, rozmiar oraz posiada kilka metod pomocniczych, takich jak na przykład metodę alokującą dynamicznie tablicę o rozmiarze odpowiadającym wczytywanemu grafowi oraz przypisuje zadane w nim wagi do owej tablicy.

3.4 Timer

Klasa pomocnicza służąca do pomiaru czasu. Sposób pomiaru czasu opisany został w kolejnej sekcji omawiającej plan eksperymentu. Wykorzystuje ona *QueryPerformanceCounter* z biblioteki windows.h.

3.5 RandomDataGenerator

Klasa pomocnicza, która zamiera metody służące wygenerowaniu danych pseudolosowych. Służą one m.in. do wygenerowania prawdopodobieństwa do funkcji akceptacji z zakresu $\langle 0,1 \rangle$ – typu zmiennoprzecinkowego double. Posiada również metodę generującą liczbę typu integer dla zadanego zakresu, w celu wygenerowania np. wierzchołków do zamiany w przypadku algorytmu symulowanego wyżarzania. Wykorzystuje ona generator liczb pseudolosowych z biblioteki random.

3.6 GeneticAlgorithm

Klasa zamierająca logikę odpowiedzialną za algorytm genetyczny wraz z funkcjami wykonującymi poszczególne operacje algorytmu. Szczegóły implementacyjne zostały przedstawione w punkcie nr 2.

Najważniejsze metody:

- mainFun – służy inicjalizacji parametrów
- solveTSP – zawiera w sobie główną pętlę algorytmu
- initializePopulationWithRandomPaths – służy inicjalizacji populacji początkowej osobnikami z chromosomami w postaci losowo wygenerowanych ścieżek.
- tournamentSelection – funkcja odpowiada za selekcję osobników w celu ich skrzyżowania (selekcja turniejowa)
- crossSubjects – funkcja służąca do krzyżowania dwóch osobników.
- mutate – funkcja służąca do mutacji chromosomu osobnika (ścieżki)

3.7 GASubject

Klasa reprezentująca pojedynczego osobnika populacji algorytmu genetycznego. Pole klasy stanowią chromosom (ścieżka) oraz przystosowanie (koszt ścieżki). Klasa posiada również metodę pomocniczą służącą do obliczenia kosztu ścieżki osobnika.

3.8 CrossoverMethods

Klasa posiada 2 metody implementujące algorytmy krzyżowania: OX (Order Crossover) oraz EX (Edge Crossover) wraz z funkcjami pomocniczymi.

3. Plan eksperymentu

4.1 Założenia

1. W celu zbadania efektywności zaimplementowanego algorytmu, należało zbadać jego dokładność dla podanych 3 plików opisujących niezależny problem ATSP (Asymetryczny problem komiwojażera):
   - ftv47.atsp – rozmiar mały
   - ftv170.atsp – rozmiar średni
   - rgb403.atsp – rozmiar duży
2. Czas działania algorytmów ustawiony był na sztywno:
   - 2 minuty dla rozmiaru małego,
   - 4 - średniego,
   - 8 - dużego
3. Dla każdej w wyżej wymienionych instancji problemu, algorytm został zbadany dla 3 różnych rozmiarów populacji:
   - 100 osobników
   - 250 osobników
   - 600 osobników
4. Dla każdego pliku oraz zadanej wielkości populacji algorytm uruchomiony został 10 razy. Wyniki umieszczone zostały w dalszej części sprawozdania.

4.2 Sposób generowania liczb pseudolosowych oraz pomiaru czasu

Generator liczb losowych wykorzystywał maszynę losującą: 32-bit Mersenne twister engine z biblioteki random, czyli *std::mt19937 gen.*

Listing 1. Metoda służąca do wygenerowania liczby pseudolosowej z podanego zakresu.

```
int RandomDataGenerator::generateVertexInRange(int from, int to) {
    std::random_device rdev;
    std::mt19937 gen(rdev());
    std::uniform_int_distribution<> dist(from, to);
    return dist(gen);
}
```

W celu uzyskania dużej dokładności pomiarowej czasu (o rozdzielczości mikrosekundowej), wykorzystana została klasa *QueryPerformanceCounter().*

Listing 2. Funkcja służąca odczytaniu licznika procesora.

```
long long int Timer::read_QPC() {
    LARGE_INTEGER count;
    QueryPerformanceCounter(&count);
    return ((long long int) count.QuadPart);
}
```

W ten sposób zmierzony został stan licznika przed rozpoczęciem testowanej operacji oraz po jej wykonaniu. Następnie, Końcowy rezultat uzyskuje się przez podzielenie odmierzonej liczby impulsów licznika przez częstotliwość, którą otrzymuje się za pomocą wywołania *QueryPerformanceFrequency().*

Listing 3. Funkcja wyznaczająca czas między dwoma zmierzonymi punktami pomiaru licznika procesora.

```
double Timer::getMicroSecondsElapsed(long long int start, long long int
end) {
    long long int elapsed, frequency;
    elapsed = end - start;
    QueryPerformanceFrequency((LARGE_INTEGER *) &frequency);
    return ((1000000.0 * elapsed) / frequency);
}
```

4.3 Ogólny plan przeprowadzania eksperymentu

- Algorytm został przetestowany dla 3 zadanych instancji problemu zawartych w plikach ftv47.atsp, ftv170.atsp oraz rgb403.atsp. Instancje problemu stanowią kolejno problem mały (przyjęto czasowe kryterium stopu 2 minuty), średni (przyjęto czasowe kryterium stopu 4 minuty) oraz duży (przyjęto czasowe kryterium stopu 8 minut). Algorytm dla zadanej instancji zostały uruchomione po 10 razy w celu otrzymania uśrednionych wyników, ponadto działanie algorytmu zostało zbadanie dla 3 różnych rozmiarów populacji.

- Poniżej przedstawione wyniki pomiarów to wartości uśrednione na podstawie 10 przebiegów algorytmu dla podanego pliku .atsp.

- Dla algorytmu zapisywany był czas znalezienia najlepszego rozwiązania w danym przebiegu. Wartości pomiarów czasu przedstawione zostały w jednostce Milisekund (ms) oraz sekund (s).

- Błąd względny wyznaczony został na podstawie najlepszych znalezionych rozwiązań oraz najlepszego znanego rozwiązania dla danego grafu:
  $\frac{|f_{zn}-f_{opt}|}{f_{zn}}$, gdzie f_{zn} – najlepsza wartość obliczona przez nasz algorytm, f_{opt} – wartość optymalna (najlepsze znane rozwiązanie).

5. Wyniki eksperymentów

Tabele przedstawiają wyniki dla poszczególnych przebiegów algorytmu genetycznego (5) dla każdego z trzech wyżej wymienionych rozmiarów populacji startowej: Czas odnalezienia najlepszego rozwiązania (s), Najniższy koszt osobnika z populacji startowej, Najlepszy odnaleziony koszt w wyniku działania algorytmu oraz jego błąd względny.

- Współczynnik mutacji: **0.01**
- Współczynnik krzyżowania **0.8**

5.1 Problem ftv47 (liczba wierzchołków grafu – 48)

Poniżej przedstawione zostały wyniki dla danych testowych z pliku ftv47 (najlepsze znane rozwiązanie: 1776).

- Kryterium czasowe zakończenia algorytmu ustawiono na 2 minuty.

5.1.1 Metoda krzyżowania OX

Rozmiar populacji: **100**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	51.76	6044	2193	23.5%
2.	32.58	5876	**2186**	23.1%
3.	1.14	6062	2422	36.4%
4.	94.49	6001	2263	27.4%
5.	105.79	6093	2314	30.3%

Rozmiar populacji: **250**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	63.68	5592	**1947**	9.6%
2.	35.97	5396	2140	20.5%
3.	16.39	5681	2136	20.3%
4.	15.18	5765	2125	19.7%
5.	15.2	5553	2147	20.9%

Rozmiar populacji: **600**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	3.22	5603	**2093**	17.8%
2.	29.05	5825	2103	18.4%
3.	0.77	5825	2126	19.7%
4.	1.72	5483	2128	19.8%
5.	1.96	5759	2157	21.5%

5.1.2 Metoda krzyżowania EX

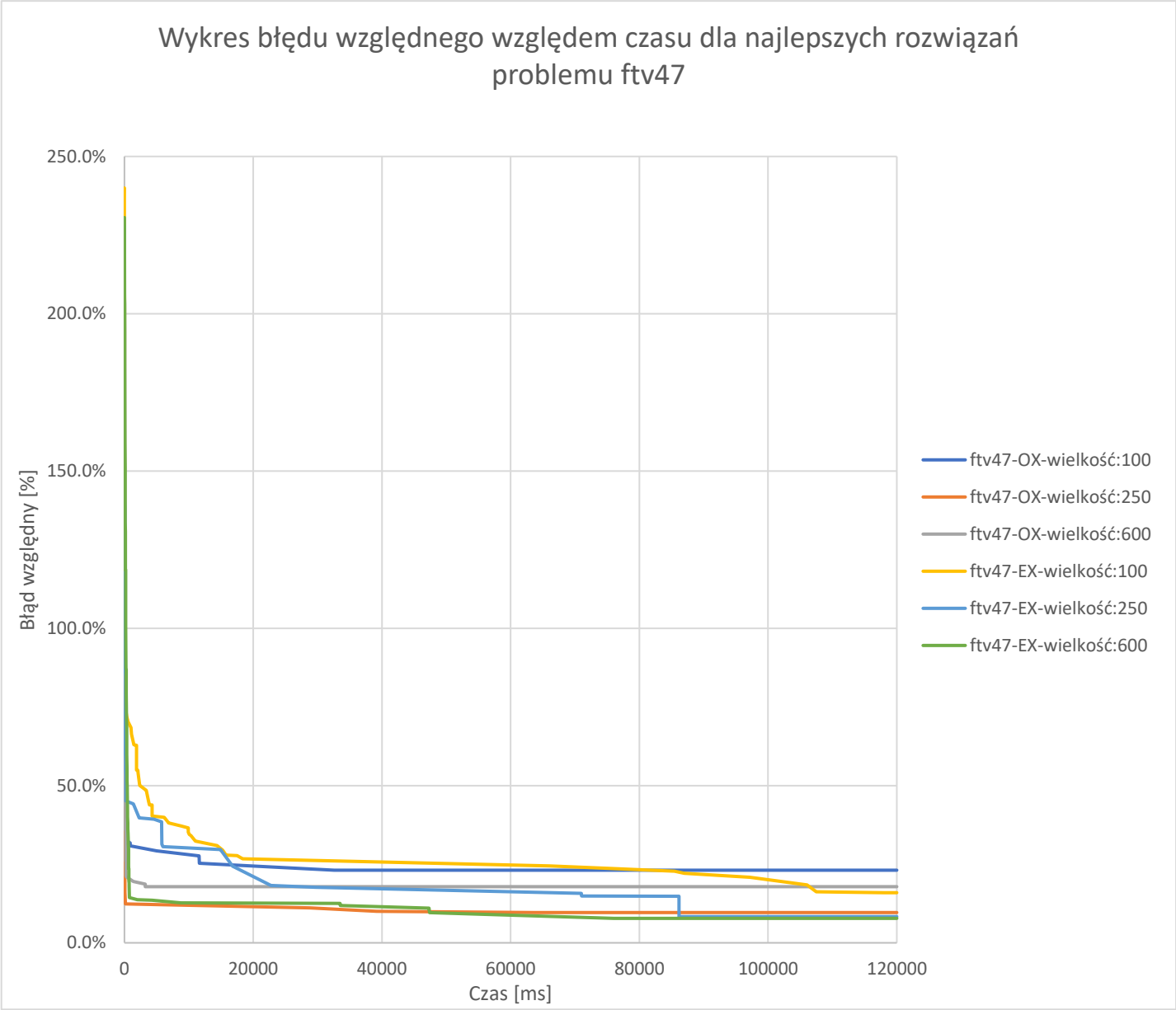
Rozmiar populacji: **100**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	118.03	6038	**2059**	15.9%
2.	34.03	5492	2253	26.9%
3.	119.10	5880	2351	32.4%
4.	51.17	6014	2240	26.1%
5.	91.79	5705	2174	22.4%

Rozmiar populacji: **250**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	118.05	5599	2044	15.1%
2.	86.17	5823	**1924**	8.3%
3.	47.90	5931	2043	15.0%
4.	50.44	5991	2076	16.9%
5.	117.27	5689	1975	11.2%

Rozmiar populacji:**600**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	29.17	5868	2131	20.0%
2.	44.67	5906	2053	15.6%
3.	76.03	5871	**1914**	7.8%
4.	103.04	5490	2128	19.8%
5.	70.04	5854	2091	17.7%

5.1.3 Wykres błędu względnego dla najlepszych rozwiązań ftv47

Poniżej przedstawiony został wykres obrazujący przebieg funkcji błędu względnego w czasie dla najlepszych przebiegów algorytmu dla danego rozmiaru populacji oraz metody krzyżowania.



5.2 Problem ftv170 (liczba wierzchołków grafu – 171)

Poniżej przedstawione zostały wyniki dla danych testowych z pliku ftv170 (najlepsze znane rozwiązanie: 2755).

- Kryterium czasowe zakończenia algorytmu ustawiono na 4 minuty (taki sam czas jak w przypadku algorytmu Tabu Search dla tego pliku).

5.2.1 Metoda krzyżowania OX

Rozmiar populacji: **100**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	194.94	24499	6578	138.8%
2.	81.65	24742	6217	125.7%
3.	193.97	24600	**5427**	97.0%
4.	239.12	24292	6065	120.1%
5.	161.43	24445	6151	123.3%

Rozmiar populacji: **250**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	142.42	24342	6355	130.7%
2.	232.28	24197	5742	108.4%
3.	226.60	24356	5582	102.6%
4.	88.09	24238	**5577**	102.4%
5.	187.75	23903	6156	123.4%

Rozmiar populacji: **600**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	204.16	23831	5712	107.3%
2.	222.01	23798	5475	98.7%
3.	145.00	23813	**4888**	77.4%
4.	106.04	24321	5358	94.5%
5.	196.87	23842	5329	93.4%

5.2.2 Metoda krzyżowania EX

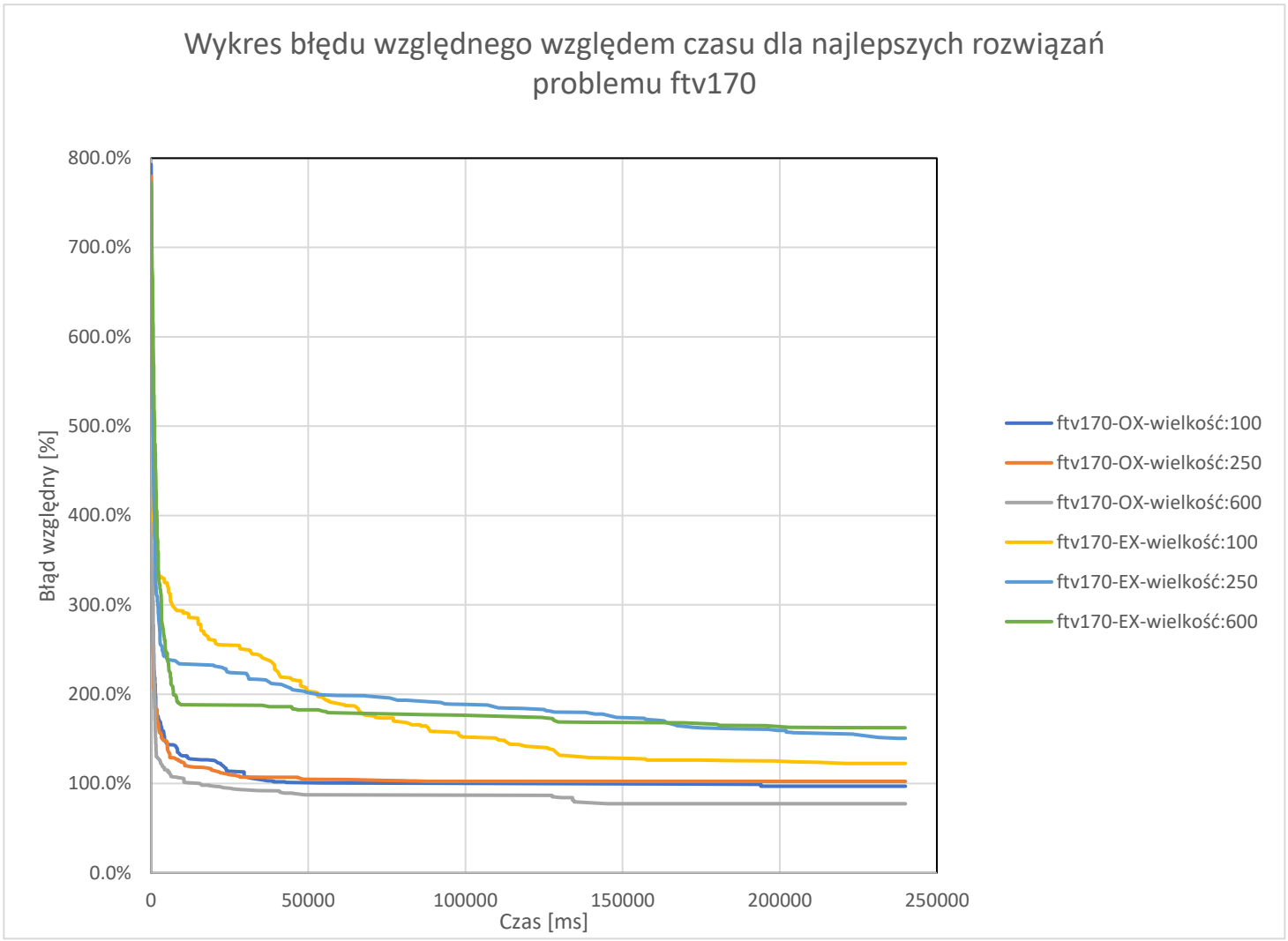
Rozmiar populacji: **100**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	237.42	24479	6597	139.5%
2.	235.37	24159	6166	123.8%
3.	237.32	24290	6408	132.6%
4.	220.85	24010	**6129**	122.5%
5.	239.71	24314	6595	139.4%

Rozmiar populacji: **250**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	237.60	23488	**6904**	150.6%
2.	229.49	24035	7195	161.2%
3.	231.50	23808	7689	179.1%
4.	237.70	24060	6993	153.8%
5.	235.30	24248	7361	167.2%

Rozmiar populacji: **600**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	239.88	24042	**7232**	162.5%
2.	214.76	24054	7595	175.7%
3.	210.50	24268	7252	163.2%
4.	229.07	23966	7639	177.3%
5.	233.64	23943	7714	180.0%

5.2.3 Wykres błędu względnego dla najlepszych rozwiązań ftv170

Poniżej przedstawiony został wykres obrazujący przebieg funkcji błędu względnego w czasie dla najlepszych przebiegów algorytmu dla danego rozmiaru populacji oraz metody krzyżowania.



5.3 Problem rbg403 (liczba wierzchołków grafu- 403)

Poniżej przedstawione zostały wyniki dla danych testowych z pliku rbg403 (najlepsze znane rozwiązanie: 2465).

- Kryterium czasowe zakończenia algorytmu ustawiono na 8 minut.

5.3.1 Metoda krzyżowania OX

Rozmiar populacji: **100**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	359.02	7406	2526	2.5%
2.	472.48	7367	2533	2.8%
3.	479.80	7372	2523	2.4%
4.	474.49	7368	2582	4.7%
5.	405.82	7343	**2508**	1.7%

Rozmiar populacji: **250**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	463.53	7285	2477	0.5%
2.	447.65	7338	2484	0.8%
3.	411.46	7195	2509	1.8%
4.	317.52	7280	2476	0.4%
5.	405.74	7329	**2474**	0.4%

Rozmiar populacji: **600**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	330.33	7262	2489	1.0%
2.	458.07	7166	2502	1.5%
3.	167.03	7276	2481	0.6%
4.	382.55	7286	**2475**	0.4%
5.	474.01	7322	2487	0.9%

5.3.2 Metoda krzyżowania EX

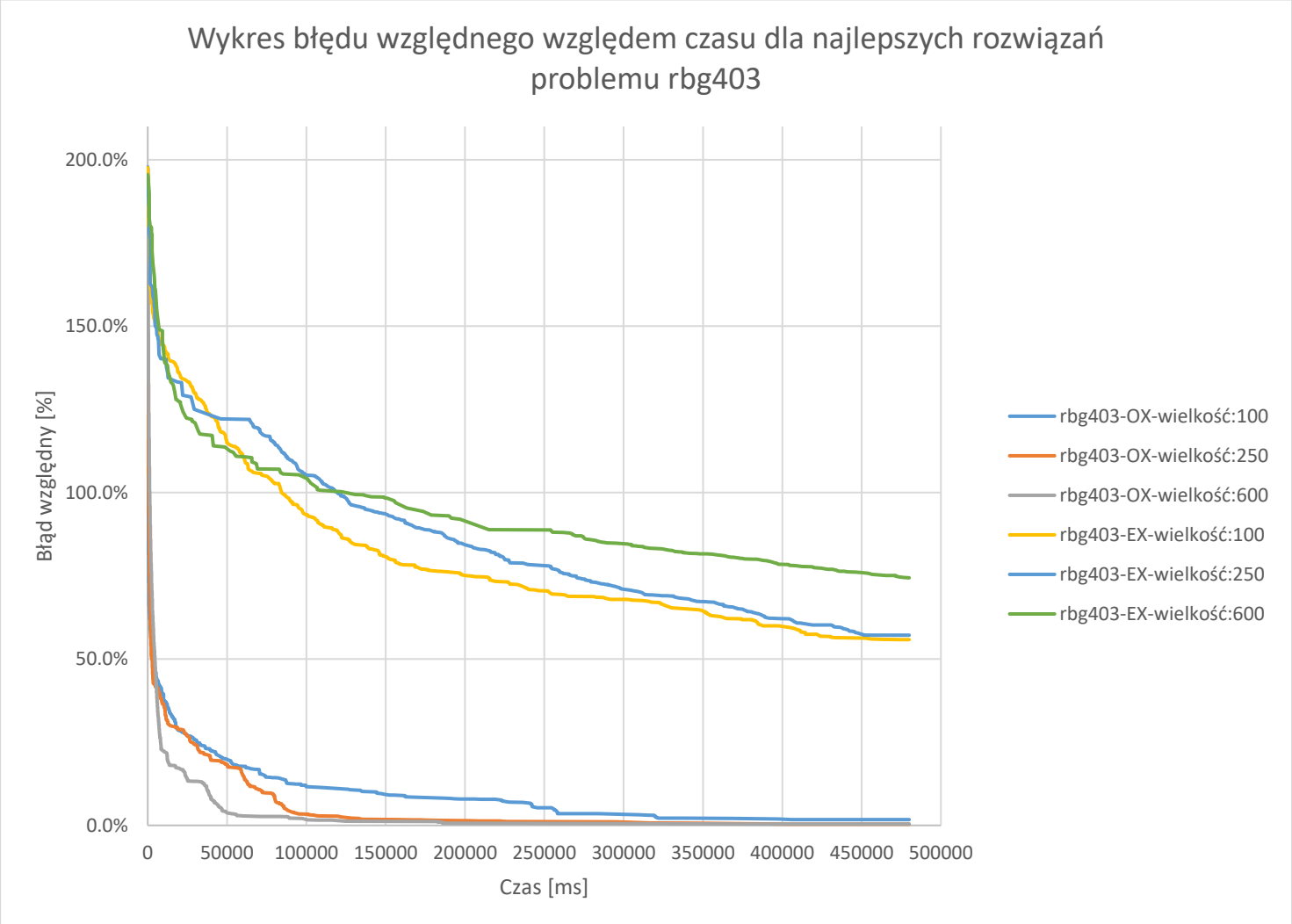
Rozmiar populacji: **100**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	478.63	7306	3896	58.1%
2.	474.10	7337	**3841**	55.8%
3.	479.53	7402	3972	61.1%
4.	472.89	7344	3873	57.1%
5.	464.46	7398	3914	58.8%

Rozmiar populacji: **250**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	479.92	7220	3952	60.3%
2.	473.14	7176	4029	63.4%
3.	458.88	7273	**3874**	57.2%
4.	476.18	7332	3918	58.9%
5.	478.70	7204	3956	60.5%

Rozmiar populacji: **600**				
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Najlepszy koszt początkowy	Najlepsze odnalezione rozwiązanie	Błąd względny
1.	479.31	7285	**4299**	74.4%
2.	479.29	7283	4444	80.3%
3.	474.02	7250	4476	81.6%
4.	479.64	7317	4355	76.7%
5.	478.85	7310	4303	74.6%

5.3.3 Wykres błędu względnego dla najlepszych rozwiązań rbg403

Poniżej przedstawiony został wykres obrazujący przebieg funkcji błędu względnego w czasie dla najlepszych przebiegów algorytmu dla danego rozmiaru populacji oraz metody krzyżowania.



6. Porównanie z algorytmem przeszukiwania tabu (Tabu Search)

Poniżej przedstawione zostały wyniki dla algorytmu Tabu Search dla instancji ftv170.atsp, które przeprowadzono przy okazji realizacji poprzedniego projektu. W wykonanej implementacji przeszukiwania tabu zrealizowano jedną definicję sąsiedztwa, w której zbiór rozwiązań sąsiednich stanowią ścieżki z zamienionymi ze sobą 2 wybranymi wierzchołkami.

Czas działania algorytmu dla tego problemu ustawiono na 4 minuty.

	Rozmiar grafu: 171			
Lp.	Czas znalezienia najlepszego rozwiązania [s]	Koszt ścieżki wyznaczonej z algorytmu zachłannego	Najlepsze znalezione rozwiązanie obliczone przez algorytm	Błąd względny
1	178.36	3582	3241	17.64%
2	228.58	3582	**3226**	17.10%
3	134.76	3582	3253	18.08%
4	43.063	3582	3265	18.51%
5	227.57	3582	3253	18.08%
6	78.010	3582	3229	17.21%
7	141.93	3582	**3226**	17.10%
8	171.51	3582	3251	18.00%
9	143.87	3582	3249	17.93%
10	217.24	3582	3238	17.53%

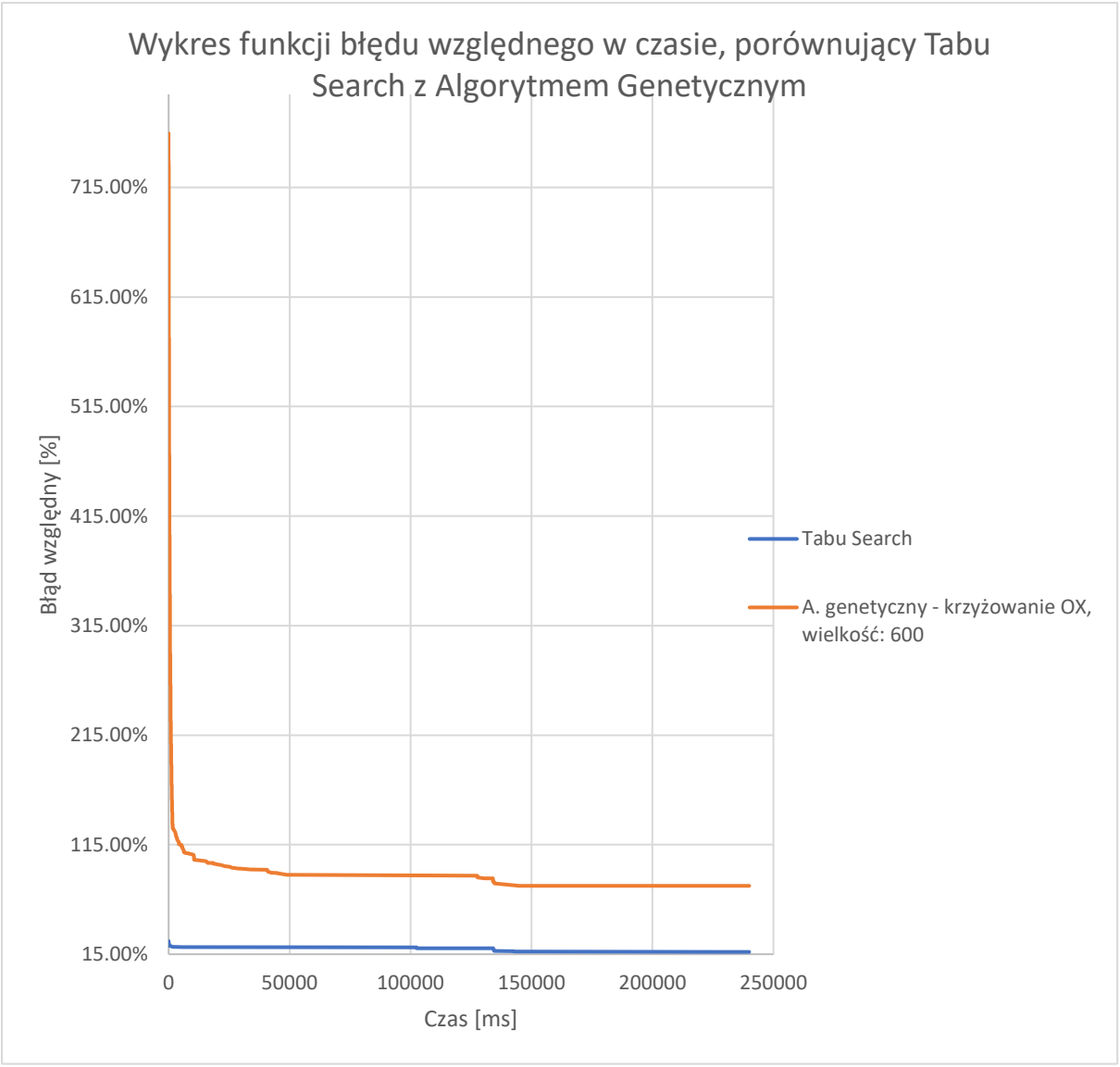
- Najlepsze znalezione rozwiązanie przez algorytm Tabu Search: 3226, błąd względny: 17.10%
- Najlepsze znalezione rozwiązanie przez algorytm genetyczny: 4888, błąd względny: 77.4%

Z eksperymentu wyniknęło jednoznacznie, że algorytm przeszukiwania tabu daje lepsze rezultaty w tym samym czasie (240 sekund) od algorytmu genetycznego.

Efektywność obu algorytmów może się znacznie różnić w zależności od instancji problemu, na przykład dla danych z pliku rbg403 algorytm genetyczny z metodą krzyżowania OX dawał bardzo dobre rezultaty – możliwe że dla danych testowych z pliku ftv170 wymagane jest lepsze dobranie parametrów lub zastosowanie bardziej zaawansowanych technik selekcji i mutacji. Zaletą algorytmu genetycznego jest natomiast jego wysoka zdolność do eksploracji przestrzeni rozwiązań.

Jeżeli przestrzeń poszukiwań problemu posiada liczne lokalne optimum, algorytm Tabu Search może być bardziej skuteczny w unikaniu lokalnych minimów i znaleźć lepsze rozwiązania, ponieważ skupia się na eksploatacji istotnych obszarów przestrzeni (cykliczne lub częściowe przeglądanie sąsiedztwa). Lista tabu pomaga w unikaniu powrotu do już odwiedzonych rozwiązań. Algorytm przeszukiwania tabu wydaje się także mniej złożony i łatwiejszy do optymalizacji pod konkretny problem (jak w wypadku tego projektu - problem Komiwojażera), gdyż nie zawiera tylu mechanizmów ile algorytm genetyczny.

Dostosowanie tych mechanizmów (selekcja, krzyżowanie, mutacja i elityzm) pod instancję problemu ftv170 mogłoby znacznie polepszyć wyniki algorytmu genetycznego, jednak proces jego optymalizacji jest trudniejszy niż w przypadku algorytmu przeszukiwania tabu.



7. Wnioski

Dla grafu małego (ftv47) najlepsze rezultaty otrzymywano dla rozmiaru populacji 250 oraz 600, zarówno w przypadku metody krzyżowania OX, jak i metody krzyżowania EX, jednakże można zauważyć przewagę metody krzyżowania EX – wyniki są średnio o kilka procent niższe, niż w przypadku metody OX dla tych samych rozmiarów populacji. Najlepszy otrzymany wynik dla tego grafu także otrzymano dla metody EX i rozmiaru populacji 600. Obserwujemy również spory rozstrzał wartości czasów odnalezienia najlepszego rozwiązania. W przypadku metody krzyżowania OX jest to przedział od nieco ponad 1 sekundy do 105 sekund. Dla metody krzyżowania EX statystyka ta prezentuje się lepiej, ponieważ czasy odnalezienia najlepszego rozwiązania pochodzą z przedziału od około 30 sekund do aż 118, więc wywnioskować można że metoda EX jest bardziej skuteczna dla tego grafu, gdyż wydaje się lepiej unikać minima lokalne. W przypadku metody OX obserwujemy często szybkie „nasycenie” się algorytmu, przez co przez większość czasu nie działa on prawidłowo.

W przypadku grafu średniego (ftv170) wyniki są zupełnie inne, ponieważ metoda krzyżowania OX okazała się znacznie bardziej skuteczna. Dla metody OX najlepszy wynik, tak jak dla grafu małego, otrzymano dla największego rozmiaru populacji (600) – 77.4% błędu względnego. Czasy odnalezienia najlepszego rozwiązania także potwierdzają skuteczność eksploracji algorytmu, gdyż pochodzą z przedziału 80 sekund do 240 sekund (aż do osiągnięcia kryterium przerwania). Gorzej natomiast wyglądają wyniki dla metody krzyżowania EX, gdzie najlepszy rezultat otrzymano dla najmniejszego rozmiaru populacji (100) – 122.5% błędu względnego, pomimo iż czasy odnajdywania najlepszego rozwiązania są bliskie wartości kryterium przerwania. Powodem takiego stanu rzeczy jest najprawdopodobniej znacznie większa złożoność obliczeniowa metody krzyżowania EX. Metoda ta wymaga korzystania z dodatkowych struktur, takich jak tablica sąsiedztwa, na której wykonujemy operacje wyszukiwania najlepiej dopasowanych sąsiadów, więc złożoność obliczeniowa oraz pamięciowa rośnie znacząco wraz z ilością zmiennych (wierzchołków) problemu, przez co algorytm genetyczny wykonuje znacznie mniej iteracji niż w przypadku metody OX. Można zatem wywnioskować, że kryterium czasowe na poziomie 4 minut dla danych testowych ftv170 (grafu o 171 wierzchołkach) jest zbyt niskie w przypadku metody krzyżowania EX – algorytm jest zbyt wcześnie przerywany.

Dla grafu dużego (rbg403) obserwujemy podobne rezultaty jak w przypadku grafu średniego. Najlepsze wyniki w przypadku metody krzyżowania OX otrzymano dla rozmiarów populacji wynoszonych 250 oraz 600 osobników (zwiększenie rozmiaru populacji z 250 na 600 nie ma widocznego wpływu na polepszenie się wyników, odwrotnie niż w przypadku zwiększenia rozmiaru populacji ze 100 na 250 gdzie widzimy znaczące polepszenie się wyników). Najlepszy rezultat otrzymano dla rozmiaru populacji wynoszącego 250 – błąd względny na poziomie 0.4%, jednakże jest to raczej kwestia losowości algorytmu, ponieważ rezultaty dla rozmiaru populacji 600 są równie dobre. W przypadku metody krzyżowania EX obserwujemy identyczną zależność, jak dla grafu ftv170, mianowicie im większy rozmiar populacji, tym gorsze wyniki otrzymujemy. Spowodowane jest to faktem, iż kryterium czasowe na poziomie 8 minut jest zbyt niskie dla metody krzyżowania EX (ze względu na jej większą złożoność czasową w porównaniu z metodą OX), przez co algorytm przerywany jest zbyt wcześnie – przed momentem jego „nasycenia”.

8. Literatura i źródła

1. https://www.cs.put.poznan.pl/mkomosinski/lectures/optimization/MK_AlgEwolucyjne.pdf
2. https://home.agh.edu.pl/~horzyk/lectures/biocyb/BIOCYB-AlgorytmyGenetyczneEwolucyjne.pdf
3. https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf
4. https://www.obitko.com/tutorials/genetic-algorithms/selection.php