# Projekt z
# Projektowania Efektywnych Algorytmów

Jakub Grelowski - 262754
grupa K00-58e
środa - 11:15

18 grudnia 2022

prowadzący: dr inż. Jarosław Mierzwa

***Zadanie projektowe nr 3***

***Rozwiązanie problemu komiwojażera za pomocą algorytmu genetycznego***

# Spis treści

# 1 Wstęp

## 1.1 Cel projektu

Celem projektu było napisanie programu umożliwiającego analize efektywności algorytmów genetycznych rozwiązujących asymetryczny problem komiwojażera.

## 1.2 Sposób przechowywania grafów

- Macierz incydencji

## 1.3 Testowany algorytm

- Algorytm genetyczny

## 1.4 Rodzaje mutacji

- Transpozycja
- Odwracanie

## 1.5 Metody krzyżowania

- OX - Order crossover

## 1.6 Implementacja

Do wykonania projektu wykorzystany został język C# oraz środowisko programistyczne .NET Framework w wersji 4.8.

# 2 Problem komiwojażera

Problem komiwojażera to problem obliczeniowy polegający na znalezieniu w grafie cyklu Hamiltona o najmniejszej wadze. Cykl Hamiltona to taki cykl, w którym każdy wierzchołek pojawia się dokładnie raz.

## 2.1 Asymetryczny problem komiwojażera

Kiedy przeszukiwany graf jest skierowany - czyli waga ścieżki z punktu $A$ do punktu $B$ jest taka sama jak waga ścieżki z punktu $B$ do punktu $A$ - mówimy wtedy o symetrycznym problemie komiwojażera. Tematem projektu jest jednak asymetryczny problem komiwojażera, czyli sytuacja przeciwna - waga ścieżki z punktu $A$ do punktu $B$ nie jest równa wadze ścieżki z punktu B do punktu A.

# 3 Algorytm genetyczny

Algorytm genetyczny jest metodą rozwiązywania problemu komiwojażera, polegającą na symulowaniu działania procesu ewolucyjnego. W algorytmie tym, rozwiązania reprezentowane są przez tzw. chromosomy (np. permutacje miast), a ich jakość jest oceniana przez funkcję celu (np. długość trasy).

## 3.1 Opis krokowy

1. Inicjalizacja populacji - losowo generowane są początkowe rozwiązania
2. Ocena jakości rozwiązań - na podstawie funkcji celu (waga ścieżki) oceniana jest jakość każdego chromosomu
3. Selekcja - na podstawie ocen jakości wybierane są najlepsze rozwiązania, które przechodzą do kolejnej generacji
4. Krzyżowanie - wybrane rozwiązania są krzyżowane ze sobą, tworząc nowe chromosomy
5. Mutacja - losowe zmiany w chromosomach, mające na celu wprowadzenie nowych cech
6. Powtarzaj kroki 2-5 aż nie zajdzie własność stopu (ograniczenie czasowe)

## 3.2 Selekcja

Wykorzystana selekcja losowo wybiera dwa chromosomy i wybiera najlepszy z nich do kolejnej generacji.

## 3.3 Krzyżowanie

Order Crossover (OX) polega na losowym wyborze dwóch fragmentów (podciągów) z jednego chromosomu rodzica, a następnie składaniu ich w nowy chromosom potomny.

1. Losowo wybierane są dwa punkty cięcia na chromosomach rodziców, tworząc dwa fragmenty chromosomów.
2. Z pierwszego chromosomu rodzica kopiuje się fragment pomiędzy punktami cięcia i umieszcza go w odpowiedniej pozycji na chromosomie potomnym.
3. Następnie reszta genów na chromosomie potomnym jest uzupełniana genami z drugiego chromosomu rodzica, w takiej kolejności w jakiej się na nim pojawiają.

## 3.4 Mutacja

W algorytmie genetycznym mutacja jest procesem, który ma na celu wprowadzenie nowych cech do chromosomów, co pozwala na poszukiwanie nowych rozwiązań i zwiększanie szansy na znalezienie optymalnego rozwiązania.

- Transpozycja: polega na losowym wyborze dwóch genów na chromosomie i ich zamianie miejscami
- Odwracanie: polega na losowym wyborze dwóch punktów cięcia na chromosomie, a następnie odwróceniu kolejności genów pomiędzy tymi punktami.

# 4 Implementacja

By zaimplementować różne metody mutacji, całość logiki algorytmu zaimplementowano w abstrakcyjnej klasie *Genetic*. Następnie utworzono konkretne implementacje tej klasy, takie jak *GeneticInverseMutation*, przesłaniające abstrakcyjną metodę *Mutate()*.

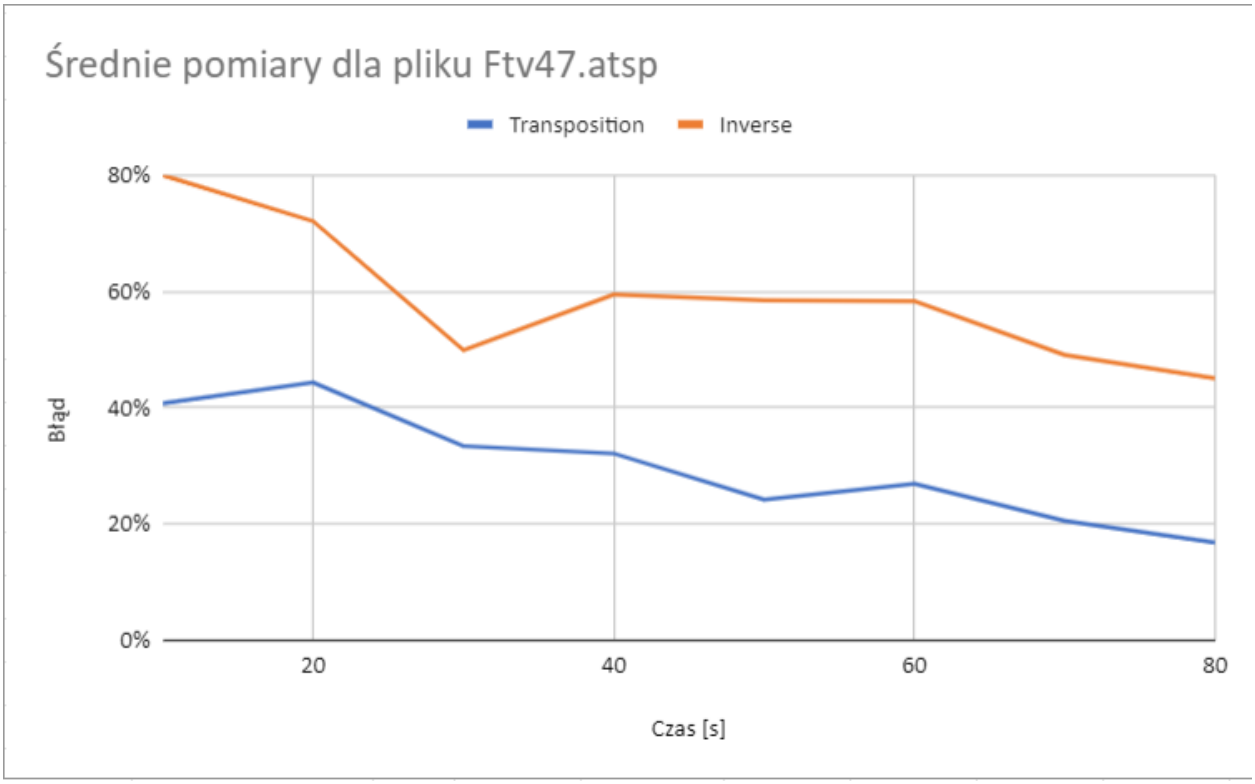
## 4.1 Najważniejsze klasy w programie

- *Genetic* - abstrakcyjna klasa przechowująca większość logiki.
- *GeneticInverseMutation* - implementacja *Genetic* implementująca odwracanie pozycji.
- *GeneticTranspositionMutation* - implementacja *Genetic* implementująca zamianę pozycjami.
- *Graph* - klasa przechowująca graf w postaci macierzy sąsiedztw oraz jego rozmiar.
- *GraphFactory* - klasa generująca losowy graf bądź ładująca graf z pliku tekstowego.

# 5 Pomiary błędu względnego

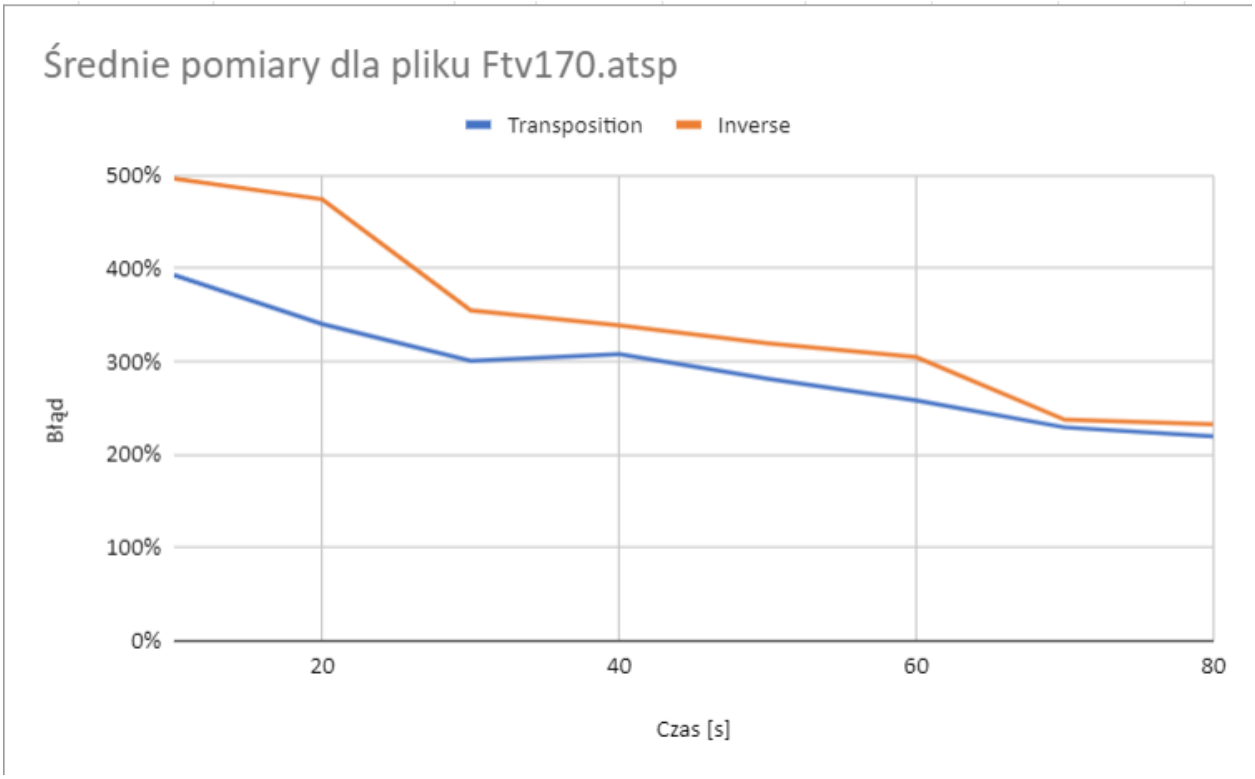
## 5.1 W zależności od czasu

### 5.1.1 ftv47.atsp



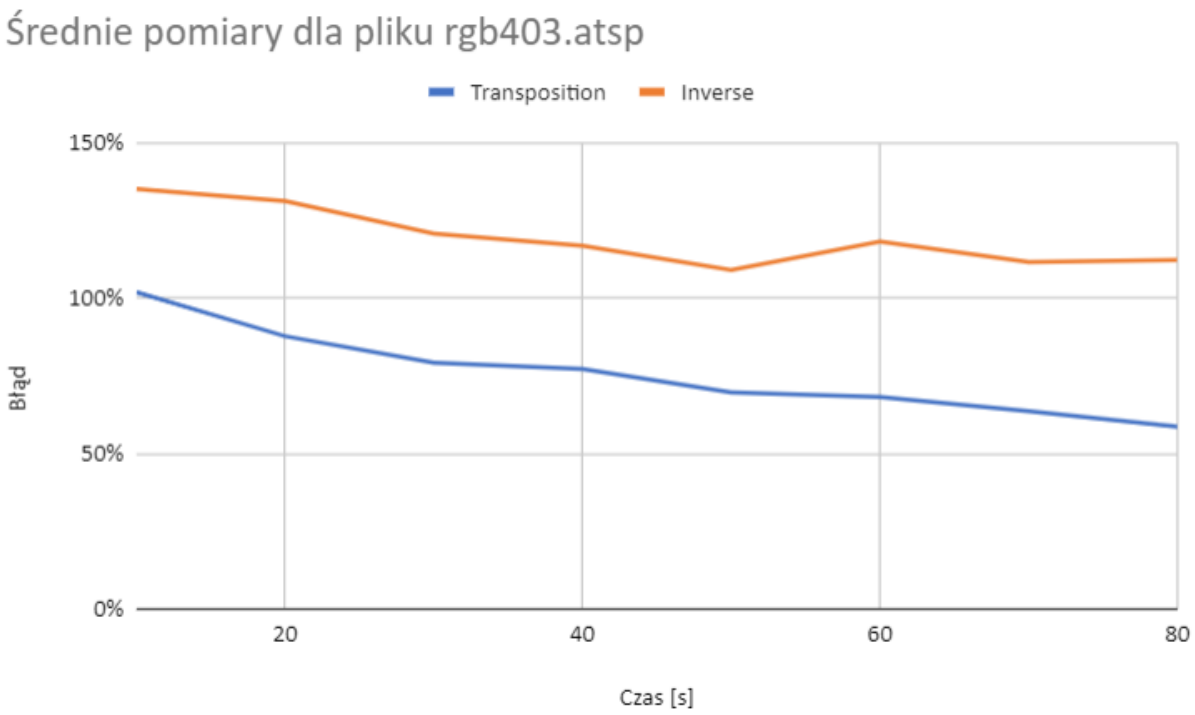

Wykres 1: Błąd względny w zależności od czasu - graf ftv47.atsp

### 5.1.2 ftv170.atsp




Wykres 2: Błąd względny w zależności od czasu - graf ftv170.atsp

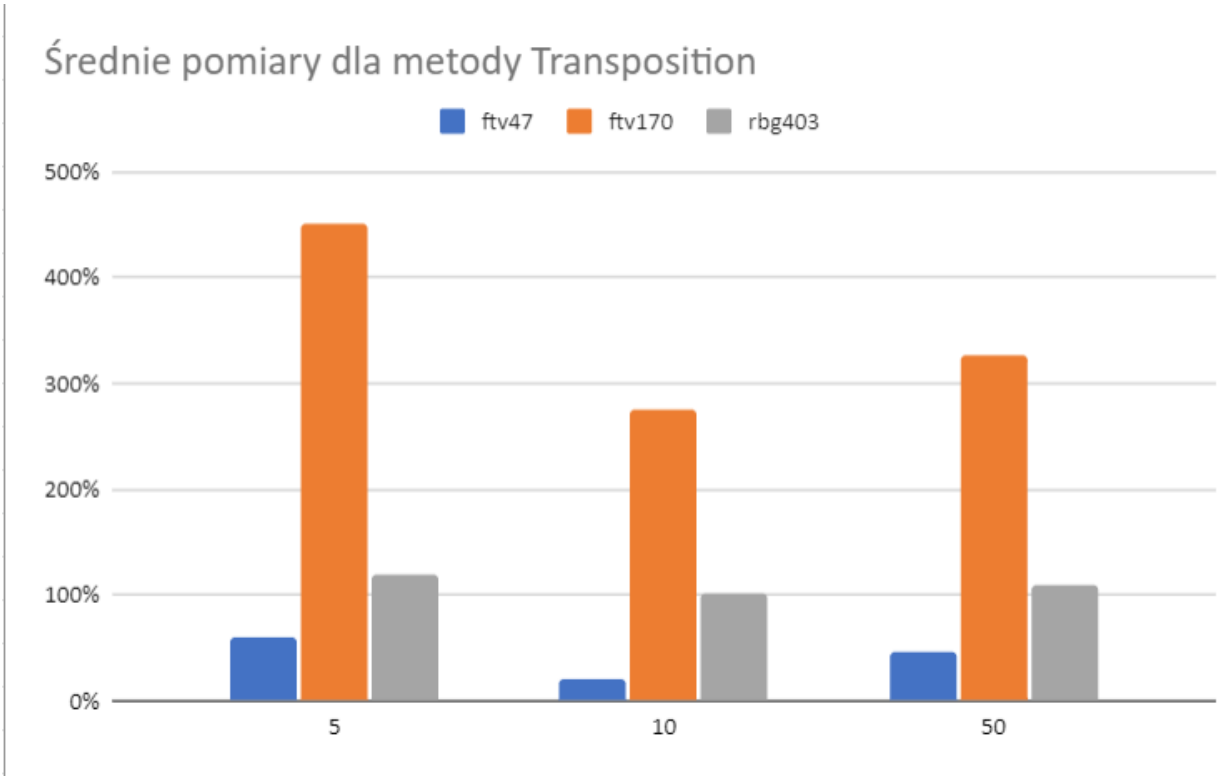
### 5.1.3 rgb403.atsp




Wykres 3: Błąd względny w zależności od czasu - graf rgb403.atsp

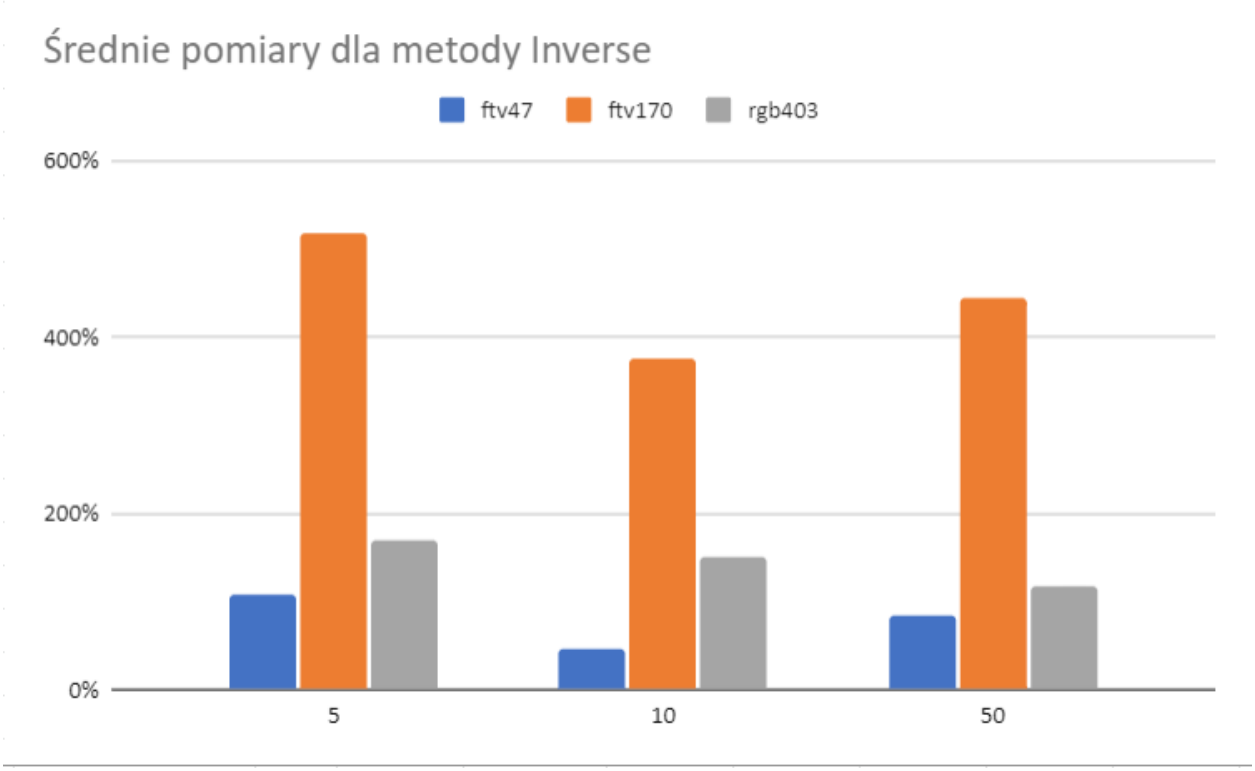
## 5.2 W zależności od populacji

### 5.2.1 Transposition




Wykres 4: Błąd względny w zależności od populacji

### 5.2.2 Inverse




Wykres 5: Błąd względny w zależności od populacji

# 6 Porównanie z przeszukiwaniem tabu

| graf | tabu search | algorytm genetyczny |
|---|---|---|
| ftv47 | 2196 | **1973** |
| ftv170 | **6443** | 8595 |
| rgb403 | **3461** | 4372 |

# 7 Wnioski

Po wynikach pomiarów widoczne jest to, że istotny wpływ na jakość rozwiązań w przypadku algorytmu genetycznego mają parametry początkowe. Średnie pomiary w zależności od populacji pokazują, że dla testowanych algorytmów optymalna populacja znajduje się w okolicach 10 osobników.

Zauważyć można także, że metoda mutacji *Inverse* daje gorsze rezultaty niż metoda *Transposition*. Podobne wyniki dawały analogiczne definicje sąsiedztwa w przeszukiwaniu tabu.

Porównując rezultaty algorytmu genetycznego z algorytmem tabu widać, że algorytm tabu poradził sobie lepiej, jednak nie w każdym przypadku.

# 8 Literatura


1. http://www.imio.polsl.pl/Dopobrania/Cw%20MH%2007%20(TSP).pdf
2. http://aragorn.pb.bialystok.pl/~wkwedlo/EA5.pdf