**PROJEKTOWANIE EFEKTYWNYCH ALGORYTMÓW**

**ZADANIE 3**

**PROWADZĄCY:**  
DR INŻ. JAROSŁAW MIERZWA

**AUTOR:**  
BARTOSZ RUDNIK 248893

**TERMIN ZAJĘĆ:**  
PN 15:15 – 16:55

WROCŁAW, 18.01.2021 r.

1. **Opis problemu**   
     
   Problem Komiwojażera (ang. Travelling salesman problem, TSP) jest to problem obliczeniowy polegający na wyznaczeniu w grafie takiego cyklu, który zawierać będzie wszystkie wierzchołki znajdujące się w grafie, każdy z tych wierzchołków odwiedzony zostanie wyłącznie jeden raz, a koszt tego cyklu będzie jak najmniejszy. Problem ten możemy wyobrazić sobie jako pracę kuriera, który na początku dnia pracy wyrusza z siedziby firmy i musi dostarczyć przesyłki do wyznaczonych miejsc, a następnie wrócić z powrotem do siedziby firmy. Rozwiązaniem problemu w przypadku pracy kuriera jest wyznaczenie mu drogi, która maksymalnie zminimalizuje koszty jego pracy. Jeśli spojrzymy na problem Komiwojażera z bardziej formalnej, matematycznej perspektywy to możemy go opisać jako problem polegający na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Cykl Hamiltona jest to taki cykl w grafie, w którym każdy z wierzchołków grafu oprócz wierzchołka startowego jest odwiedzany wyłącznie raz. Problem Komiwojażera możemy podzielić na symetryczny problem komiwojażera, w którym odległość pomiędzy dowolnym miastem A i B jest taka sama jak odległość między miastem B i A oraz asymetryczny problem Komiwojażera, w którym odległości pomiędzy miastami A i B oraz B i A mogą być różne. Największą trudnością na jaką możemy natrafić podczas rozwiązywania problemu komiwojażera jest liczba danych jaką musimy przeanalizować. Przykładowo jeśli badany przez nas graf ma ‘n’ wierzchołków to jeśli chcemy zbadać wszystkie możliwe cykle w grafie to otrzymamy (n – 1)! możliwych kombinacji. Prowadzi to do otrzymania złożoności obliczeniowej wynoszącej O((n – 1)!) co oznacza, że dla większych wartości n problem ten będzie nierozwiązywalny. Problem komiwojażera należy do problemów NP-trudnych, czyli nie są znane algorytmy, które są w stanie wyznaczyć rozwiązanie tego problemu w wielomianowej złożoności obliczeniowej. Trzecie zadanie projektowe polegało na zaimplementowaniu algorytmu genetycznego rozwiązującego problem komiwojażera. Algorytm Genetyczny należy do grupy algorytmów ewolucyjnych. Algorytm Genetyczny swoim działaniem naśladuje procesy ewolucyjne gatunków znane ze świata rzeczywistego. Swoją pracę algorytm genetyczny rozpoczyna od wytworzenia początkowej populacji osobników. Następnie stosowane są: metody selekcji mające na celu wybrać spośród populacji osobników przeznaczonych do rozmnażania, metody krzyżowania tworzące nowe osobniki, a także metody mutacji wprowadzające modyfikacje w wytworzonych nowych osobnikach.
2. **Generowanie początkowej populacji**  
     
   Algorytm swoje działanie rozpoczyna od wygenerowania początkowej populacji osobników o zadanym przez użytkownika rozmiarze. W swojej implementacji algorytmu genetycznego do generacji początkowej populacji użyłem:  
     
   **1) Algorytmu Zachłannego (1 osobnik)**  
   Algorytm zachłanny dokonuje pełnego przeglądu sąsiedztwa bieżącego rozwiązania x aż do momentu gdy w sąsiedztwie rozwiązania x nie istnieje rozwiązanie o mniejszej wartości funkcji celu. Algorytm zachłanny zwraca rozwiązanie będące lokalnym minimum, przeszukując przy tym bardzo niewielką część przestrzeni wszystkich rozwiązań.  
     
   **2) Algorytmu zachłanno-losowego (80% populacji początkowej)**  
   Algorytm zachłanno-losowy jest modyfikacją algorytmu zachłannego polegająca na wygenerowaniu pierwszych n wierzchołków w sposób losowy i dla w ten sposób częściowo wyznaczonej ścieżki użycia algorytmu zachłannego w celu jej uzupełnienia.   
      
   **3) algorytmu losowego (pozostali osobnicy)**   
   ------------TO-DO------------------
3. **Metody selekcji**  
     
    Metody selekcji służą do wyselekcjonowania z populacji osobników, którzy w dalszej części działania algorytmu zostaną użyci w metodach krzyżowania do stworzenia nowego pokolenia osobników. W swojej implementacji algorytmu genetycznego przygotowałem trzy metody selekcji rodziców:  
     
   **1) Selekcja turniejowa**  
   W selekcji turniejowej tworzymy n turniejów, do każdego z turniejów losujemy k osobników. Następnie z każdego z turniejów wybieramy najlepszego osobnika i tworzymy ponownie turniej dla wszystkich najlepszych osobników. Wynikiem działania selekcji turniejowej jest zwrócenie osobnika, który okazał się najlepszy w turnieju dla najlepszych osobników  
     
   **public** int[] tournament(int numberOfVertex,  
    int populationSize, int n) {

**Random** random = **new** **Random**();

int[] finalBest = **new** int[numberOfVertex + 2];

int finalBestCost = **Integer**.MAX\_VALUE;

**for** (int j = 0; j < n; j++) {

int[] actualBest = **new** int[numberOfVertex + 2];

int actualBestCost = **Integer**.MAX\_VALUE;

boolean[] test = **new** boolean[populationSize];

**for** (int i = 0; i < n; i++) {

int randomIndex = random.nextInt(populationSize - 1);

**if** (test[randomIndex]) {

i--;

**continue**;

} **else**

test[randomIndex] = **true**;

int[] currentRoute = population.get(randomIndex);

**if** (currentRoute[currentRoute.length-1]<actualBestCost){

actualBestCost = currentRoute[currentRoute.length - 1];

actualBest = currentRoute.clone();

}

}

**if** (actualBest[actualBest.length - 1] < finalBestCost) {

finalBestCost = actualBest[actualBest.length - 1];

finalBest = actualBest.clone();

}

}

**return** finalBest;

}

**2) Selekcja koła ruletki**  
Stosując selekcję koła ruletki dla każdego z osobników należących do naszej populacji musimy wyznaczyć ich funkcję zdatności. Wartość funkcji zdatności określa zdatność danego osobnika do rozmnożenia. Im większa jest wartość zdatności dla danego osobnika tym większa jest szansa wybrania go jako rodzica. Do określenia wartości funkcji zdatności możemy wykorzystać wzór: , gdzie k jest kosztem przejścia ścieżki badanego osobnika. Po wyznaczeniu wartości funkcji zdatności dla wszystkich osobników w populacji należy te wartości zsumować, a w dalszej kolejności obliczyć każdemu osobnikowi prawdopodobieństwo wyboru poprzez podzielenie wartości jego funkcji zdatności przez sumę wszystkich funkcji zdatności.  
  
**public** int[] roulette(int populationSize) {

**Random** random = **new** **Random**();

**List**<int[]> tmpPopulation = **new** **ArrayList**<>(population);

double[] fitnessValue = **new** double[populationSize];

double totalFitnessValue = 0.0;

double fitnessSum = 0.0;

int returnIndex = 0;

sortPopulation(tmpPopulation);

**for** (int i = 0; i < populationSize; i++) {

int[] route = tmpPopulation.get(i);

int pathCost = route[route.length - 1];

fitnessValue[i] = 1 / (double) pathCost;

}

**for** (int i = 0; i < populationSize; i++)

totalFitnessValue += fitnessValue[i];

**for** (int i = 0; i < populationSize; i++)

fitnessValue[i] /= totalFitnessValue;

double testValue = random.nextDouble();

**for** (int i = 0; i < populationSize; i++) {

fitnessSum += fitnessValue[i];

**if** (fitnessSum >= testValue) {

returnIndex = i;

**break**;

}

}

**return** tmpPopulation.get(returnIndex);

}   
  
**3) Selekcja rankingowa**  
  
Selekcja rankingowa podobnie jak selekcja koła ruletki korzysta z funkcji zdatności jednak jej wartość jest obliczana z innego wzoru niż to ma miejsce w przypadku selekcji koła ruletki. Prawdopodobieństwo wyboru danego osobnika w selekcji rankingowej zależy od miejsca tego osobnika w rankingu. Ranking ten sortuje wszystkie osobniki należące do populacji w kolejności od osobnika o najmniejszym koszcie przejścia ścieżki do osobnika o największym koszcie przejścia ścieżki. Wzór, który możemy wykorzystać do obliczenia wartości funkcji zdatności dla danego osobnika w selekcji rankingowej jest postaci: gdzie popSize jest rozmiarem populacji, a r jest pozycją w rankingu danego osobnika.   
  
**public** int[] ranking(int populationSize) {

**Random** random = **new** **Random**();

**List**<int[]> tmpPopulation = **new** **ArrayList**<>(population);

double[] fitnessValue = **new** double[populationSize];

double totalFitnessValue = 0.0;

double fitnessSum = 0.0;

int returnIndex = 0;

sortPopulation(tmpPopulation);

**for** (int i = 0; i < populationSize; i++) {

fitnessValue[i] = (populationSize - i) / (double) populationSize \* (populationSize - 1);

}

**for** (int i = 0; i < populationSize; i++)

totalFitnessValue += fitnessValue[i];

int helpRank = populationSize;

**for** (int i = 0; i < populationSize; i++) {

fitnessValue[i] = helpRank / totalFitnessValue;

helpRank--;

}

double testValue = random.nextDouble();

**for** (int i = 0; i < populationSize; i++) {

fitnessSum += fitnessValue[i];

**if** (fitnessSum >= testValue) {

returnIndex = i;

**break**;

}

}

**return** tmpPopulation.get(returnIndex);

}

1. **Metody krzyżowania**  
   Krzyżowanie polega na łączeniu cech osobników wybranych przez metody selekcji. Celem zastosowania metod krzyżowania jest wygenerowanie nowego pokolenia osobników, które dzięki wyborowi odpowiednich rodziców okaże się pokoleniem dającym lepsze wyniki niż poprzednie pokolenia. W mojej implementacji algorytmu genetycznego umieściłem następujące algorytmy krzyżowania: single-point crossover, two-point crossover, cycle crossover, cycle2 crossover, order crossover, partially mapped crossover, sequential constructive crossover oraz enhanced sequential constructive crossover. W dalszej części opisane zostaną Order Crossover oraz Enhanced Sequential Crossover, które podczas testów uzyskały najlepsze wyniki.   
     
   **1) Order Croossover**  
     
   W pierwszej kolejności należy wyznaczyć dwa indeksy i oraz j takie, że (i > j), pomiędzy którymi w osobnikach będących rodzicami stworzona zostanie sekcja dopasowania. Dane znajdujące się w sekcji dopasowania pierwszego rodzica kopiujemy do tworzonego potomka i umieszczamy na tych samych pozycjach co w pierwszym rodzicu. Następnie rozpoczynając od indeksu (i + 1) w przypadku gdy nie występuje kolizja (czyli gdy w dziecku umieszczony jest już kopiowany element) kopiujemy do dziecka dane z drugiego rodzica. W przypadku dojścia do końca ścieżki wracamy na jej początek. Dane są kopiowane do dziecka aż do momentu dojścia do początku sekcji dopasowania.  
     
   **Przykład:**  
   1. Wyznaczenie sekcji dopasowania  
   2. Skopiowanie danych z sekcji dopasowania pierwszego rodzica do dziecka  
   3. Skopiowanie danych z drugiego rodzica do dziecka  
      
   Najpierw z drugiego rodzica kopiujemy ‘5’, potem pomijamy ‘4’ ponieważ znajduje się już w dziecku. Ponieważ doszliśmy do końca ścieżki drugiego rodzica, a nie uzupełniliśmy jeszcze wszystkich elementów dziecka to wracamy na początek drugiego rodzica.  
   Kopiując do dziecka element 2 doszliśmy tym samym do miejsca rozpoczęcia sekcji dopasowania, czyli skończyliśmy proces tworzenia dziecka. Wynikowa struktura stworzonego dziecka przedstawia się następująco:  
   Implementacja algorytmu order crossover w języku Java użyta w projekcie:  
     
   **public** int[] order(int[][] graph, int[] firstParent, int[] secondParent, int numberOfVertex) {

int[] child = **new** int[numberOfVertex + 2];

**Random** random = **new** **Random**();

int i = 0;

int j = 0;

**while** (i == j) {

i = random.nextInt(numberOfVertex - 2) + 1;

j = random.nextInt(numberOfVertex - 2) + 1;

}

**if** (j > i) {

int tmp = i;

i = j;

j = tmp;

}

int[] chosenVertexes = **new** int[i - j + 1];

int tmpIndex = 0;

**for** (int start = j; start <= i; start++) {

chosenVertexes[tmpIndex] = firstParent[start];

tmpIndex++;

}

tmpIndex = i + 1;

int tmpIndex2 = i + 1;

**if** (tmpIndex == secondParent.length - 2)

tmpIndex = 1;

**if** (tmpIndex2 == secondParent.length - 2)

tmpIndex2 = 1;

**for** (int k = 0; k < numberOfVertex; k++) {

int tmpVertex = secondParent[tmpIndex2];

boolean test = **true**;

**for** (int chosenVertex : chosenVertexes) {

**if** (tmpVertex == chosenVertex) {

test = **false**;

**break**;

}

}

**if** (test) {

child[tmpIndex] = tmpVertex;

tmpIndex++;

}

tmpIndex2++;

**if** (tmpIndex == secondParent.length - 2)

tmpIndex = 1;

**if** (tmpIndex2 == secondParent.length - 2)

tmpIndex2 = 1;

}

tmpIndex2 = 0;

**for** (int k = j; k <= i; k++) {

child[k] = chosenVertexes[tmpIndex2];

tmpIndex2++;

}

child[0] = 0;

child[child.length - 2] = 0;

child[child.length - 1] = utils.getRouteCost(graph, child);

**return** child;

}

**2) Enhanced Sequential Crossover**  
  
Jest to algorytm krzyżowania, który w swoim działaniu oprócz wykorzystania rodziców, korzysta także z zadanej macierzy kosztów przejścia w celu stworzenia potomków lepszej jakości. Algorytm swoje działanie rozpoczyna od skopiowania pierwszego elementu z pierwszego rodzica do dziecka. Następnie w obu rodzicach wyznaczane są elementy występujące na następnej pozycji po elemencie skopiowanym do dziecka. Po wyznaczeniu tych elementów w obu rodzicach wybieramy element z tego rodzica, dla którego obliczony przewidywany koszt będzie mniejszy. Operacje tą wykonujemy według wzoru:  
  
, gdzie:  
Procedura wyboru kolejnych wierzchołków kopiowanych do dziecka odbywa się analogicznie aż do pełnego wypełnienia ścieżki dziecka.  
  
Przykład:  
1. Dane wejściowe

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | -1 | 45 | 23 | 12 | 67 | 29 |
| 1 | 45 | -1 | 78 | 42 | 90 | 56 |
| 2 | 23 | 78 | -1 | 34 | 38 | 22 |
| 3 | 12 | 42 | 34 | -1 | 78 | 99 |
| 4 | 67 | 90 | 38 | 78 | -1 | 5 |
| 5 | 29 | 56 | 22 | 99 | 5 | -1 |

2. Skopiowanie do dziecka pierwszego elementu z pierwszego rodzica  
  
3. Wyznaczenie elementów występujących po elemencie 2 i obliczenie przewidywanych kosztów  
ponieważ 43 < 76, to kolejnym dodanym do dziecka wierzchołkiem będzie 4.  
  
4. Wyznaczenie elementów występujących po elemencie 4   
  
ponieważ w rodzicu2 po elemencie 4 występuje element 0, który już znajduje się w dziecku musimy wybrać pierwszy nie występujący w dziecku element ze zbioru {1, 2, 3, 4, 5}.  
ponieważ 61 < 132 to kolejnym elementem dodanym do dziecka będzie 5.  
  
Wykonujemy analogiczne kroki aż do momentu pełnego wypełnienia ścieżki dziecka.  
  
5. Ostateczna struktura stworzonego dziecka  
  
Implementacja algorytmu enhanced sequential constructive crossover w języku Java użyta w projekcie:  
  
**public** int[] enhancedSequentialConstructive(int[][] graph, int[] firstParent, int[] secondParent, int numberOfVertex) {

int[] child = **new** int[numberOfVertex + 2];

int position = 1;

int addNode = firstParent[position];

int firstNode;

int secondNode;

int firstParentIndex;

int secondParentIndex;

child[position] = addNode;

position++;

**while** (position < numberOfVertex) {

firstParentIndex = sequentialIndex(numberOfVertex, firstParent, addNode);

secondParentIndex = sequentialIndex(numberOfVertex, secondParent, addNode);

firstNode = sequentialNode(numberOfVertex, firstParentIndex, position, firstParent, child);

secondNode = sequentialNode(numberOfVertex, secondParentIndex, position, secondParent, child);

int firstParentMin = minRowValue(firstNode, graph, numberOfVertex, child);

int secondParentMin = minRowValue(secondNode, graph, numberOfVertex, child);

**if** (graph[addNode][firstNode] + firstParentMin < graph[addNode][secondNode] + secondParentMin) {

child[position] = firstNode;

addNode = firstNode;

} **else** {

child[position] = secondNode;

addNode = secondNode;

}

position++;

}

child[0] = 0;

child[child.length - 2] = 0;

child[child.length - 1] = utils.getRouteCost(graph, child);

**return** child;

}

1. **Metody mutacji**Mutacja wprowadza do osobników losowe zmiany, jej zadaniem jest wprowadzenie różnorodności w populacji czyli zabieganie przedwczesnej zbieżności algorytmu. Mutacje zachodzą z zadanym przez użytkownika prawdopodobieństwem. Zbyt małe prawdopodobieństwo mutacji może spowodować zbyt dużą jednorodność populacji, zbyt duże prawdopodobieństwo mutacji może spowodować zbyt duże pogorszenie dobrych osobników wytworzonych podczas krzyżowania. W swojej implementacji algorytmu genetycznego przygotowałem czy algorytmu mutacyjne: swap, insert oraz reverse. Algorytmy insert oraz reverse pozwalały osiągnąć lepsze wyniki i to one będą opisane w dalszej części.  
     
   **1) Insert**  
     
   Funkcja insert(i, j) umieszcza wierzchołek o wartości ‘i’ na pozycji o indeksie ‘j’ w zadanej tablicy. Wywołanie funkcji insert (5, 2) dla ścieżki [0, 1, 2, 3, 4, 5, 0] zwróci następującą ścieżkę [0, 1, 5, 2, 3, 4, 0].

Implementacja funkcji insert() w języku JAVA wykorzystana do przeprowadzenia mutacji:   
  
**public** int[] insertRoute(int[] route, int i, int j) {

int indexI = 0;

int tmp = 0;

int[] tmpArray = **new** int[route.length];

**for** (int k = 1; k < route.length - 1; k++) {

**if** (route[k] == i) {

indexI = k;

**break**;

}

}

**if** (indexI > j) {

**for** (int k = j; k < route.length; k++) {

**if** (route[k] != i)

tmpArray[k - tmp] = route[k];

**else**

tmp = 1;

}

route[j] = i;

**if** (route.length - (j + 1) >= 0)

**System**.arraycopy(tmpArray, j + 1 - 1, route, j + 1, route.length - (j + 1));

} **else** {

**if** (j - indexI >= 0)

**System**.arraycopy(route, indexI + 1, route, indexI, j - indexI);

route[j] = i;

}

**return** route;

}

**2) Reverse**Funkcja reverse(i, j) wykonuje odwrócenie kolejności pomiędzy wierzchołkami o wartości ‘i’ oraz ‘j’, włącznie z wierzchołkiem o wartości ‘i’ i ‘j’. Wywołanie funkcji reverse(1, 3) dla tablicy [0, 1, 2, 3, 4, 5, 0] zwróci tablicę [0, 3, 2, 1, 4, 5, 0].   
  
Implementacja funkcji reverse() w języku JAVA wykorzystana w projekcie:  
  
**public** int[] reverseRoute(int[] route, int i, int j) {

int[] index = getIndex(route, i, j);

int iIndex = index[0];

int jIndex = index[1];

**if** (iIndex < jIndex) {

**while** (iIndex < jIndex) {

int tmp = route[iIndex];

route[iIndex] = route[jIndex];

route[jIndex] = tmp;

iIndex++;

jIndex--;

}

} **else** {

**while** (jIndex < iIndex) {

int tmp = route[iIndex];

route[iIndex] = route[jIndex];

route[jIndex] = tmp;

iIndex--;

jIndex++;

}

}

**return** route;

}

1. **Elitaryzm**Jest to mechanizm pozwalający zachować n najlepszych osobników ze starej populacji. Użytkownik może podać wartość parametru n w celu dopasowania go do badanego problemu. Zastosowanie mechanizmu elitaryzmu pozwala na zachowanie w populacji najlepiej dostosowane osobniki dzięki czemu nadal istnieje szansa, że będą mogły się rozmnożyć.
2. **Algorytm Memetyczny**Jest to algorytm będący rozszerzeniem algorytmu genetycznego, oprócz operacji znanych w algorytmie genetycznym takich jak: operacje selekcji, krzyżowania oraz mutacji w algorytmie memetycznym dochodzi operacja lokalnej optymalizacji. Celem wykonywania lokalnej optymalizacji jest osiągnięcie lepszych rozwiązań. W mojej implementacji algorytmu genetycznego rozszerzenie memetyczne może zostać włączone lub wyłączone przez użytkownika. W metodzie wykonującej lokalną optymalizację w zależności od wyboru użytkownika wykorzystywana jest jedna z trzech dostępnych metod:  
   1)  Insert  
   2)  Swap  
   3) Reverse  
   Metody Insert oraz Reverse zostały opisane w rozdziale dotyczącym mutacji. Poniżej zamieszczony został opis funkcji swap:  
     
   Funkcja swap(i, j) zamienia miejscami wierzchołek o numerze ‘i’ z wierzchołkiem o numerze ‘j’ Wywołanie funkcji swap(3,5) dla tablicy wierzchołków [0, 1, 2, 3, 4, 5] zwróci następującą tablice: [0, 1, 2, 5, 4, 3]. Implementacja funkcji swap() w języku JAVA wykorzystana w projekcie:  
     
   **public** int[] swapRoute(int[] route, int i, int j) {

int[] index = getIndex(route, i, j);

int iIndex = index[0];

int jIndex = index[1];

int tmp = route[iIndex];

route[iIndex] = route[jIndex];

route[jIndex] = tmp;

**return** route;

}

Metoda lokalnej optymalizacji przeszukuje sąsiedztwo danego użytkownika i zapamiętuje wykorzystane parametry, dla których udało się uzyskać najbardziej korzystny rezultat.  
  
**public** int[] bestRoute(int[][] graph, int[] route, int numberOfVertex, int mutationType) {

int[] parameters = **new** int[2];

int bestCost = **Integer**.MAX\_VALUE;

**for** (int i = 1; i < numberOfVertex - 1; i++) {

**for** (int j = i + 1; j < numberOfVertex; j++) {

int[] newRoute = route.clone();

**if** (mutationType == 0)

newRoute = insertRoute(newRoute, i, j);

**else** **if** (mutationType == 1)

newRoute = swapRoute(newRoute, i, j);

**else**

newRoute = reverseRoute(newRoute, i, j);

newRoute[newRoute.length - 1] = utils.getRouteCost(graph, newRoute);

**if** (newRoute[newRoute.length - 1] < bestCost) {

bestCost = newRoute[newRoute.length - 1];

parameters[0] = i;

parameters[1] = j;

}

}

}

**return** parameters;

}

1. **Przebieg działania algorytmu**Algorytm swoje działanie rozpoczyna od obliczeniawyznaczenie czasu zakończenia swojego działania, następnie generowana jest początkowa populacja. Rozmiar populacji jest parametrem ustawianym przez użytkownika. W dalszej kolejności wyznaczane jest najlepsze dotychczasowe rozwiązanie. Kolejnym krokiem w algorytmie jest pętla while działająca do momentu osiągnięcia warunku stopu algorytmu. W pętli tej znajduje się druga pętla while służąca do generacji nowych osobników. W zależności od wyboru użytkownika rodzice mogą być wybrani przy pomocy selekcji turniejowej, selekcji rankingowej lub selekcji koła ruletki. Następnie jeśli spełniony został warunek prawdopodobieństwa krzyżowania to przy pomocy wyznaczonych wcześniej rodziców powstają nowe osobniki. Po wytworzeniu wymaganej liczby osobników nowej, tymczasowej populacji algorytm przechodzi do pętli for, w której odbywa się mutacja osobników należących do nowo utworzonej, tymczasowej populacji. Mutacja zachodzi jeśli został spełniony warunek prawdopodobieństwa mutacji. Prawdopodobieństwo mutacji oraz prawdopodobieństwo krzyżowania są parametrami ustawianymi przez użytkownika. Po zakończeniu procesu mutacji, w zależności od wyboru użytkownika może zostać przeprowadzony proces lokalnej optymalizacji. Kolejnym krokiem jest dodanie osobników należących do tymczasowej populacji do populacji głównej z zachowaniem zasady elitaryzmu. Ostatnim krokiem wykonywanym w głównej pętli while jest sprawdzenie czy nowo wytworzona populacja zawiera w sobie lepszego osobnika niż dotychczasowy najlepszy osobnik. Po spełnieniu warunku stopu algorytmu wypisywana jest ścieżka należąca do najlepszego znalezionego osobnika oraz koszt jej przejścia.  
    **public** int[] algorithm(int[][] graph, int numberOfVertex, int seconds, int populationSize, int exclusivity, int selectionType, int crossoverType, double mutationChance, int mutationType, int memeticType, double crossoverChance, boolean memetic) {

**Random** random = **new** **Random**();

Crossover crossover = **new** Crossover(numberOfVertex);

Mutation mutation = **new** Mutation(numberOfVertex);

Selection selection = **new** Selection(population);

int[] bestRoute = **new** int[numberOfVertex + 2];

bestRoute[bestRoute.length - 1] = **Integer**.MAX\_VALUE;

generatePopulation(graph, numberOfVertex, populationSize);

long finishTime = **System**.currentTimeMillis() + seconds \* 1000L;

**for** (int[] route : population) {

**if** (route[route.length - 1] < bestRoute[bestRoute.length - 1]) {

bestRoute = route.clone();

}

}

**while** (**System**.currentTimeMillis() < finishTime) {

int[] firstParent;

int[] secondParent;

**List**<int[]> newPopulation = **new** **ArrayList**<>();

**while** (newPopulation.size() < (populationSize - exclusivity)) {

double chance = random.nextDouble();

int[] child1;

int[] child2;

**if** (selectionType == 0) {

firstParent = selection.tournament(numberOfVertex, population.size(), 2);

secondParent = selection.tournament(numberOfVertex, population.size(), 2);

} **else** **if** (selectionType == 1) {

firstParent = selection.roulette(population.size());

secondParent = selection.roulette(population.size());

} **else** {

firstParent = selection.ranking(population.size());

secondParent = selection.ranking(population.size());

}

**if** (crossoverChance >= chance) {

**if** (crossoverType == 0) {

child1 = crossover.twoPoint(graph, firstParent, secondParent, numberOfVertex);

child2 = crossover.twoPoint(graph, secondParent, firstParent, numberOfVertex);

} **else** **if** (crossoverType == 1) {

child1 = crossover.order(graph, firstParent, secondParent, numberOfVertex);

child2 = crossover.order(graph, secondParent, firstParent, numberOfVertex);

} **else** **if** (crossoverType == 2) {

child1 = crossover.partiallyMapped(graph, firstParent, secondParent, numberOfVertex);

child2 = crossover.partiallyMapped(graph, secondParent, firstParent, numberOfVertex);

} **else** **if** (crossoverType == 3) {

child1 = crossover.cycle(graph, firstParent, secondParent, numberOfVertex);

child2 = crossover.cycle(graph, secondParent, firstParent, numberOfVertex);

} **else** **if** (crossoverType == 4) {

int[][] children = crossover.cycle2(graph, firstParent, secondParent, numberOfVertex);

child1 = children[0];

child2 = children[1];

} **else** **if** (crossoverType == 5) {

child1 = crossover.sequentialConstructive(graph, firstParent, secondParent, numberOfVertex);

child2 = crossover.sequentialConstructive(graph, secondParent, firstParent, numberOfVertex);

} **else** **if** (crossoverType == 6) {

child1 = crossover.enhancedSequentialConstructive(graph, firstParent, secondParent, numberOfVertex);

child2 = crossover.enhancedSequentialConstructive(graph, secondParent, firstParent, numberOfVertex);

} **else** {

child1 = crossover.singlePoint(graph, firstParent, secondParent, numberOfVertex);

child2 = crossover.singlePoint(graph, secondParent, firstParent, numberOfVertex);

}

newPopulation.add(child1);

newPopulation.add(child2);

}

}

**for** (int[] route : newPopulation) {

double chance = random.nextDouble();

**if** (mutationChance >= chance) {

int start = 0;

int end = 0;

**while** (start == end) {

start = random.nextInt(numberOfVertex - 2) + 1;

end = random.nextInt(numberOfVertex - 2) + 1;

}

**if** (mutationType == 0)

route = mutation.insertRoute(route, start, end);

**if** (mutationType == 1)

route = mutation.swapRoute(route, start, end);

**if** (mutationType == 2)

route = mutation.reverseRoute(route, start, end);

route[route.length - 1] = utils.getRouteCost(graph, route);

}

}

**if** (memetic) {

**for** (int[] route : newPopulation) {

int[] parameters = mutation.bestRoute(graph, route, numberOfVertex, memeticType);

**if** (memeticType == 0)

route = mutation.insertRoute(route, parameters[0], parameters[1]);

**else** **if** (memeticType == 1)

route = mutation.swapRoute(route, parameters[0], parameters[1]);

**else**

route = mutation.reverseRoute(route, parameters[0], parameters[1]);

route[route.length - 1] = utils.getRouteCost(graph, route);

}

}

clearPopulation(population.size(), exclusivity);

population.addAll(newPopulation);

**for** (int[] route : population) {

**if** (route[route.length - 1] < bestRoute[bestRoute.length - 1]) {

bestRoute = route.clone();

}

}

}

**for** (int i : bestRoute)

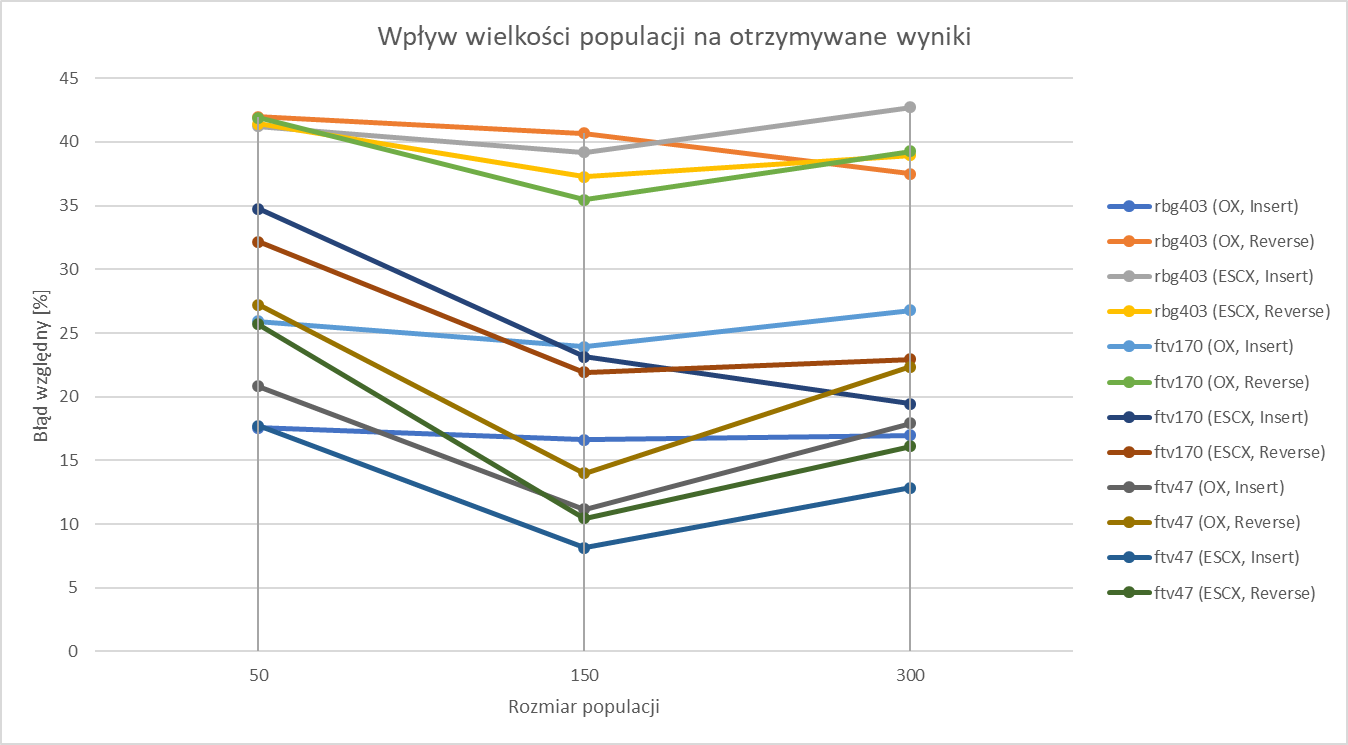
**System**.out.print(i + " ");

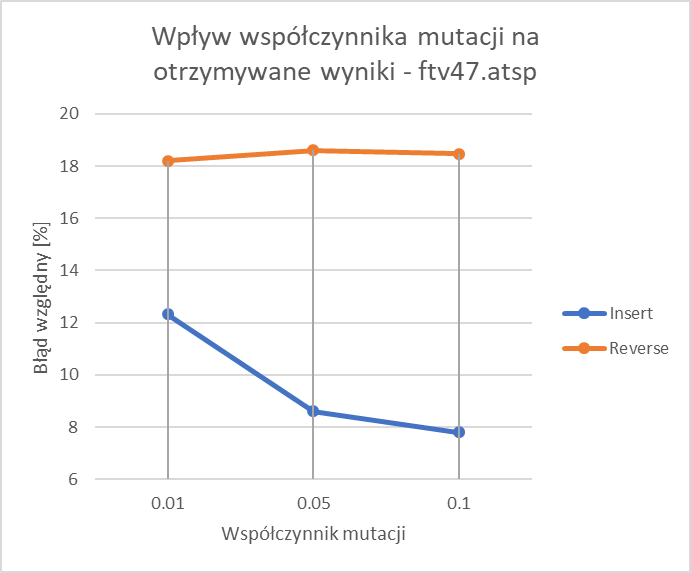
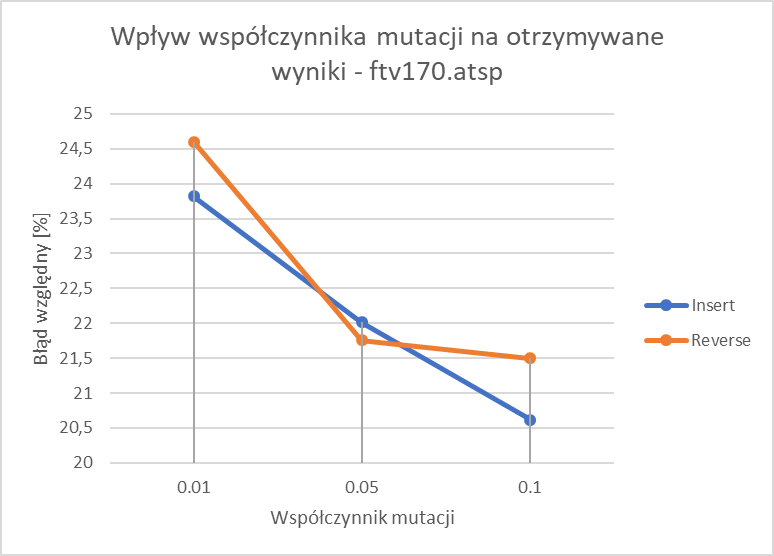
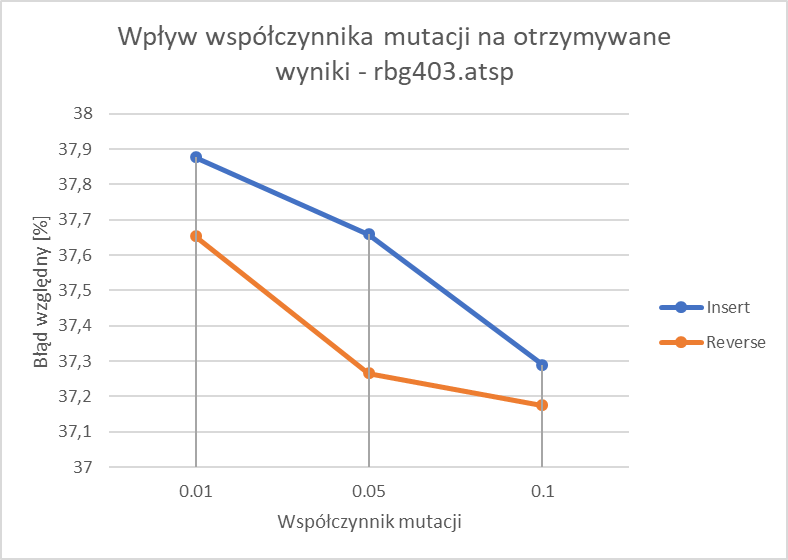
**return** bestRoute;

}

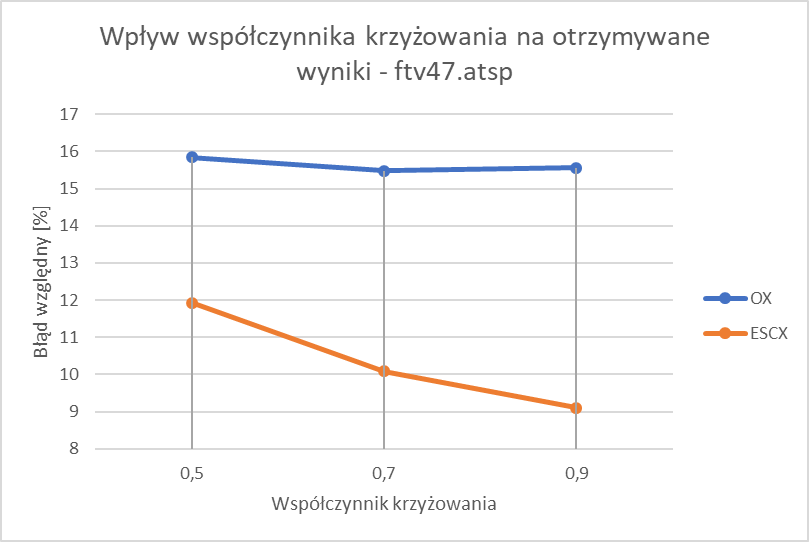
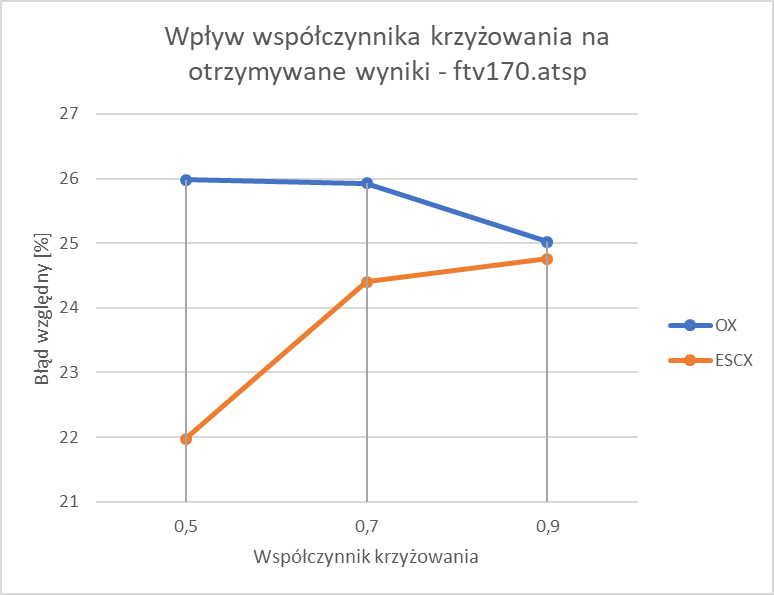
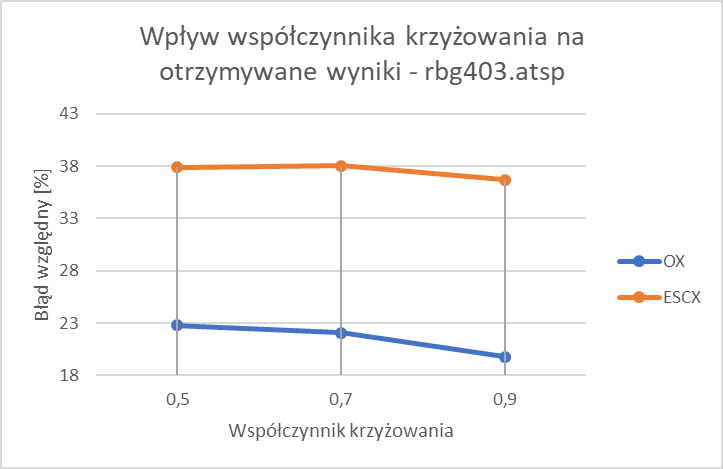
1. **Plan eksperymentu pomiarowego**  
     
   Algorytmy zostały zaimplementowane przy użyciu języka programowania JAVA. Pomiary przeprowadzone zostały w systemie operacyjnym Ubuntu 20.04.01 LTS. Procesor, na którym przeprowadzono pomiary to Intel Core i5-8250u o bazowej częstotliwości 1,60 Ghz.

Eksperyment pomiarowy został przeprowadzony dla danych zawartych w trzech plikach:  
 1) ftv47.atsp  
 2) ftv170.atsp  
 3) rbg403.atsp  
  
Dla wymienionych plików przeprowadzone zostały pomiary błędu względnego. Dla algorytmu genetycznego wykonane zostały pomiary badające wpływ wielkości populacji na otrzymywane wyniki. Dla najlepszego rozmiaru populacji uzyskanego podczas tych pomiarów zostały również przeprowadzone pomiary wpływu współczynnika mutacji na otrzymywane wyniki oraz wpływu współczynnika krzyżowania na wyniki. Dla pliku ftv47.atsp czas wykonywanie się algorytmu ustawiony był na 20 sekund, dla pliku ftv170.atsp było to 40 sekund, a dla pliku rbg403.atsp było to 60 sekund.

1. **Eksperyment pomiarowy  
     
   10.1 Badanie wpływu wielkości populacji na otrzymywane wyniki** ****Na podstawie przeprowadzonych pomiarów możemy zauważyć, że dla większości przeprowadzonych testów rozmiarem populacji dającym najlepsze wyniki okazała się populacja licząca 150 osobników. Zdecydowanie najgorsza okazała się populacja licząca 50 osobników. Populacja licząca 300 osobników dla dwóch testów okazała się widocznie lepsza od pozostałych rozmiarów populacji jednak dla pozostałych 10 testów uzyskała gorsze rezultaty od populacji z 150 osobnikami. Biorąc pod uwagę wyniki powyższych pomiarów do kolejnych pomiarów zdecydowałem się wybrać populacje liczącą 150 osobników.

**10.2 Badanie wpływu współczynnika mutacji na otrzymywane wyniki**a) ftv47.atsp ****b) ftv170.atsp  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
c) rbg403.atsp  
  
  
  
  
  
  
  
  
  
Na podstawie przeprowadzonych pomiarów możemy zauważyć, że dla plików ftv47.atsp oraz ftv170.atsp lepszą metodą mutacji okazała się metoda Insert, a dla pliku rbg403.atsp była to metoda Reverse. Ze wszystkich przeprowadzonych w tym punkcie pomiarów jedynie dla metody Reverse dla pliku ftv47.atsp wzrost wartości współczynnika pomiarów spowodował pogorszenie uzyskiwanych rezultatów, dla pozostałych przypadków testowych wzrost współczynnika mutacji umożliwiał otrzymywanie coraz to lepszych wyników. W przypadku metody Insert dla pliku ftv47.atsp wzrost ten był znaczący natomiast dla metod Insert i Reverse dla plików ftv170.atsp oraz rbg403.atsp wzrost ten okazał się niewielki.

**10.3 Wpływ współczynnika krzyżowania na otrzymywane wyniki**a) ftv47.atsp  
  
  
  


****b) ftv170.atsp ****c) rbg403.atsp ****Analizując wyniki pomiarów możemy zauważyć, że dla plików ftv47.atsp oraz ftv170.atsp lepszą metodą krzyżowania osobników okazała się metoda ESCX, a dla pliku rbg403.atsp była to metoda OX. W przypadku pliku rbg403.atsp dla obu metod krzyżowania możemy zauważyć polepszenie jakości otrzymywanych wyników wraz ze wzrostem wartości współczynnika krzyżowania. Dla pliku ftv170.atsp w przypadku krzyżowania OX wraz ze wzrostem współczynnika krzyżowania algorytm zwraca rozwiązania coraz to lepszej jakości natomiast w przypadku krzyżowania ESCX wzrost współczynnika krzyżowania spowodował uzyskanie rozwiązań gorszej jakości. W przypadku metody krzyżowania ESCX dla pliku ftv47.atsp możemy zauważyć zdecydowany wzrost jakości otrzymywanych rozwiązań wraz ze wzrostem współczynnika krzyżowania, dla metody krzyżowania OX wzrost jakości rozwiązań jest praktycznie niezauważalny.  
  
**10.4 Porównanie najlepszych wyników algorytmu Tabu Search z najlepszymi wynikami Algorytmu Genetycznego  
  
a) ftv47.atsp**Najlepszym wynikiem uzyskanym przez Tabu Search dla pliku ftv47.atsp był wynik **1776** wraz ze ścieżką:   
0-25-1-9-33-27-2-41-43-22-20-38-37-18-17-12-32-7-23-34-13-46-36-14-35-15-16-45-39-19-44-21-40-47-26-42-28-3-24-4-29-30-31-5-6-8-11-10-0

Najlepszym wynikiem uzyskanym przez algorytm genetyczny dla pliku ftv47.atsp był wynik **1790** wraz ze ścieżką:  
0-25-1-9-33-27-2-28-3-24-4-29-30-5-31-6-10-8-11-37-38-18-17-12-32-7-23-34-13-46-36-14-35-15-16-45-39-19-44-21-40-47-26-22-41-43-42-20-0  
  
**b) ftv170.atsp**  
  
Najlepszym wynikiem uzyskanym przez Tabu Search dla pliku ftv170.atsp był wynik **3268**wraz ze ścieżką:  
0-1-2-77-73-170-49-50-51-52-53-43-55-54-58-59-60-68-67-167-70-87-86-85-84-69-66-63-64-56-57-62-61-65-88-153-154-89-90-91-94-96-97-98-95-92-93-166-107-106-105-165-163-99-100-102-103-117-118-119-120-121-122-123-162-101-104-114-113-164-127-126-125-129-128-130-131-132-133-134-6-7-8-9-10-76-74-75-11-12-13-17-18-19-20-21-29-22-23-26-27-28-30-31-33-34-156-40-39-38-35-36-157-41-155-42-45-44-46-47-48-168-72-78-82-79-80-81-3-4-5-169-111-110-109-108-83-71-37-158-32-15-159-16-24-25-150-161-160-14-151-152-142-149-148-147-137-136-138-135-139-140-115-116-124-146-145-144-143-141-112-0  
  
Najlepszym wynikiem uzyskanym przez algorytm genetyczny dla pliku ftv170.atsp był wynik **3149**wraz ze ścieżką:  
0 77 1 2 3 4 5 169 111 110 109 107 106 105 165 163 99 100 102 103 117 118 119 120 121 122 123 162 101 98 95 92 93 166 108 83 84 69 66 65 64 56 57 62 63 88 153 154 89 90 91 94 96 97 104 114 113 164 127 126 125 129 128 130 135 138 139 140 141 6 7 8 9 10 76 74 75 11 12 18 19 20 158 32 36 157 33 31 30 28 29 16 17 21 22 23 26 27 24 15 159 13 37 38 39 40 35 34 156 44 41 155 42 45 46 47 48 51 52 53 43 55 54 58 59 60 61 68 67 167 70 87 85 86 71 50 49 170 73 168 72 78 82 79 80 81 112 132 133 134 131 115 116 124 137 136 146 145 144 143 147 148 149 161 152 142 14 25 150 160 151 0  
  
**c) rbg403.atsp**  
  
Najlepszym wynikiem uzyskanym przez Tabu Search dla pliku rbg403.atsp był wyniki **2513**wraz ze ścieżką:  
0-206-267-46-23-14-62-13-205-204-24-36-95-334-32-274-83-33-376-68-38-270-19-18-42-402-287-107-61-257-43-34-295-50-56-103-308-394-225-58-8-6-64-3-2-386-47-112-397-5-286-273-370-322-272-28-102-314-11-152-310-29-220-84-281-77-31-52-40-39-78-226-147-79-69-245-81-22-21-82-247-231-288-86-101-153-75-126-27-131-164-15-360-96-66-60-44-87-385-67-94-88-353-263-177-90-72-57-160-25-141-116-232-201-89-140-92-51-49-37-65-301-120-392-351-293-93-145-364-303-71-150-349-73-99-389-187-373-355-115-114-304-260-313-20-302-118-372-278-122-119-168-182-240-219-242-117-213-356-123-318-179-368-251-328-317-104-203-387-384-383-238-146-144-105-391-154-235-113-106-340-163-365-158-339-374-363-289-108-255-109-275-210-110-265-326-258-91-359-254-223-214-192-189-162-48-124-284-290-125-74-216-200-198-307-323-305-215-309-127-191-237-234-228-224-129-130-285-282-357-358-395-132-243-175-291-315-133-1-85-333-30-161-148-325-134-306-319-135-280-279-217-9-312-35-172-26-194-324-320-202-139-4-336-292-329-294-236-227-381-253-268-16-188-299-259-142-248-348-361-229-393-283-277-352-149-347-401-354-342-266-218-197-166-300-97-178-138-297-151-59-111-369-221-199-335-128-208-276-298-171-63-337-230-382-212-176-156-390-311-55-155-7-331-378-398-396-366-321-362-157-269-169-10-170-271-327-233-380-379-195-241-330-343-239-70-345-173-344-332-296-167-174-185-136-193-211-262-261-207-45-375-371-350-252-399-377-180-165-183-17-12-190-100-98-209-41-222-250-246-244-80-264-121-159-143-316-256-249-137-54-367-76-341-186-53-400-181-346-338-388-184-196-0  
  
Najlepszym wynikiem uzyskanym przez algorytm genetyczny dla pliku rbg403.atsp był wynik **2906**wraz ze ścieżką**:**  
0 30 332 310 29 77 31 52 40 39 78 226 147 79 69 245 81 22 21 82 247 54 308 26 86 367 75 25 399 163 116 387 67 66 60 44 88 96 94 90 72 57 91 206 160 365 181 1 267 281 92 51 49 37 288 249 301 120 392 351 293 93 145 338 213 41 85 87 56 118 165 95 364 303 71 97 312 260 253 70 218 349 73 99 389 307 359 202 164 15 158 269 131 355 115 114 353 263 102 372 278 122 119 168 103 5 141 356 328 317 104 384 383 197 146 144 105 391 154 106 340 209 363 289 108 255 379 190 212 109 275 210 110 326 258 111 254 223 214 192 189 123 48 358 327 124 284 290 125 74 216 200 198 126 155 323 305 215 309 127 191 128 228 224 129 219 172 273 395 130 187 193 194 196 45 207 4 195 100 98 132 243 175 291 315 133 148 325 134 306 319 135 280 279 136 236 227 320 137 184 321 362 139 336 292 329 294 231 12 178 138 266 201 188 173 344 239 345 299 259 142 229 233 380 234 235 237 238 7 240 149 347 352 150 300 232 251 156 390 151 11 342 297 152 16 324 381 153 401 354 378 337 361 276 298 171 63 377 157 339 179 161 186 53 162 252 371 166 80 10 167 170 334 398 396 357 330 388 348 343 374 271 350 360 176 159 143 397 296 182 230 382 241 113 242 117 208 177 250 366 368 174 180 369 183 217 203 221 199 222 185 220 244 248 246 302 89 304 35 311 313 346 314 59 393 318 333 169 27 335 341 375 277 283 17 285 282 370 286 385 265 402 316 256 23 14 62 13 205 204 24 36 32 274 46 33 376 68 38 270 19 18 42 287 83 43 34 295 50 394 225 58 8 6 64 3 2 61 107 386 47 112 322 257 65 9 84 272 28 101 55 400 140 76 211 121 262 261 264 268 331 20 373 0

1. **Wnioski**Algorytm genetyczny jest narzędziem mogącym zostać zastosowanym do wielu problemów optymalizacyjnych. W przypadku problemu komiwojażera najważniejszą rolę w algorytmie genetycznym odgrywa dobrze dobrane połączenie algorytmu selekcji rodziców wraz z algorytmem krzyżowania. Zastosowanie mechanizmu elitaryzmu pozwala zachować w populacji najlepiej przystosowane osobniki co wraz z upływem iteracji algorytmu przyczynia się do tworzenia coraz to lepszych dzieci. Porównując wyniki osiągane przez algorytm genetyczny z wynikami uzyskanymi podczas drugiego etapu projektu przez algorytm Tabu Search, algorytm genetyczny uzyskał lepszy wynik wyłącznie dla pliku ftv170.atsp, dla pozostałych dwóch plików testowych skuteczniejszą metodą rozwiązywania problemu komiwojażera okazał się Tabu Search. Podobnie jak miało to miejsce w przypadku algorytmu Symulowanego Wyżarzania oraz Tabu Search tak samo w przypadku algorytmu genetycznego bardzo ważną rolę w uzyskiwaniu rozwiązań dobrej jakości mają odpowiednio dobrane parametry, które mogą się różnić w zależności od badanej instancji problemu.
2. **Bibliografia**  
   - <http://aragorn.pb.bialystok.pl/~wkwedlo/EA5.pdf>, dostęp 18.01.2021 r.  
   -  <https://pl.wikipedia.org/wiki/Algorytm_genetyczny>, dostęp 18.01.2021 r.  
   - <https://sound.eti.pg.gda.pl/student/isd/isd03-algorytmy_genetyczne.pdf>, dostęp 18.01.2021 r.  
   - <https://www.geeksforgeeks.org/tournament-selection-ga/>, dostęp 18.01.2021 r.  
   - <https://www.hindawi.com/journals/cin/2017/7430125/>, dostęp 18.01.2021 r.  
   -<https://www.researchgate.net/publication/329643742_Genetic_Algorithm_For_The_Travelling_Salesman_Problem_using_Enhanced_Sequential_Constructive_Crossover_Operator>, dostęp 18.01.2021 r.