**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

Prowadzący:  
Dr inż. Marcin Łopuszyński

Autor:  
Eryk Mika 264451

Spis treści

[1. Wstęp teoretyczny 1](#_Toc156123012)

[1.1. Metody krzyżowania 1](#_Toc156123013)

[1.2. Metody mutacji 2](#_Toc156123014)

[1.3. Metoda selekcji 2](#_Toc156123015)

[1.4. Populacja początkowa 3](#_Toc156123016)

[2. Opis implementacji algorytmów 3](#_Toc156123017)

[2.1. Klasa *Graph* 3](#_Toc156123018)

[2.2. Klasa *Graph* – metoda *generateInitialSolution()* 3](#_Toc156123019)

[2.3. Klasa *Graph* – metoda *solveGA()* 4](#_Toc156123020)

[2.4. Klasa *Route* 6](#_Toc156123021)

[3. Sposób przeprowadzenia badania 9](#_Toc156123022)

[4. Wyniki przeprowadzonego badania 9](#_Toc156123023)

[4.1. Wszystkie wyniki przedstawione tabelarycznie 9](#_Toc156123024)

[4.2. Wyniki przedstawione za pomocą wykresów 15](#_Toc156123025)

# Wstęp teoretyczny[[1]](#footnote-1)

Zgodnie z informacjami przedstawionymi w poprzednich sprawozdaniach, problem komiwojażera (*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[[2]](#footnote-2). 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*.

## 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[[3]](#footnote-3) – szczegółowy opis algorytmu zawarty jest w opisie implementacji.

## Metody mutacji[[4]](#footnote-4)

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 (≤1%), co ma na celu zachowanie równowagi pomiędzy przeszukiwaniem lokalnym (wokół pewnej grupy rozwiązań) oraz zwiększaniem przeszukiwanej przestrzeni rozwiązań[[5]](#footnote-5).

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).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** |  |  |  |  | **1** | **2** | **5** | **4** | **3** |

Rysunek . Przykład zastosowania operatora inverse

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

Rysunek . Przykład zastosowania operatora scramble

Źródło: opracowanie własne

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** |  |  |  |  | **1** | **5** | **3** | **4** | **2** |

Źródło: opracowanie własne

## 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[[6]](#footnote-6)**, 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 . Przykład zastosowania metody rankingowej. Populacja jest przedstawiona w postaci tablicy kosztów osobników.

Źródło: opracowanie własne

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

## 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[[7]](#footnote-7). 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[[8]](#footnote-8). „Ziarno” to stanowi pewną część osobników populacji początkowej oprócz osobników wygenerowanych losowo. Ten sposób został wykorzystany w tym projekcie.

# 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.

## 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*.

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

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

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 *curIndex* 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 odwiedzonego wierzchołka do korzenia (wierzchołek 0), jest dodawana do kosztu. Wygenerowana trasa jest zwracana z metody.

## 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).

Źródło: opracowanie własne

Rysunek . Algorytm tworzenia populacji początkowej



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

Źródło: opracowanie własne

Rysunek . Główna pętla algorytmu genetycznego w postaci pseudokodu

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 sortowania 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 najlepszej trasy, koszt najlepszej trasy)*.

## 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()*.

Źródło: opracowanie własne

Rysunek . Implementacja operatorów mutacji w klasie Route

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 . Algorytm operatora PMX

Źródło: opracowanie własne

# Sposób przeprowadzenia badania

W celu realizacji badania – eksperymentu zostały wykorzystane omówione wcześniej klasy. Podobnie jak w przypadku realizacji wcześniejszych projektów, w celu oceny efektywności badanych algorytmów został wykorzystany pomiar czasu przy wykorzystaniu funkcjonalności biblioteki *<chrono>* i zawartej w niej klasy *steady\_clock*, która reprezentuje zegar monotoniczny, dla którego gwarantowane jest, że różnica czasu (przykładowo przekonwertowanego do milisekund – tak jak w programie) będzie większa od zera dla dwóch momentów czasu badania, z których drugi występuje później niż pierwszy. Po wykonaniu metody realizującej algorytm genetyczny wypisywany jest czas uzyskania wyniku w milisekundach oraz koszt ścieżki.

Do badań zostały wykorzystane problemy wskazane w wytycznych dot. projektu. Są one reprezentowane przez pliki:

* *ftv47.atsp* (1776),
* *ftv170.atsp* (2755) ,
* *rgb403.atsp* (2465).

W nawiasach zostały podane najlepsze znane rozwiązania – koszty tras – dla tych problemów[[9]](#footnote-9).

Przeanalizowano rezultaty działania algorytmu dla dwóch opisanych wcześniej operatorów mutacji oraz jednego operatora przypisania. Badania zrealizowano w pierwszej kolejności dla współczynnika mutacji 0,01 oraz współczynnika krzyżowania 0,8 oraz następujących wielkości populacji **103**, **104** oraz **105**.

Następnie ustalono, że najlepsze rezultaty uzyskiwane są dla największej liczebności populacji (105), w związku z czym przeprowadzono kolejne badania dla tej wartości – przeanalizowano wpływ współczynnika mutacji na wyniki (wartości **0,02**, **0,5**, **0,1**) dla obu operatorów mutacji.

Ustalono warunek stopu – limit czasu wykonywania algorytmu na: 2 minuty dla problemu *ftv47.atsp*, 4 minuty dla problemu *ftv170.atsp* oraz 6 minut dla problemu *rbg403.atsp* – identyczny jak w poprzednim projekcie.

# Wyniki przeprowadzonego badania

Wyniki zostały przedstawione za pomocą tabel oraz wykresów.

## Wszystkie wyniki przedstawione tabelarycznie

Wyniki uruchomień algorytmu dla poszczególnych problemów z ustalonymi parametrami zostały zestawione w tabelach.

Obraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, numer, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, numer, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, Czcionka

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, paragon, numer

Opis wygenerowany automatycznie

## Wyniki przedstawione za pomocą wykresów

1. https://sound.eti.pg.gda.pl/student/isd/isd03-algorytmy\_genetyczne.pdf [↑](#footnote-ref-1)
2. https://pl.wikipedia.org/wiki/Algorytm\_genetyczny [↑](#footnote-ref-2)
3. https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf [↑](#footnote-ref-3)
4. https://www.aragorn.wi.pb.edu.pl/~wkwedlo/EA5.pdf [↑](#footnote-ref-4)
5. https://www.baeldung.com/cs/genetic-algorithms-crossover-probability-and-mutation-probability [↑](#footnote-ref-5)
6. https://en.wikipedia.org/wiki/Selection\_(genetic\_algorithm) [↑](#footnote-ref-6)
7. https://en.wikipedia.org/wiki/Genetic\_algorithm [↑](#footnote-ref-7)
8. https://medium.datadriveninvestor.com/population-initialization-in-genetic-algorithms-ddb037da6773 [↑](#footnote-ref-8)
9. <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ATSP.html> [↑](#footnote-ref-9)