

Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Projektowanie Efektywnych Algorytmów
Zadanie Projektowe 3

Autor:

Stanisław Strauchold 259142

Prowadzący:

dr inż. Jarosław Mierzwa

Grupa:

Środa 11:15 – 13:00 – grupa wcześniejsza

Termin oddania:

18.01.2023

1. Wstęp teoretyczny

Algorytm genetyczny jest algorytmem polegającym na zdefiniowaniu pewnego środowiska, w którym istnieje populacja osobników, która jest zbiorem rozwiązań problemu (w naszym przypadku jest to problem komiwojażera). Każdy z osobników przechowuje informacje konieczne do wyliczenia rozwiązania problemu, dla którego wykonywany jest algorytm. Informacje te zwane są genotypem. Algorytm działa analogicznie do pojęcia zwanego ewolucją, to znaczy najsilniejsze osobniki krzyżują się ze sobą w celu stworzenia osobnika o genotypie najlepiej przystosowanym do przetrwania w środowisku, w którym się znajduje. W problemie komiwojażera miara siły oceniana jest przez długość ścieżki, którą zawiera dany osobnik. Im krótsza ścieżka, tym silniejszy jest osobnik. Ogólny zarys algorytmu polega na tworzeniu kolejnych populacji, z których każda będzie silniejsza od poprzedniej.

Zaimplementowany przeze mnie algorytm w postaci listy kroków prezentuje się następująco:

Krok 1. Utworzenie populacji początkowej.

Krok 2. Selekcja osobników i stworzenie ich potomków.

Krok 3. Dodanie potomków do nowej populacji.

Krok 3. Mutacja części osobników.

Krok 4. Zastąpienie populacji początkowej nową populacją .

Krok 5. Znalezienie najsilniejszego osobnika.

Krok 6. Powrót do Kroku 2.

1.1. Utworzenie populacji początkowej

Utworzenie populacji początkowej polega na wygenerowaniu zadanej liczby osobników. Dla każdego osobnika losowana jest ścieżka odpowiadająca kolejności odwiedzanych miast oraz obliczana jest suma odległości między miastami, która stanowi siłę osobnika.

1.2. Selekcja osobników oraz krzyżowanie.

Wybór osobników, które zostaną poddane krzyżowaniu polega na przeprowadzeniu między nimi turnieju. Z populacji losowanych jest $wspolczynnik_krzyzowania * wielkosc_populacji$ osobników, następnie wybierany jest najlepszy z nich (posiadający najkrótszą długość ścieżki), zwany dalej rodzicem. Następnie w ten sam sposób wybierany jest drugi rodzic. Na wybranych osobnikach przeprowadza się krzyżowanie w celu stworzenia ich potomków. Z każdej pary rodziców powstaje dwóch potomków. Następnie powtarza się cały proces, do momentu wyboru liczby rodziców odpowiadającej liczbie osobników w populacji. Metoda nie pozwala wybrać w tej samej iteracji dwóch tych samych osobników.

1.3. Krzyżowanie

Po wybraniu dwóch rodziców następuje operacja krzyżowania. Jednak zanim to nastąpi losowana jest liczba z przedziału 0.01 do 0.99, która jest porównywana z wprowadzonym przez użytkownika współczynnikiem krzyżowania. Jeżeli jest od niego większa, pomijamy

krzyżowanie, a do nowej populacji dodajemy wybranych rodziców. W programie zostało zaimplementowane krzyżowanie za pomocą operatora OX (*ang. Order crossover*). Jego działanie zostanie wyjaśnione na przykładzie.

Założmy, że wybrani rodzice, oznaczeni jako R1 oraz R2, posiadają następujące ścieżki reprezentujące kolejność odwiedzanych przez komiwojazzera miast:

R1:

5	6	7	3	4	1	2
---	---	---	---	---	---	---

R2:

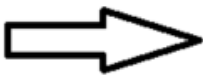
6	2	3	5	1	4	7
---	---	---	---	---	---	---

W sposób losowy wybierane są dwa punkty krzyżowania (indeksy zawierające się w tablicy reprezentującej ścieżkę). Na ich podstawie wybierany jest segment, który następnie jest kopiowany z R1 do pierwszego potomka zwanego dalej P1.

Założmy, że wylosowane liczby to 3 oraz 6. Reprezentują one indeksy 3 oraz 6 w tablicy ze ścieżką. Co za tym idzie z R1 kopiowane są elementy o indeksach 3, 4 oraz 5. Wstawiamy je na tych samych pozycjach w P1:

R1:

5	6	7	3	4	1	2
---	---	---	---	---	---	---


 P1:

			3	4	1	
--	--	--	---	---	---	--

Następnie rozpoczynamy kopiowanie elementów z R2, rozpoczynając od indeksu 6, czyli drugiej z wylosowanych wcześniej liczb. Jeśli dana liczba znajdowała się w skopiowanym segmencie, pomijamy ją i bierzemy następną. Jeśli dojdziemy do końca tablicy w R2 lub w P1, przechodzimy na jej początek. Kopiujemy elementy do momentu zapełnienia całej tablicy P1.

R2:

6	2	3	5	1	4	7
---	---	---	---	---	---	---

 P1:


			3	4	1	7
--	--	--	---	---	---	---

Liczba 7 nie znajduje się w skopiowanym na początku segmencie, zatem kopiujemy ją do P1 na kolejny wolny indeks.

Z racji na dojście do końca tablicy w R2 oraz w P1, przechodzimy na ich początek.

R2:

6	2	3	5	1	4	7
---	---	---	---	---	---	---

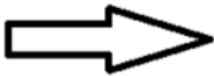
 P1:

6			3	4	1	7
---	--	--	---	---	---	---

Liczba 6 nie znajduje się w skopiowanym segmencie, zatem kopiujemy ją do P1.

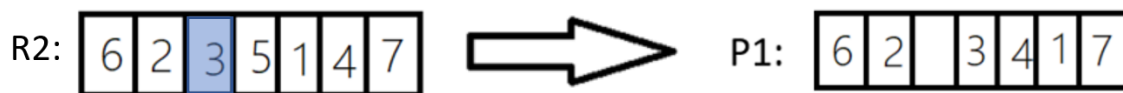
R2:

6	2	3	5	1	4	7
---	---	---	---	---	---	---

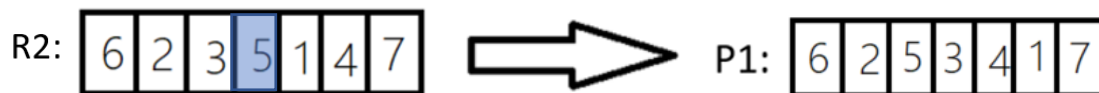
 P1:

6	2		3	4	1	7
---	---	--	---	---	---	---

Liczba 2 nie znajduje się w skopiowanym segmencie, zatem kopiujemy ją do P1.



Liczba 3 znajduje się w skopiowanym segmencie, zatem nie kopiujemy ją do P1.



Wypełniliśmy całą tablicę P1, zatem kończymy tworzenie P1. Analogicznie zostaje stworzony P2.

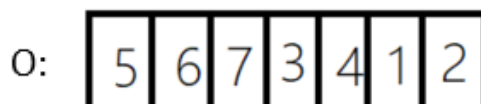
1.4. Mutacja

Po stworzeniu nowej populacji, zwanej również nowym pokoleniem, przeprowadzana jest z pewnym prawdopodobieństwem mutacja jednego z nich. Losowana jest liczba z przedziału 0.001 do 0.999. Jeżeli wybrana liczba jest większa od współczynnika mutacji wprowadzonego przez użytkownika, mutacja jest pomijana.

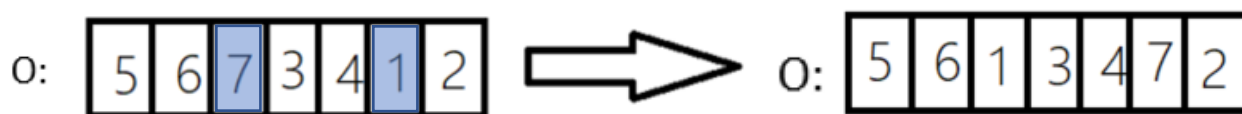
W programie zaimplementowane są dwie metody mutacji.

1.4.1. Mutacja przez zamianę

Z nowej populacji w sposób losowy wybierany jest osobnik, oznaczony O. Przechowywana przez niego ścieżka prezentuje się następująco:



Losowane są dwa indeksy spośród tych, zawierających się w tablicy. Przyjmijmy, że są to liczby 2 i 5. Teraz następuje zamiana elementów o wylosowanych indeksach.



2. Opis programu

W niniejszej sekcji znajduje się opis najważniejszych klas w projekcie.

2.1. Klasa Main.cpp

Główna klasa projektu, w której znajduje się funkcja main(). Odpowiedzialna jest ona za sterowanie programem przez użytkownika. Z jej poziomu wywoływane są metody zdefiniowane w innych klasach. Główną część metody main() stanowi instrukcja typu *switch-case*, obudowana jest ona w pętlę *while* z warunkiem true, dzięki czemu menu główne programu wyświetla się po każdej wywołanej operacji, dopóki użytkownik nie zasygnalizuje chęci wyjścia z programu.

Opisana wyżej pętla *while* prezentuje się następująco:

```
while (true)
{
    Interface::menu_glowne();
    cin >> choice;
    switch (choice)
    {
        case 1: //wczytaj dane z pliku
            cout << "Podaj nazwe pliku" << endl;
            cin >> fileName;
            czyPoprawna = matrix->load_matrix(fileName);
            if (!czyPoprawna)
                cout << "Operacja się nie udała" << endl;
            break;
        case 2: //wyświetl graf
            czyPoprawna = matrix->show_matrix();
            if (!czyPoprawna)
                cout << "operacja się nie udała" << endl;
            break;
        case 3: //wielkosc populacji początkowej
            cout << "Podaj wielkosc populacji początkowej:"<<endl;
            cin >> matrix->wielkosc_populacji;
            break;
        case 4: //współczynnik mutacji
            cout << "Wprowadz współczynnik mutacji:" << endl;
            cin >> matrix->wspolczynn timer_mutacji;
            break;
        case 5: //współczynnik krzyżowania
            cout << "Wprowadz współczynnik krzyzowania:" << endl;
            cin >> matrix->wspolczynn timer_krzyzowania;
            break;
        case 6: //metoda krzyżowania
            break;
        case 7: //metoda mutacji
            break;
        case 8: //uruchom algorytm
            matrix->genetic_algorithm();
            break;
        case 9:
            cout << "Wprowadz kryterium stopu: " << endl;
            cin >> matrix->kryterium_stopu;
            matrix->kryterium_stopu = matrix->kryterium_stopu * 1000000000;
            //w nanosekundach
            break;
        case 10: //wyjdź z programu
            return 0;
            break;
        default:
            cout << "Wprowadzono zły znak" << endl;
    }
}
```

2.2. Klasa AdjacencyMatrix.cpp

Klasa zawierająca struktury danych przechowujące grafy oraz metody odpowiedzialne za działanie algorytmu genetycznego.

2.2.1. load_matrix()

Metoda odpowiedzialna za wczytanie grafu z pliku .atsp do macierzy. Metoda ta wczytuje z pliku informację o rozmiarze grafu, alokuje potrzebną pamięć, a następnie odpowiednim elementom macierzy przypisuje wagi krawędzi opisane w pliku. W przypadku jeżeli w pamięci istnieje już macierz reprezentująca graf, metoda najpierw tą pamięć zwolni.

```

bool AdjacencyMatrix::load_matrix(string fileName)
{
    if (n != 0)
    {
        for (int i = 0; i < n; i++)
            delete A[i];
        delete[] A;
    }
    else
    {
        fstream file;
        file.open(fileName.c_str(), ios::in);
        if (file.good() == false)
            return false;
        string help;
        for (int i = 0; i < 9; i++)
            file >> help;
        if (help == "(Ascheuer)")
        {
            file >> help;
        }
        file >> n;
        for (int i = 0; i < 5; i++)
            file >> help;
        A = new int* [n];
        for (int i = 0; i < n; i++)
        {
            A[i] = new int[n];           // alokacja n komorek w
            kazdej komorce tablicy, efekt macierzy nxn
        }
        for (int i = 0; i < n; i++)
        {
            for (int j = 0; j < n; j++)
            {
                file >> A[i][j];         // wypelnienie kolejnych
                komorek macierzy kolejnymi wartosciami z pliku tekstowego
            }
        }

    }
    return true;
}

```

2.2.2. show_matrix()

Metoda odpowiedzialna za wyświetlenie macierzy sąsiedztwa na ekranie. Metoda ta nie jest konieczna w projekcie. Jednak została ona dodana w celu sprawdzenia poprawności wczytanych grafów.

```

bool AdjacencyMatrix::show_matrix()           // metoda sluzaca do wyswietlania
zawartosci macierzy
{
    if (n == 0)
        return false;                       // jezeli liczba
    przechowujaca ilosc wierzchołkow rowna sie 0, funkcja sie zakonczy
    else
    {
        cout << " ";
        for (int i = 1; i < n + 1; i++)      // gorny wiersz wypisujacy
            wierzchołki grafu                printf("%5d", i);

        cout << endl;
        for (int i = 0; i < n; i++)
        {
            printf("%5d", i + 1);           // wypisanie wierzchołka
            for (int j = 0; j < n; j++)

```

```

        {
            printf("%5d", A[i][j]);           // wypisanie wartosci
krawedzi idacych od wypisanego wczesniej wierzchołka do wierzchołkow w gornym wierszu
        }
        cout << endl;
    }

    }
    return true;
}

```

2.2.3. generate_population()

Metoda odpowiedzialna za wygenerowanie populacji początkowej, wykorzystanej do algorytmu. Na początku metody alokowana jest tablica struktur zawierających osobniki. Każda struktura zawiera tablicę przechowującą ścieżkę oraz długość tej ścieżki, to znaczy sumę odległości pomiędzy miastami w ścieżce. Następnie dla każdego elementu w tablicy w sposób losowy generowana jest ścieżka oraz obliczana jest jej długość. Ilość osobników jest określona przez użytkownika. Nad niepowtarzalnością elementów w ścieżce danego osobnika czuwa tablica *odwiedzony*, która zawiera informację, czy dany wierzchołek jest już dodany do ścieżki.

```

void AdjacencyMatrix::generate_population() // generowanie populacji początkowej
{
    int wierzcholek;
    bool* odwiedzony;
    odwiedzony = new bool[n]; // tablica wygenerowanych wierzchołkow
    int dlugosc_nowego;
    populacja_początkowa = new osobnik[wielkosc_populacji]; // tablica struktur
przechowujących populację początkową
    for (int j = 0; j < wielkosc_populacji; j++) //ilosc wygenerowanych osobników
    {
        dlugosc_nowego = 0;
        populacja_początkowa[j].sciezka = new int[n]; // alokacja sciezki kazdego
osobnika
        for (int i = 0; i < n; i++) // tablica wygenerowanych wierzchołkow
            odwiedzony[i] = false;
        //tworzenie pojedynczego osobnika
        populacja_początkowa[j].sciezka[0] = 0; // 0 jest wierzchołkiem początkowym dla
kazdego osobnika
        odwiedzony[0] = true;
        for (int i = 1; i < n; i++)
        {
            do
            {
                wierzcholek = rand() % (n - 1) + 1; //wierzcholek od 0 do (n-1)
            } while (odwiedzony[wierzcholek] == true); // powtorz losowanie jesli
wierzcholek jest juz uzyty

            odwiedzony[wierzcholek] = true;
            populacja_początkowa[j].sciezka[i] = wierzcholek; //dodanie wierzchołka do
sciezki osobnika

        }
        for (int k = 0; k < (n - 1); k++)
        {
            dlugosc_nowego +=
A[populacja_początkowa[j].sciezka[k]][populacja_początkowa[j].sciezka[k + 1]]; //obliczanie
dlugosci sciezki dla kazdego osobnika
        }
        dlugosc_nowego += A[populacja_początkowa[j].sciezka[n - 1]][0];
        populacja_początkowa[j].dlugosc = dlugosc_nowego;
    }

    delete[] odwiedzony; // zwolnienie pamieci po tablicy odwiedzonych wierzchołkow
}

```

2.2.4. selection()

Metoda odpowiedzialna za selekcję osobników i wybranie tych, które staną się rodzicami dla kolejnego pokolenia. Z populacji wybierane są losowo dwa zestawy po $wspolczynnik_krzyzowania * wielkosc_populacji$ osobników. Następnie z każdego z zestawów wybierany jest osobnik z najkrótszą długością ścieżki. Dwa wybrane osobniki stają się rodzicami. Po wybraniu rodziców w metodzie *selection()* wywoływana jest metoda odpowiedzialna za krzyżowanie, która dla każdej pary rodziców zwraca parę potomków. Cały proces, poczynając od wyboru zestawów powtarzany jest $wielkosc_populacji/2$ razy, tak aby powstało $wielkosc_populacji$ potomków. Z racji na wielkość metody zaprezentowany będzie fragment kodu odpowiadający za pojedyncze wybranie rodzica.

```
for (int j = 0; j < k; j++) //wybor 4 kandydatow do turnieju
{
    do
    {
        rodzic = rand() % (wielkosc_populacji - 1);
    } while (wybrany[rodzic] == true); //kazdy musi byc inny
    wybrany[rodzic] = true;
    turniej[j] = rodzic; //dodanie kandydata do turnieju
}

min_dlugosc = 2147483647;
for (int j = 0; j < k; j++)
{
    if (populacja_początkowa[turniej[j]].dlugosc < min_dlugosc)
    {
        min_dlugosc = populacja_początkowa[turniej[j]].dlugosc;
        //wybranie najlepszego z turnieju jako rodzica1
        rodzic1 = turniej[j];
    }
}
```

Po wybraniu drugiego rodzica w ten sam sposób wywoływana jest metoda *order_crossover()*.

2.2.5. order_crossover()

Metoda odpowiedzialna za krzyżowanie osobników i stworzenie potomków. Na początku metody, w sposób opisany w punkcie 1.3. podejmowana jest decyzja czy nastąpi krzyżowanie. Jeżeli nie, to dwójka rodziców przeznaczonych do krzyżowania jest kopiowana do nowej populacji, tj. następnego pokolenia i metoda kończy działanie.

```
float czy_krzyzowanie = rand() % 101 * 0.01;

if (czy_krzyzowanie > wspolczynnik_krzyzowania)
{
    for (int i = 0; i < n; i++)
        nowa_populacja[j].sciezka[i] = populacja_początkowa[rodzic1].sciezka[i];
    nowa_populacja[j].dlugosc = populacja_początkowa[rodzic1].dlugosc;

    for (int i = 0; i < n; i++)
        nowa_populacja[j + 1].sciezka[i] =
populacja_początkowa[rodzic2].sciezka[i];
    nowa_populacja[j+1].dlugosc = populacja_początkowa[rodzic2].dlugosc;
    return;
}
```

W przeciwnym wypadku następuje losowanie dwóch liczb odpowiedzialnych za wyznaczenie segmentu, który będzie kopiowany do pierwszego potomka.


```

int punkt_podzialu1 = rand() % (n - 2) + 2; //losowanie punktu podzialu
int punkt_podzialu2;
do
{
    punkt_podzialu2 = rand() % (n - 2) + 2;
} while (punkt_podzialu2 == punkt_podzialu1);

if (punkt_podzialu1 < punkt_podzialu2)
{
    pierwszy = punkt_podzialu1;
    drugi = punkt_podzialu2;
}
else
{
    pierwszy = punkt_podzialu2;
    drugi = punkt_podzialu1;
}

```

Po wylosowaniu punktów następuje kopiowanie segmentu ścieżki do pierwszego z potomków. Nowa populacja, podobnie jak stara, przechowywana jest w tablicy struktur przechowujących ścieżkę osobnika oraz jej długość.

```

nowa_populacja[j].sciezka[0] = 0;
for (int i = pierwszy, k = 0; i < drugi; i++,k++)
{
    nowa_populacja[j].sciezka[i] = populacja_pocatkowa[rodzic1].sciezka[i];
//kopiowanie segmentu z rodzica1
    odwiedzony[nowa_populacja[j].sciezka[i]] = true;
}

```

Następnie z drugiego rodzica kopiowane są elementy ścieżki zaczynając od indeksu, który stanowił górną granicę segmentu. Po dojściu do końca ścieżki rodzica elementy są kopiowane od początku ścieżki aż do wypełnienia całej tablicy. Elementy te zapisywane są w tablicy, która będzie później służyła do uzupełniania ścieżki pierwszego potomka.

```

tablica2 = new int[n-1];
//kopiowanie pozostałych elementów rodzica do tablicy od rodzica2
int pozycja = 0;
for (int i = drugi; i < n; i++, pozycja++)
{
    tablica2[pozycja] = populacja_pocatkowa[rodzic2].sciezka[i];
}
for (int i = 1; i < drugi; i++,pozycja++)
{
    tablica2[pozycja] = populacja_pocatkowa[rodzic2].sciezka[i];
}

```

Kolejny fragment metody odpowiedzialny jest za dodawanie elementów uprzednio dodanych do tablicy do ścieżki pierwszego potomka w sposób opisany w punkcie 1.3., czyli elementy nieobecne w skopiowanym wcześniej segmencie zostają pominięte. Po dojściu do końca ścieżki pierwszego potomka, indeks przeskakuje na jej początek i elementy są wypełnianie do momentu wypełnienia całej ścieżki.

```

pozycja = drugi;
int pozycja2 = 0;
for (int i = 0; pozycja < n; i++, pozycja++,pozycja2++)
{
    if (odwiedzony[tablica2[i]] == true)
    {
        pozycja--;
        continue;
    }
    else

```

```

        {
            nowa_populacja[j].sciezka[pozycja] = tablica2[i];
        }
    }

    pozycja = 1;

    for (int i = pozycja2; pozycja < pierwszy; i++)
    {
        if (odwiedzony[tablica2[i]] == true)
        {
            continue;
        }
        else
        {
            nowa_populacja[j].sciezka[pozycja] = tablica2[i];
            pozycja++;
        }
    }
}

```

Po stworzeniu pierwszego potomka, w identyczny sposób tworzony jest drugi potomek z tych samych rodziców.

Po stworzeniu drugiego potomka, obliczana jest długość ścieżki obu potomków.

```

int dlugosc1 = 0;
int dlugosc2 = 0;

for (int i = 0; i < (n-1); i++)
{
    dlugosc1 += A[nowa_populacja[j].sciezka[i]][nowa_populacja[j].sciezka[i + 1]];
    dlugosc2 += A[nowa_populacja[j+1].sciezka[i]][nowa_populacja[j+1].sciezka[i +
1]];
}
//cout << "Obliczanie sumy2" << endl;
dlugosc1 += A[nowa_populacja[j].sciezka[n - 1]][0];
nowa_populacja[j].dlugosc = dlugosc1;
dlugosc2 += A[nowa_populacja[j+1].sciezka[n - 1]][0];
nowa_populacja[j+1].dlugosc = dlugosc2;

```

Na końcu metody zwalniana jest pamięć zaalokowana wcześniej do przechowywania elementów tablic pomocniczych.

2.2.6. mutation()

Metoda odpowiedzialna za mutację losowego osobnika. Na początku metody, w sposób opisany w punkcie 1.4. podejmowana jest decyzja czy nastąpi mutacja. Jeżeli nie, metoda kończy działanie.

W przeciwnym wypadku losowane są indeksy dwóch elementów, które zostaną zamienione w ścieżce.

```

pozycja1 = rand() % (n - 2) + 2;
do
{
    pozycja2 = rand() % (n - 2) + 2;
} while (pozycja2 == pozycja1);

```

Następnie losowany jest osobnik, który zostanie poddany mutacji i dokonywana jest zamiana dwóch wcześniej wybranych elementów w ścieżce.

```

int losowy_osobnik = rand() % (wielkosc_populacji - 1);
int pomoc = nowa_populacja[losowy_osobnik].sciezka[pozycja1];
nowa_populacja[losowy_osobnik].sciezka[pozycja1] =
nowa_populacja[losowy_osobnik].sciezka[pozycja2];

```

```
nowa_populacja[losowy_osobnik].sciezka[pozycja2] = pomoc;
```

Po zamianie na nowo obliczana jest długość ścieżki zmutowanego osobnika.

```
int dlugosc = 0;
for (int i = 0; i < (n - 1); i++)
{
    dlugosc +=
A[nowa_populacja[losowy_osobnik].sciezka[i]][nowa_populacja[losowy_osobnik].sciezka[i + 1]];
}
dlugosc += A[nowa_populacja[losowy_osobnik].sciezka[n -
1]][nowa_populacja[losowy_osobnik].sciezka[0]];
nowa_populacja[losowy_osobnik].dlugosc = dlugosc;
```

2.2.7. change_population()

Metoda odpowiedzialna za podmienienie starego pokolenia na nowe. Polega na kopiowaniu każdego elementu tablicy przechowującej nowe pokolenie, do tablicy przechowującej stare pokolenie.

```
void AdjacencyMatrix::change_population()
{
    for (int i = 0; i < wielkosc_populacji; i++)
    {
        populacja_początkowa[i].dlugosc = nowa_populacja[i].dlugosc;
        for (int j = 0; j < n; j++)
            populacja_początkowa[i].sciezka[j] = nowa_populacja[i].sciezka[j];
    }
}
```

2.2.8. genetic_algorithm()

Metoda odpowiedzialna za przeprowadzenie całego algorytmu genetycznego. W jej środku wywoływane są opisane wcześniej metody. Na początku generowana jest populacja początkowa. Następnie w pętli *while* pilnującej czasu wykonywania algorytmu wywoływane są metody odpowiedzialne za tworzenie nowej populacji, mutację osobnika oraz zamianę aktualnego pokolenia, na najnowsze. Po każdym takim przebiegu szukany jest najsilniejszy osobnik. Po wyjściu z pętli *while* wypisywana jest ścieżka najsilniejszego osobnika oraz jej długość.

```
void AdjacencyMatrix::genetic_algorithm()
{
    nowa_populacja = new osobnik[wielkosc_populacji];
    for (int i = 0; i < wielkosc_populacji; i++)
        nowa_populacja[i].sciezka = new int[n];

    sciezka = new int[n];
    min = 2147483647;
    double czas = 10000000000;
    StartCounter();
    generate_population();
    while (GetCounter() < kryterium_stopu)
    {
        selection(); // i krzyzowanie
        mutation();
        change_population();
        for (int i = 0; i < wielkosc_populacji; i++)
        {
            if (populacja_początkowa[i].dlugosc < min)
            {
                min = populacja_początkowa[i].dlugosc;
                for (int j = 0; j < n; j++)
                {
                    sciezka[j] = populacja_początkowa[i].sciezka[j];
                }
            }
        }
    }
}
```

```

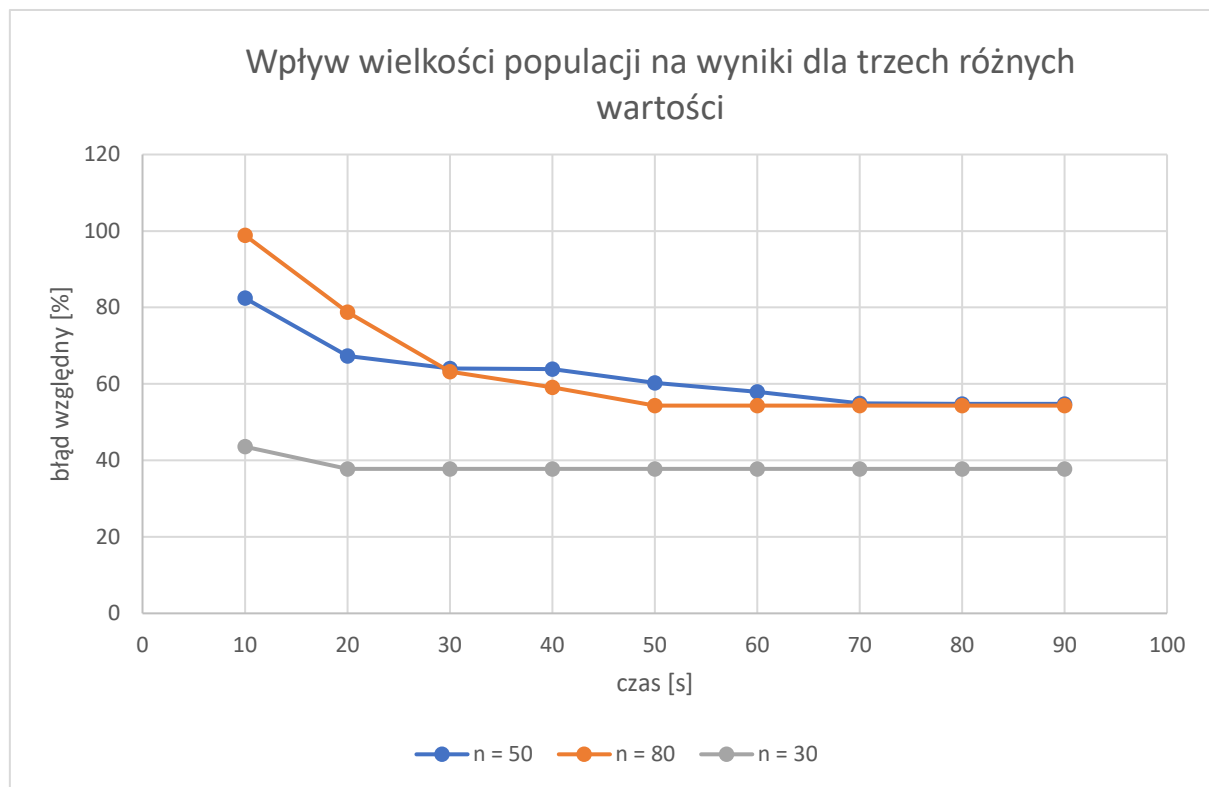
    }
    }
    cout << min << endl;
}
cout << "Zakończono chyba z sukcesem" << endl;
cout << "Najlepsza znaleziona dlugosc: " << min << endl;
cout << "Najkrotsza sciezka:" << endl;
for (int i = 0; i < n; i++)
    cout << sciezka[i]<<" ";
}

```

3. Testy

W niniejszej sekcji znajdują się testy efektywności zaimplementowanego algorytmu genetycznego. Dla każdego pliku zamieszczone są wykresy błędu funkcji czasu. Błąd względny liczony jest ze wzoru $|f_{zn} - f_{opt}|/f_{opt}$, gdzie f_{zn} oznacza wartość obliczoną przez algorytm, natomiast f_{opt} najlepsze znane rozwiązanie.

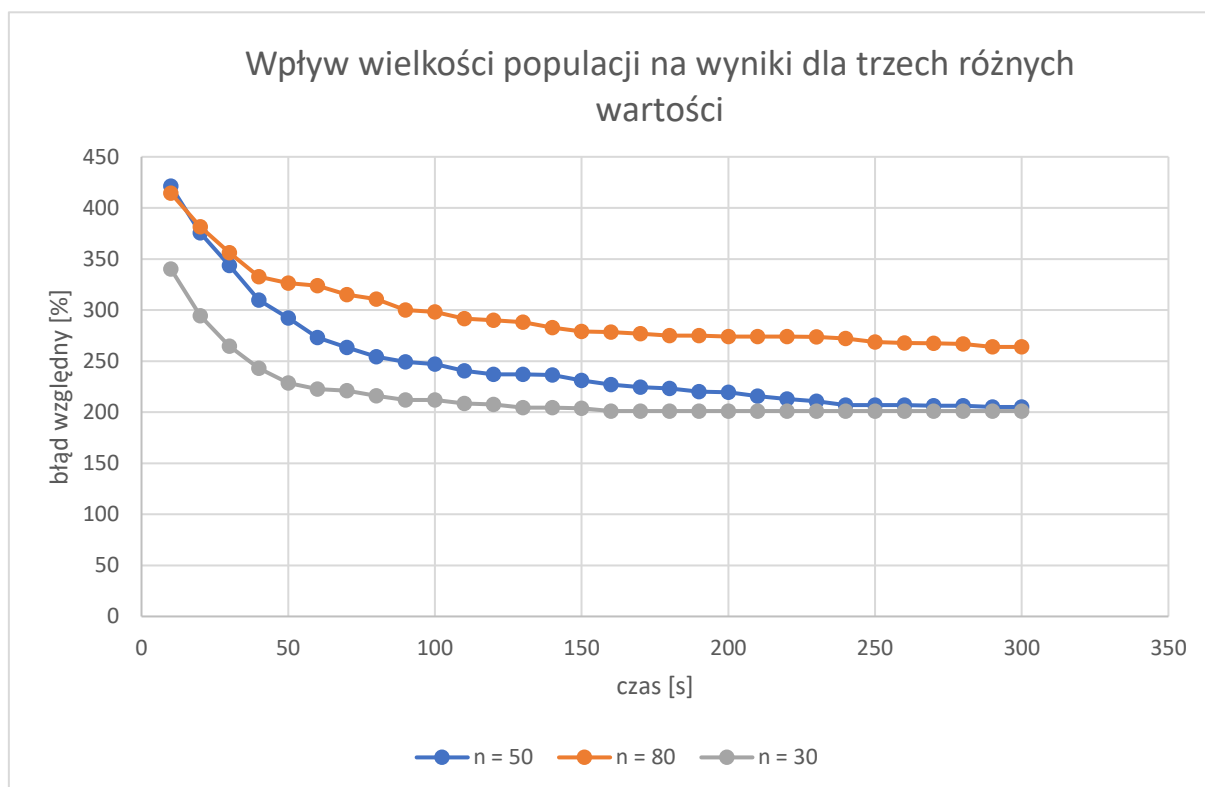
3.1. Testy dla pliku ftv47.atsp



Rysunek 1. Wykres błędu względnego dla pliku ftv47.atsp

Jak widać na zamieszczonym powyżej wykresie, im większy był rozmiar populacji, tym gorsze było pierwsze rozwiązanie. Co za tym idzie w kolejnych ewolucjach mniejsze populacje zachowywały przewagę pod względem znalezienia najsilniejszego osobnika. Najlepszy rezultat został osiągnięty dla najmniejszej populacji.

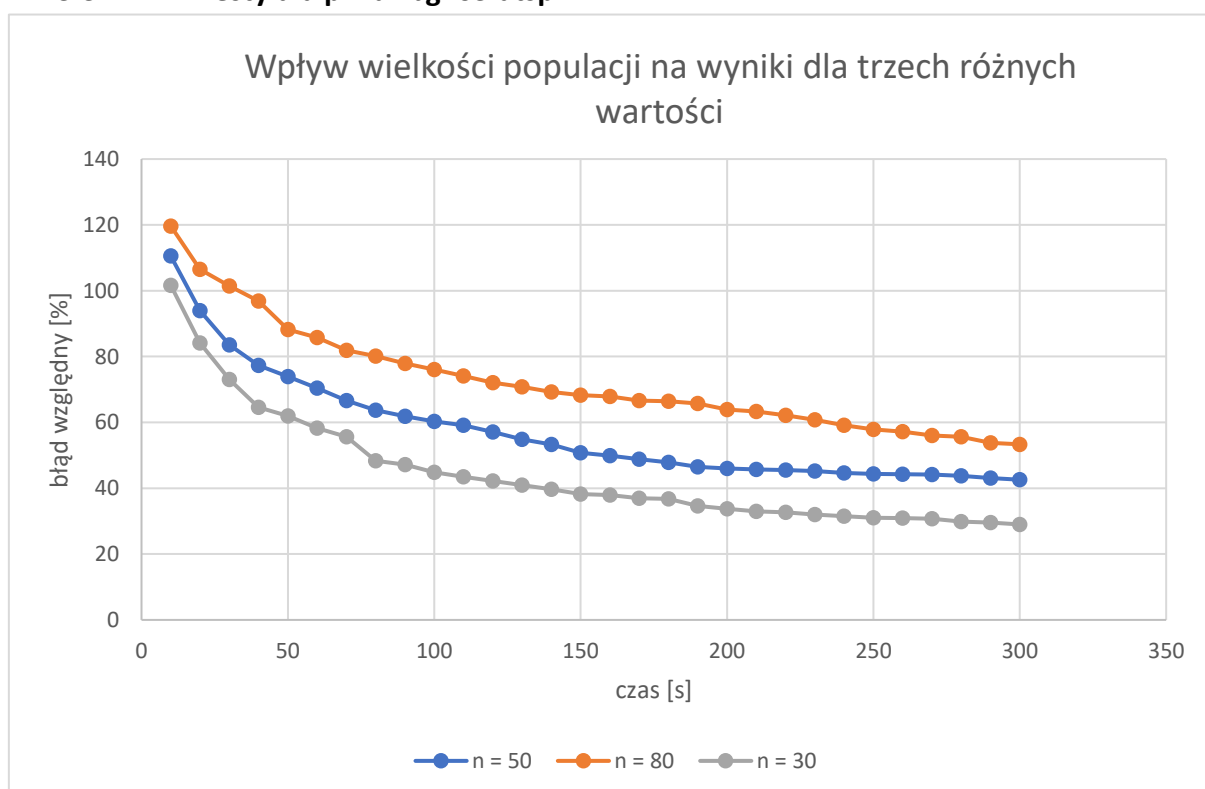
3.2. Testy dla pliku ftv170.atsp



Rysunek 2. Wykres błędu względnego dla pliku ftv170.atsp

Podobnie jak w poprzednim przykładzie, im mniejsza była populacja początkowa, tym lepsze osiągała ona wyniki. Warto tutaj jednak zauważyć, że w pewnym momencie populacja najmniejsza przestała się rozwijać, zaś pozostałe dwie mimo zwolnienia tempa, osiągały coraz to lepsze rezultaty.

3.3. Testy dla pliku rbg403.atsp

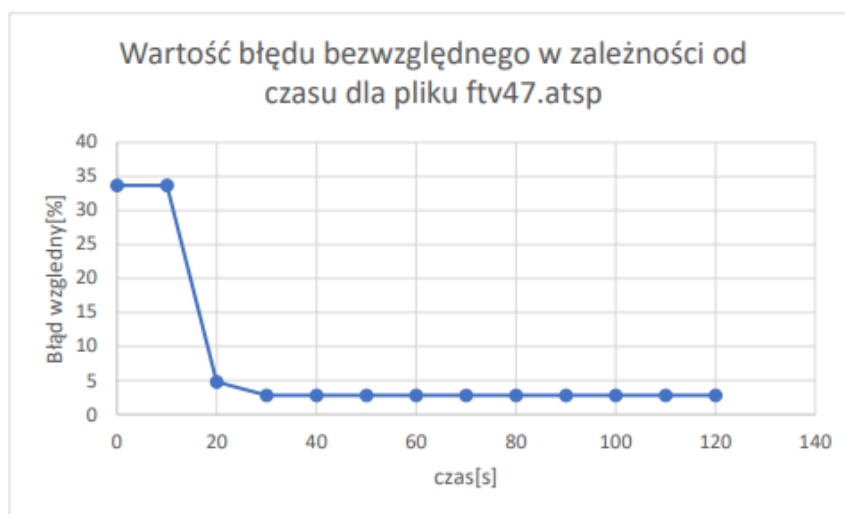


Rysunek 3. Wykres błędu względnego pliku rbg403.atsp

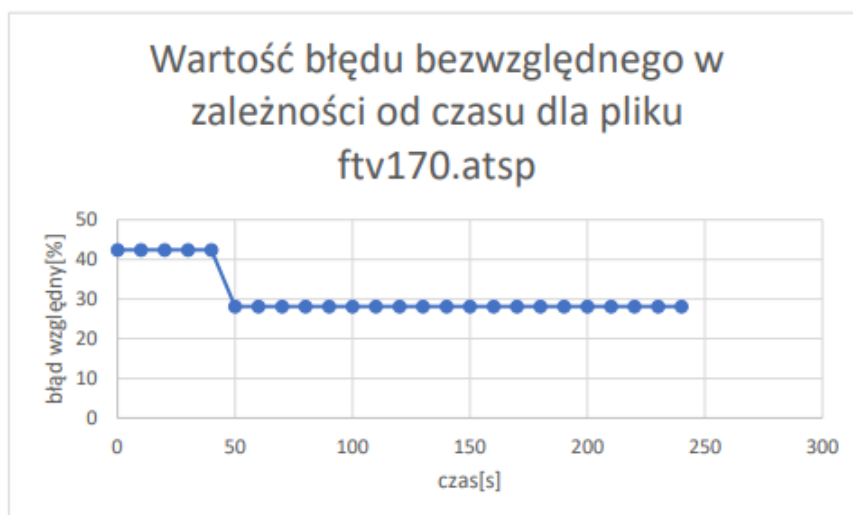
Jak można zauważyć, w tym przypadku ponownie jak w poprzednich, im większa była populacja początkowa, tym gorsze były rozwiązania początkowe. Wszystkie populacje rozwijały się w podobnym tempie.

4. Porównanie z algorytmem symulowanego wyżarzania z poprzedniego zadania projektowego

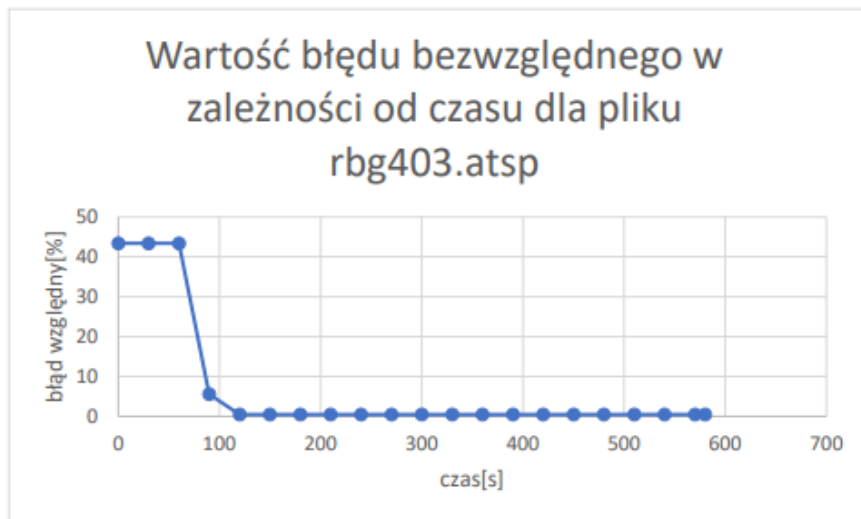
W celu lepszego zobrazowania różnic w wynikach między oboma algorytmami, warto przedstawić wykresy błędów względnego dla algorytmu symulowanego wyżarzania.



Rysunek 4. Wykres błędów względnego dla algorytmu SW i pliku ftv47.atsp



Rysunek 5. Wykres błędów względnego dla algorytmu SW i pliku ftv170.atsp



Rysunek 6. Wykres błędu względnego dla SW i pliku rbg403.atsp

Porównując powyższe wykresy z wykresami zamieszczonymi w poprzednim punkcie łatwo można zauważyć, że algorytm symulowanego wyżarzania dużo szybciej osiągał znacznie lepsze wyniki od algorytmu genetycznego. Poza tym algorytm genetyczny przy niektórych parametrach przestawał się rozwijać.

5. Wnioski

Na podstawie wykresów w punkcie 3. i 4. można stwierdzić, że algorytm genetyczny dla podanych plików działa znacznie gorzej niż algorytm symulowanego wyżarzania. Ponadto da się zauważyć zależność pomiędzy wynikiem początkowym, a rozmiarem populacji początkowej. Im mniejsza populacja, tym lepszy jest wynik początkowy. Dzięki temu mniejsze populacje w kolejnych etapach ewolucji również osiągają korzystniejsze wyniki, natomiast szybciej przestają się rozwijać. Można zatem stwierdzić, że aby zoptymalizować działanie algorytmu genetycznego należy odpowiednio dobrać parametry takie jak: wielkość populacji początkowej, współczynnik krzyżowania oraz współczynnik mutacji, a także ilość osobników biorących udział w turnieju w trakcie selekcji. Za sukces można uznać fakt, że chociaż powoli, to kolejne pokolenia zawierają silniejsze osobniki.