

**Politechnika Wrocławska**  
**Wydział Informatyki i Telekomunikacji**

---

# **Projektowanie efektywnych algorytmów**

Projekt nr 4 - Implementacja i analiza efektywności  
algorytmu genetycznego dla problemu komiwojażera

---

semestr zimowy 2023/2024

Autor:  
Eryk Mika 264451

Prowadzący:  
Dr inż. Marcin Łopuszyński

# Spis treści

1. WSTĘP TEORETYCZNY .....	1
1.1. METODY KRZYŻOWANIA .....	1
1.2. METODY MUTACJI .....	2
1.3. METODA SELEKCJI .....	2
1.4. POPULACJA POCZĄTKOWA .....	3
2. OPIS IMPLEMENTACJI ALGORYTMÓW .....	3
2.1. KLASA <i>GRAPH</i> .....	3
2.2. KLASA <i>GRAPH</i> – METODA <i>GENERATEINITIALSOLUTION()</i> .....	3
2.3. KLASA <i>GRAPH</i> – METODA <i>SOLVEGA()</i> .....	4
2.4. KLASA <i>ROUTE</i> .....	6
3. SPOSÓB PRZEPROWADZENIA BADANIA .....	9

# 1. Wstęp teoretyczny<sup>1</sup>

Zgodnie z informacjami przedstawionymi w poprzednich sprawozdaniach, problem komiwożera (*TSP*) jest problemem trudnym pod względem obliczeniowym. W tym opracowaniu zostanie omówione rozwiązanie tego problemu z wykorzystaniem algorytmu genetycznego. Jest to rodzaj algorytmu ewolucyjnego - jest wzorowany na biologicznej ewolucji oraz stosowany jest do optymalizacji oraz planowania.

Algorytm genetyczny jest heurystyką<sup>2</sup>. Symuluje on proces naturalnej selekcji poprzez ocenę adaptacji poszczególnych jednostek, eliminację słabszych osobników oraz krzyżowanie tych o największym przystosowaniu – w ten sposób powstają nowe osobniki w populacji. Każdy osobnik reprezentuje określony sposób rozwiązania problemu, który wyznacza dany *chromosom*. Osobniki oceniane są według pewnego kryterium – *funkcji oceny* – która, w przypadku problemu *TSP*, może być rozumiana jako funkcja przyporządkowująca koszt do danej trasy. Istotnym elementem algorytmu jest także mutacja, która polega na zmianie pewnych elementów rozwiązania według pewnego wzorca z określonym prawdopodobieństwem. Efektem tego procesu jest populacja jednostek, z których wybierane są te o najwyższym stopniu przystosowania. Zbiór informacji całej populacji określa się jako *genotyp*.

## 1.1. Metody krzyżowania

Krzyżowanie, realizowane poprzez *operator krzyżowania*, polega na kombinacji cech różnych osobników z populacji, co prowadzi do powstania nowych rozwiązań. Krzyżowanie zachodzi z pewnym ustalonym prawdopodobieństwem.

W zaimplementowanym i omawianym algorytmie zastosowano operator krzyżowania **PMX** (ang. *partially matched crossover*) – krzyżowanie z częściowym odwzorowaniem. W algorytmie realizującym ten operator wybierane są dwa punkty podziału, które wyznaczają tzw. sekcję dopasowania (ang. *matching section*). W ten sposób definiowane są punkty, które wyznaczają sposób transpozycji (zmianę miejsc) elementów danego rozwiązania<sup>3</sup> – szczegółowy opis algorytmu zawarty jest w opisie implementacji.

---

<sup>1</sup> [https://sound.eti.pg.gda.pl/student/isd/isd03-algorytmy\\_genetyczne.pdf](https://sound.eti.pg.gda.pl/student/isd/isd03-algorytmy_genetyczne.pdf)

<sup>2</sup> [https://pl.wikipedia.org/wiki/Algorytm\\_genetyczny](https://pl.wikipedia.org/wiki/Algorytm_genetyczny)

<sup>3</sup> <https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf>

## 1.2. Metody mutacji<sup>4</sup>

Mutacja polega na wprowadzaniu losowych zmian do genotypu populacji. Ma to na celu zwiększenie różnorodności generowanych rozwiązań. Mutacja zachodzi z pewnym ustalonym prawdopodobieństwem, które z reguły jest niewielkie ( $\leq 1\%$ ), co ma na celu zachowanie równowagi pomiędzy przeszukiwaniem lokalnym (wokół pewnej grupy rozwiązań) oraz zwiększaniem przeszukiwanej przestrzeni rozwiązań<sup>5</sup>.

W prezentowanym projekcie zastosowano dwa operatory mutacji: *inverse* oraz *scramble*.

Operator *inverse* polega na odwróceniu kolejności elementów rozwiązania pomiędzy dwoma przyjętymi punktami w chromosomie (Rysunek 1.1).

Rysunek 1.1 Przykład zastosowania operatora *inverse*



Źródło: opracowanie własne

Operator *scramble* polega na losowym przestawianiu wybranych elementów z genotypu (Rysunek 1.2).

Rysunek 1.2 Przykład zastosowania operatora *scramble*



Źródło: opracowanie własne

## 1.3. Metoda selekcji

Selekcja w algorytmie genetycznym polega na wybieraniu osobników z populacji, które przejdą do następnego pokolenia (iteracji algorytmu). Możliwa jest realizacja selekcji na wiele sposobów. Do najważniejszych należą między innymi tzw. metoda ruletki oraz **metoda rankingowa**<sup>6</sup>, która została zaimplementowana i użyta w przedstawionym projekcie.

Metoda rankingowa w zaimplementowanej postaci polega na posortowaniu osobników w populacji rosnąco według przyjętej funkcji oceny – kosztu danej trasy. Następnie, zakładając, że do następnego pokolenia przechodzi  $n$  najlepszych osobników, pozostałe (gorsze) są usuwane (przykład przedstawia Rysunek 1.3).

Rysunek 1.3 Przykład zastosowania metody rankingowej. Populacja jest przedstawiona w postaci tablicy kosztów osobników.

[55, 62, 74, 80, 91, 100, 120, 182] → [55, 62, 74, 80, 91] $n = 5$
--

Źródło: opracowanie własne

<sup>4</sup> <https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf>

<sup>5</sup> <https://www.baeldung.com/cs/genetic-algorithms-crossover-probability-and-mutation-probability>

<sup>6</sup> [https://en.wikipedia.org/wiki/Selection\\_\(genetic\\_algorithm\)](https://en.wikipedia.org/wiki/Selection_(genetic_algorithm))

## 1.4. Populacja początkowa

Populacja początkowa jest grupą osobników – rozwiązań, od których zaczyna swoje działanie algorytm. Wielkość tej populacji jest różna i zazwyczaj zależy ona od specyfiki rozwiązywanego problemu<sup>7</sup>. Często stosuje się wygenerowanie całości populacji w sposób losowy, jednakże spotyka się także podejście z wykorzystaniem „ziarna”, które stanowią osobniki, o których wstępnie wiadomo, że mogą być obiecujące – są wygenerowane, na przykład, za pomocą metody zachłannej<sup>8</sup>. „Ziarno” to stanowi pewną część osobników populacji początkowej oprócz osobników wygenerowanych losowo. Ten sposób został wykorzystany w tym projekcie.

## 2. Opis implementacji algorytmów

W celu analizy efektywności omawianych algorytmów został napisany program w języku C++ z wykorzystaniem obiektowego paradygmatu programowania. Najistotniejszymi komponentami aplikacji są klasy *Graph* oraz *Route*, których pola (struktury danych) oraz metody są odpowiedzialne za realizację algorytmu. Wiele istotnych kwestii związanych z implementacją zostało wyjaśnionych w komentarzach w plikach źródłowych.

### 2.1. Klasa *Graph*

Klasa *Graph* jest główną klasą programu, która jest odpowiedzialna za przechowywanie struktury i metod grafu, na którym wykonywane są badane algorytmy. Pola prywatne klasy – dwuwymiarowa tablica *std::vector matrix* oraz *size* są użyte do przechowywania długości krawędzi w postaci macierzy kwadratowej – kosztów - *matrix* stopnia *size*. Oba pola przechowują liczby stałoprzecinkowe typu *int*. W macierzy komórka o współrzędnych *i, j* zawiera odległość pomiędzy wierzchołkami *i i j*.

Zaimplementowano konstruktor wczytujący instancję z pliku tekstowego, przeładowany operator przypisania oraz metodę wypisującą graf (macierz) na ekran. Została zaimplementowana metoda *calculateRouteCost()*, która służy do obliczania kosztu danej trasy komiwojażera w grafie – poprzez iterowanie po krawędziach w ścieżce i dodanie ich długości do kosztu, który jest przypisywany do odpowiedniego pola w obiekcie klasy *Route*.

### 2.2. Klasa *Graph* – metoda *generateInitialSolution()*

Metoda ta użyta do wyznaczenia rozwiązania początkowego w sposób **zachłanny**. Rozwiązanie to wykorzystane jest do tworzenia „ziarna” populacji początkowej.

---

<sup>7</sup> [https://en.wikipedia.org/wiki/Genetic\\_algorithm](https://en.wikipedia.org/wiki/Genetic_algorithm)

<sup>8</sup> <https://medium.datadriveninvestor.com/population-initialization-in-genetic-algorithms-ddb037da6773>

#### Algorytm:

1. Utworzenie listy *visited*, zainicjowanej odwiedzeniem korzenia (wierzchołka 0).
2. Rozpoczyna się pętla, która wykonuje się *size-1* razy, ponieważ trasa musi odwiedzić wszystkie wierzchołki oprócz korzenia.
3. Dla aktualnego wierzchołka *curlIndex* na trasie, znajdowany jest najbliższy nieodwiedzony wierzchołek *dst* z najmniejszą wagą krawędzi. Wartość minimalnej wagi przechowywana jest w zmiennej *dstMin*.
4. Znaleziony wierzchołek *dst* jest dodany do listy *visited* i do wynikowej trasy *res* na odpowiedniej pozycji.
5. Po zakończeniu pętli, waga krawędzi powrotnej do korzenia, czyli od ostatnio odwiedzanego wierzchołka do korzenia (wierzchołek 0), jest dodawana do kosztu. Wygenerowana trasa jest zwracana z metody.

### 2.3. Klasa *Graph* – metoda *solveGA()*

Metoda ta użyta jest do rozwiązywania problemu komiwojażera za pomocą algorytmu genetycznego.

W metodzie tworzony jest wektor *population* przechowujący populację tras. Tworzona jest populacja początkowa. 10% osobników w tej populacji stanowią rozwiązanie uzyskane metodą zachłanną oraz osobniki pochodzące z tego rozwiązania poprzez wywoływania operatora *swap()* (zamiana elementów miejscami). Uzyskana populacja jest sortowana (Rysunek 2.1).

Rysunek 2.1 Algorytm tworzenia populacji początkowej

```
/*
    10% osobników w początkowej populacji wywodzi się z rozwiązania zachłannego
    razem z bazowym rozwiązaniem wygenerowanym zachłannie
*/
while(population.size() < (unsigned)(0.1 * initialPopulation))
{
    /* Wygenerowanie osobników pochodzących z rozwiązania zachłannego (greedy) - losowe
    przestawianie elementów trasy */
    Route r = greedy;
    r.procedureSwap(rand() % routeElements, rand() % routeElements);
    r.procedureSwap(rand() % routeElements, rand() % routeElements);
    calculateRouteCost(r);
    population.emplace_back(r);
}

// Reszta osobników jest wygenerowana losowo
while(population.size() < initialPopulation)
{
    Route r(routeElements);
    r.randomize();
    calculateRouteCost(r);
    population.emplace_back(r);
}

// Sortowanie populacji wg kosztów tras - podejście rankingowe
std::sort(population.begin(), population.end());
```

Źródło: opracowanie własne

Następnie, po inicjalizacji zmiennych związanych z obsługą pomiaru czasu, rozpoczyna się główna pętla algorytmu (Rysunek 2.2).

*Rysunek 2.2 Główna pętla algorytmu genetycznego w postaci pseudokodu*

```
// Główna pętla algorytmu
while(warunek stopu)
{
    // Rozmiar populacji w obecnej iteracji
    unsigned populationSize = population.size();

    for(unsigned i=0; i<populationSize; i++)
    {
        // Mutacja
        if( (double)rand() / (double)RAND_MAX < mutationFactor )
        {
            // Mutacja typu scramble
            if( mutationChoice )
            {
                // Wybierana jest losowa liczba elementów trasy do poprzestawiania
                population[i].mutateScramble( rand() % routeElements );
            }
            // Mutacja typu inverse
            else
            {
                // Odwracanie trasy pomiędzy indeksami idx1, idx2
                int idx1 = 0, idx2 = 0;

                // Uniknięcie braku efektu mutacji
                while(idx1==idx2)
                {
                    idx1 = rand() % routeElements;
                    idx2 = rand() % routeElements;
                }
                population[i].mutateInverse( idx1, idx2 );
            }
            calculateRouteCost(population[i]);
        }

        // Krzyżowanie
        if( (double)rand() / (double)RAND_MAX < crossoverFactor )
        {
            // Wybranie drugiego osobnika - rozwiązania - do krzyżowania
            unsigned secIndex = rand() % populationSize;
            if(secIndex == i) secIndex = ( secIndex + 1 ) % populationSize;
            Route offspring = population[i].crossoverPMX( population[secIndex] );
            calculateRouteCost(offspring);
            population.emplace_back(offspring);
        }
    }

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

    // Jeżeli znaleziono nowe najlepsze rozwiązanie
    if( population[0].getCost() < bestSolution )
    {
        bestSolution = population[0].getCost();
        pomiarczasu;
    }

    /*
    Selekcja rodziców do następnej iteracji
    Wybierane jest 'eliteSize' najbardziej obiecujących (najlepszych) rozwiązań
    */
    if( population.size() > initialPopulation )
    {
        population.erase( population.begin() + eliteSize, population.end() );
    }

    // Sprawdzenie warunku stopu
    if( czas przekroczony )
    {
        break;
    }
}
}
```

*Źródło: opracowanie własne*

W każdej iteracji algorytmu zachodzą operacje mutacji oraz krzyżowania – z ustalonym prawdopodobieństwem odpowiednio *mutationFactor* oraz *crossoverFactor*. W przypadku mutacji możliwe jest wybranie jednego z dwóch operatorów. W przypadku operatora *scramble* losowana jest liczba elementów do przestawiania, natomiast w przypadku operatora *inverse* losowane są dwa indeksy, które wyznaczają odwracany fragment rozwiązania. Po zakończeniu tych operacji populacja jest sortowana i, jeżeli znaleziono nowe najlepsze rozwiązanie, jest ono zapisywane (*bestSolution*). Jednocześnie dokonywany jest pomiar czasu jego znalezienia. Następuje selekcja osobników zgodnie z podejściem rankingowym – zostawiane jest *eliteSize* najlepszych osobników w populacji. Parametr *eliteSize* określony jest jako 50% liczebności populacji początkowej. Na końcu sprawdzany jest warunek stopu jako przekroczony czas – pomiar ten jest wykonywany co 10. iterację pętli w celu zapobieżenia nadmiernego wpływu pomiaru czasu na czas wykonywania właściwego algorytmu. Efekty działania algorytmu są zwracane z metody jako para (*czas znalezienia najlepszego rozwiązania, koszt najlepszej trasy*).

## 2.4. Klasa *Route*

Klasa ta jest użyta do reprezentowania ścieżki – trasy komiwojażera bez pierwszego i ostatniego przystanku na trasie, który jest przyjęty jako 0. Oznacza to, że przykładowo ciąg wierzchołków 0-1-2-3-0 jest w tej klasie reprezentowany jako ciąg 1-2-3. Ciąg wierzchołków jest przechowywany w postaci tablicy `std::vector<int> route`.

Zostały zaimplementowane następujące komponenty klasy:

- Konstruktor *Route(int n)* - tworzy obiekt trasy o rozmiarze *n*. Inicjalizuje wektor *route* o zadanej wielkości,
- Metoda *randomize()* generuje losową permutację trasy, reprezentującą trasę komiwojażera bez pierwszego i ostatniego przystanku na trasie (0),
- Metoda *toString()* zwraca tekstową reprezentację trasy, gdzie kolejne liczby są oddzielone spacją.
- Operator przypisania *operator=* przypisuje zawartość jednej trasy do drugiej.
- Operator dostępu do elementu *operator[]* umożliwia odczyt i modyfikację elementów trasy.
- Operator porównania *operator==* porównuje dwie trasy i zwraca *true*, jeśli są identyczne.
- Metoda *procedureSwap()* wykonuje operację swap (zamiana miejscami) dla dwóch wierzchołków na trasie.
- Metoda *procedureInverse()* wykonuje operację odwracania kolejności elementów trasy między dwoma wskazanymi indeksami.
- Metoda *getSize()* zwraca rozmiar trasy.



- Pomocnicza metoda *swap()* zamienia miejscami dwa elementy trasy na podstawie ich indeksów.

W klasie tej zostały zaimplementowane operatory mutacji oraz krzyżowania.

W przypadku operatorów mutacji, operator *inverse* odwraca kolejność elementów w danym fragmencie rozwiązania poprzez iteracyjne zamienianie elementów z pozycji *i* i *j*. Dla operatora *scramble* losowane są indeksy elementów które będą przestawiane (*chosenIndices*) – dodawane są te indeksy, które nie zostały jeszcze wybrane – nieoznaczone w trasie *markedAsVisited* jako 0. Następnie wybrane wcześniej elementy są losowo przestawiane za pomocą metody pomocniczej *swap()*.

Rysunek 2.3 Implementacja operatorów mutacji w klasie *Route*

```
// Mutacja typu inverse - iteracyjne odwracanie kolejności elementów
void Route::mutateInverse(unsigned i, unsigned j)
{
    if(i > j)
    {
        int temp = i;
        i = j;
        j = temp;
    }

    while(i < j) swap(i++, j--);
}

// Mutacja typu scramble - losowe przestawianie k elementów trasy
void Route::mutateScramble(unsigned k)
{
    if(k > route.size()) return;
    // Kopia obecnej trasy służąca do oznaczania wybranych elementów/indeksów - jako 0
    Route markedAsVisited = *this;
    // Wybrane indeksy do przestawiania
    std::vector<int> chosenIndices = {};

    // Wybierane są k elementy trasy, które będą losowo przestawiane
    while(chosenIndices.size() < k)
    {
        int chosenIndex = rand() % ( route.size() );
        if( markedAsVisited[chosenIndex] != 0 )
        {
            chosenIndices.push_back(chosenIndex);
            markedAsVisited[chosenIndex] = 0;
        }
    }

    // Wcześniej wybrane elementy trasy są losowo przestawiane - wywołanie własnej metody pomocniczej
    for(unsigned i=0; i<chosenIndices.size(); i++)
        swap( chosenIndices[i], chosenIndices[ rand() % chosenIndices.size() ] );
}
```

Źródło: opracowanie własne

W klasie zaimplementowano operator krzyżowania *PMX*. Algorytm zaczyna się od losowego wyboru dwóch indeksów (*a* i *b*), które określają segment poddawany krzyżowaniu. Zapewnione jest, że *a* jest mniejsze niż *b*. Następnie potomek jest tworzony, inicjalizując go trasą drugiego rodzica (*sec*). W kolejnym kroku następuje kopiowanie wybranego segmentu z pierwszego rodzica (*this*) do potomka. Dla każdego elementu w tym segmencie, kopiowany jest do potomka. Po skopiowaniu segmentu identyfikowane są pary elementów, które nie zostały skopiowane w tym segmencie z drugiego rodzica. Tworzona jest lista par (*i, j*), gdzie *i* to element z drugiego rodzica, a *j* to odpowiadający mu element z pierwszego rodzica. Następnie następuje iteracja przez te pary i elementy potomka są umieszczane w miejscach określonych przez te pary. W przypadku konfliktu, gdy element, który ma być umieszczony, już istnieje w potomku, dokonywane są odpowiednie zamiany. Na koniec zwracana jest trasa potomka, która zawiera połączenie cech obu rodziców z uwzględnieniem krzyżowania *PMX*.

Rysunek 2.4 Algorytm operatora *PMX*

```
// Zamieniamy jeżeli taka konieczność, aby a <= b
if(a > b) {
    unsigned temp = a;
    a = b;
    b = temp;
}
// Wynikowy potomek
Route offspring = sec;
// Skopiowanie wybranego segmentu z pierwszego rodzica (*this)
for(unsigned i=a; i<=b; i++)
{
    offspring[i] = this->route[i];
}

std::vector<std::pair<int, int>> pairs = {};

// Wyznaczenie par (i, j); elementy, które nie zostały skopiowane w analogicznym segmencie z drugiego rodzica
for(unsigned i=a; i<=b; i++)
{
    int secRouteElement = sec.route[i];
    bool isFound = false;
    for(unsigned k=a; k<=b; k++) {
        if(secRouteElement == this->route[k]) {
            isFound = true;
            break;
        }
    }
    if(isFound) continue;
    pairs.push_back(std::pair<int, int>(secRouteElement, this->route[i]));
}

// Umieszczanie elementów z par (i, j)
for(unsigned p=0; p<pairs.size(); p++)
{
    int i = pairs[p].first;
    int j = pairs[p].second;
    int destIndex = -1;

    unsigned it = 0;
    while( it < routeSize ) {
        // Czy wewnątrz skopiowanego fragmentu
        bool withinCopiedSegment = ( it >= a && it <= b );
        if( sec.route[it] == j && withinCopiedSegment )
        {
            j = this->route[it];
            it = 0;
            continue;
        } else if ( sec.route[it] == i ){
            destIndex = it;
            break;
        }
        it++;
    }
    // Umieszczenie i na pozycji zajmowanej przez j
    offspring[destIndex] = i;
}
return offspring;
```

Źródło: opracowanie własne

### 3. Sposób przeprowadzenia badania