Wydział Informatyki Katedra Systemów Informacyjnych i Sieci Komputerowych Pracownia specjalistyczna sztucznej inteligencji	Data: 04.06.2019
Projekt Temat: **Bieszczadzkie szlaki turystyczne – problem komiwojażera** *Mateusz Wawreszuk*	Prowadzący: dr hab. Piotr Hońko Ocena:

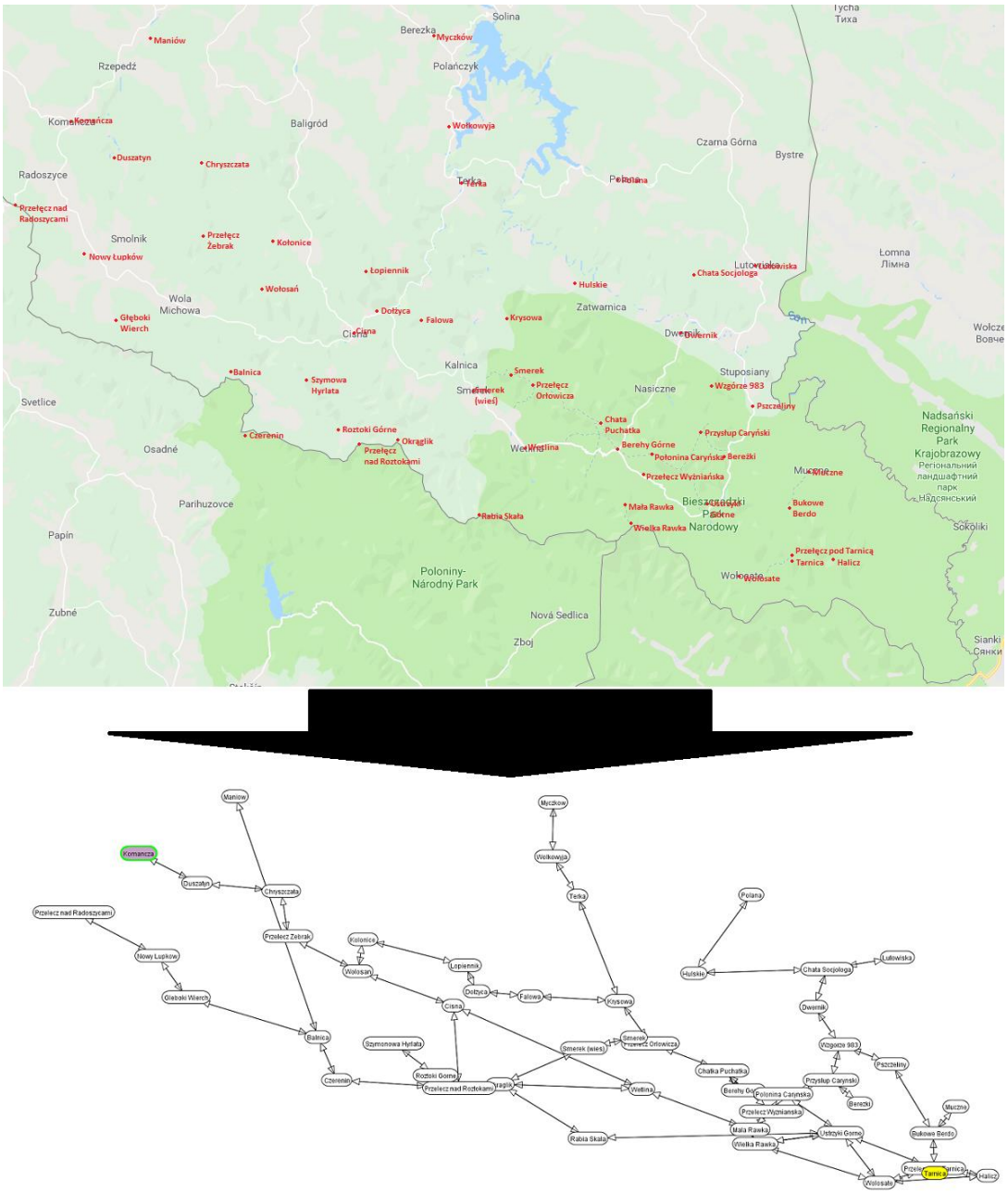
Spis treści:

1. Opis problemu

W ramach projektu zostaną wykorzystane dane stworzone na potrzeby sprawozdania do ćwiczenia 2 „Strategie przeszukiwania”.

Bieszczadzkie szlaki turystyczne – w Bieszczadach jest wyznaczonych kilka szlaków turystycznych. Szlaki przechodzą przez różne punkty – wzgórza, przełęcze, miejscowości, schroniska turystyczne. Odległości między punktami są opisane czasami przejścia w minutach – ze względu na różnicę wysokości czas przejścia z punktu A do punktu B może się różnić od czasu przejścia z punktu B do punktu A. Różne szlaki spotykają się w określonych miejscach, a niektóre się rozwidlają.

Punkty na szlakach reprezentują węzły w grafie, a połączenia między nimi są krawędziami grafu o kosztach określonych czasami przejścia:



W ramach tego projektu spróbuję ustalić najkrótszą możliwą trasę pozwalającą przejść wszystkie punkty na szlakach, wracając do punktu wyjścia (problem komiwojażera). Ze względu na to, że do części z punktów prowadzi tylko jedna krawędź

(np. Maniów, Komańcza, Tarnica), zastosowałem uproszczenie pozwalające na przechodzenie przez niektóre punkty wielokrotnie, co odbiega od typowego problemu komiwojażera.

2. Prezentacja danych

Dane do zadania zapisano w dwóch plikach w łatwej do odczytania programistycznie postaci:

- input_node_names.txt – zawiera dane na temat wierzchołków grafu, składające się kolejno z indeksu, nazwy, pozycji X i pozycji Y. W pliku mają postać:

```
Balnica
0
7254
5249
Berehy Gorne
1
7697
5304
Berezki
2
7818
5318
Bukowe Berdo
3
7894
5349
```

- input_connection_matrix.txt – zawiera dane o krawędziach grafu, składające się z indeksu węzła początkowego, indeksu węzła końcowego oraz wagi krawędzi (czas przejścia między punktami w minutach). Dane te w pliku mają postać:

```
STARTINDEX  ENDINDEX   COST
20 10 120
10 20 120
10 6  150
6  10 120
6  43 60
43 6  75
43 17 120
17 43 105
```

3. Metody zastosowane do rozwiązania

Rozwiązanie problemu podzieliłem na dwie części:

- uzupełnienie macierzy przejść – tak, aby dowolne dwa punkty były połączone krawędzią. W tym celu wykorzystałem heurystyczny algorytm A*, znajdujący najkrótszą ścieżkę pomiędzy każdą parą punktów, które nie są połączone bezpośrednio.
- algorytm genetyczny – znając połączenia między dowolnymi dwoma punktami dalsze wyszukiwanie rozwiązania przeprowadziłem w oparciu o algorytm genetyczny, w którym chromosomem osobnika jest ścieżka zawierająca każdy z wierzchołków grafu (bez powtórzeń). Miarą dostosowania osobnika jest całkowita długość ścieżki z jego chromosomu (im niższa, tym lepsze dostosowanie).

4. Implementacja rozwiązania

Rozwiązanie zaimplementowano w języku Java. Implementacja zawiera dwie klasy z algorytmami – Astar (algorytm A*) oraz GeneticAlgorithm (algorytm genetyczny). Wczytywanie danych i zapis danych do pliku odbywa się w klasach FileReader i FileWriter. Modele danych stanowią klasy Individual (osobnik do algorytmu genetycznego) oraz Node (węzeł grafu). Wszystkie elementy są łączone ze sobą w klasie Main.

4.1. Reprezentacja danych w programie

Po wczytaniu do programu dane są zapisywane do kilku struktur:

- informacje o wierzchołkach – lista obiektów klasy Node
- informacje o krawędziach – dwuwymiarowa tablica int connectionMatrix
- połączenia (ścieżki) między wierzchołkami – dwuwymiarowa tablica list LinkedList, zawierających obiekty klasy Node

W kolejnych podpunktach opisałem te struktury danych wraz z ich zastosowaniem.

4.1.1. Klasa Node

Reprezentuje pojedynczy wierzchołek. Zawiera następujące pola:

```
private int id;
private String name;
private Node cameFrom;
private int distanceFromStart;
private int xPos;
private int yPos;
private Map<Integer, Integer> children;
private int heuristicDistance;
```

pole	opis
id	indeks węzła – wczytywany z pliku
name	nazwa węzła – wczytywana z pliku
cameFrom	wierzchołek, z którego prowadzi krawędź do tego wierzchołka – stosowane w algorytmie A*, po wczytaniu z pliku pole zostaje puste, zostaje wypełnione w trakcie wykonywania algorytmu A*
distanceFromStart	długość ścieżki z punktu początkowego – stosowane w algorytmie A*, po wczytaniu z pliku pole zostaje puste, odległość obliczana jest w trakcie wykonywania algorytmu A*
xPos	współrzędne x węzła – wczytywane z pliku
yPos	współrzędne y węzła – wczytywane z pliku
children	mapa wierzchołków, do których prowadzą krawędzie z tego wierzchołka. Kluczami są indeksy wierzchołków, a wartościami – długości krawędzi (odległości). Po wczytaniu z pliku pole jest pustą mapą, kolejne elementy zostają dodane w miarę analizowania i uzupełniania tablicy connectionMatrix
heuristicDistance	szacowana odległość do punktu docelowego – stosowane w algorytmie A*, po wczytaniu z pliku pole zostaje puste, wartość (jako odległość między dwoma punktami w układzie współrzędnych) obliczana jest w momencie inicjalizacji algorytmu A*

Oprócz powyższych pól, klasa Node zawiera dwa konstruktory (w tym jeden tworzący głęboką kopię) oraz niezbędne gettery i settery.

4.1.2. Tablica connectionMatrix

```
int[][] connectionsMatrix
```

Opisuje krawędzie między wierzchołkami. Wiersze i kolumny tablicy to wierzchołki, a ich przecięcia zawierają wagę krawędzi między tymi dwoma wierzchołkami. Brak krawędzi reprezentowany jest wartością Integer.MAX_VALUE. Wszystkie wartości na głównej przekątnej tablicy są równe zeru. Po wczytaniu danych z pliku, tablica ma postać (myślnikami zastąpiono wartości Integer.MAX_VALUE):

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
0	0	-	-	-	-	-	-	-	60	-	-	-	-	180	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	75	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	0	-	-	-	90	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	90	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	60	-	-	-	-	-	-
3	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	60	-	-	-	-	-	-	-	-	120	-	-	-	120	-	-	-	-	-
4	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	90	-	-	-	-
5	-	75	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	90	-	-	-	60	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	0	-	-	-	120	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	60	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-	-	-	-	165	-	-	-	-	-	-	-	-	-	-	195	-	-	-	-	-	-	-	-
8	45	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	150	-	-	-	-	-	-	-	-	-
9	-	-	-	-	-	-	-	-	-	0	-	75	-	-	-	-	-	-	-	-	-	-	-	-	-	-	105	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	150	-	-	-	0	-	-	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	210	-	-	-	-	-	-	-	-	-	-	-	60	-	-	-	-	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-	45	-	-	0	-	-	-	-	-	-	-	-	-	-	-	180	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
13	135	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	150	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	75	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	150	105	-	-	-	-	-	-	-	-	-	-	-	-	-	-
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	180	-	0	-	-	-	-	120	-	-	-	75	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-
17	-	-	-	-	-	-	165	-	-	-	-	-	-	-	-	-	-	0	-	90	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	105	-	-	-	-	-	-	-
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	195	-	-	-	-	-	-	-	-	-	-	-	-	-	-	150
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	135	-	0	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
20	-	-	-	-	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
21	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	90	-	-	-	-	0	-	-	-	120	-	-	45	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
22	-	-	-	-	-	-	-	-	-	-	-	150	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	15	60	-	-	-	-
23	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	150	-	-	-	-	-	-	-	-	-	-	-	-	-	-	195	-
24	-	-	-	-	-	-	-	-	-	-	-	-	210	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	105	-	-	-	-	-	-	-	-	-	225	-
25	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	75	-	-	-	-	210	-	-	-	0	-	-	-	-	-	-	-	-	-	165	-	-	-	180	-	-	-	315	-	-	-	-
26	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	105	-	-	-	-	-	-	-	-	-	-	-	-	-	-
27	-	-	-	-	-	-	-	-	-	75	-	-	-	-	-	-	-	-	-	75	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
28	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	45	-	-	-	-	-	-	0	120	-	-	-	-	-	-	-	-	-	-	-	60	-	-	-	-	-	-
29	-	-	-	-	-	-	-	105	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	210	0	-	-	-	75	-	-	-	-	-	-	-	-	-	-	-	-	-	-
30	90	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
31	-	-	-	90	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
32	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	150	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
33	-	-	-	-	-	-	-	-	-	-	-	-	-	150	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	150	-	-	-	-	-	-	-	-	-	-
34	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	150	-	-	-	-	-	-	-	-	-	75	-	-	-	-	0	-	-	-	-	75	-	-	-	210	-	-	-	-	-
35	-	-	-	-	-	-	-	-	-	-	-	45	-	-	-	150	-	-	-	-	-	-	-	-	-	-	75	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-
36	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-	-
37	-	60	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-	-	0	-	-	-	-	60	-	60	-	-	-	-	-	-
38	-	-	-	-	-	105	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	90	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	15
39	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	180	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-
40	-	-	-	-	-	-	180	150	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	105	-	-	-	-	0	-	-	-	-	-	75	-	-	-	-
41	-	-	-	135	-	-	-	-	-	-	-	-	-	-	75	-	60	-	-	-	-	-	-	-	-	135	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	15	-	-
42	-	-	-	-	-	75	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	105	-	-	-	-	-	-	90	-	-	-	-	0	-	-	-	-	-	-	-	-
43	-	-	-	-	-	-	75	-	-	-	-	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-
44	-	-	45	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	15	-	-	-	-	-	-	-	-	-	-	-	-	120	-	-	-	-	0	-	-	-	-	-	-	-	-
45	-	-	-	150	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	105	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-
46	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	300	-	-	-	-	-	-	-	210	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-
47	-	-	-	-	135	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	45	-	-	-	-	-	0	-	-	-
48	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	15	-	-	-	-	-	-	0	-	-
49	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	180	240	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-
50	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	120	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	15	-	-	-	-	-	-	-	-	-	-	0

4.1.3. Tablica ścieżek paths

```
LinkedList<Node>[][] paths = new LinkedList[nodes.size()][nodes.size()];
```

Podobnie jak w tablicy connectionMatrix, wiersze i kolumny tablicy to wierzchołki. W miejscu ich przecięć zawarte są ścieżki od jednego wierzchołka do drugiego. Mogą mieć różną postać:

- jeśli wierzchołek początkowy jest taki sam jak docelowy (zerowa krawędź) – ścieżka składa się z jednego elementu, będącego tym wierzchołkiem
- jeśli wierzchołki są połączone bezpośrednią krawędzią – ścieżka składa się z tych dwóch wierzchołków
- w pozostałych przypadkach – jest to najkrótsza ścieżka między danymi wierzchołkami, wyszukana za pomocą algorytmu A*

4.2. Klasa Main

Klasa zawiera statyczne pola będące parametrami algorytmu genetycznego (zostały opisane w podpunkcie poświęconym opisie implementacji tego algorytmu):

```
private final static int POPULATION = 1000;
private final static int MAX_GENERATIONS_COUNT = -1;
private final static int MAX_GENERATIONS_WITH_NO_IMPROVEMENT = 1000;
private final static double CROSS_FACTOR = 0.9;
private final static double MUTATE_FACTOR = 0.01;
```

oraz metodę statyczną

```
public static void main(String[] args)
```

wywoływaną po uruchomieniu programu. W metodzie tej wywoływane są poszczególne składowe programu:

- następuje wczytanie danych

```
FileReader fileReader = new FileReader();
List<Node> nodes = fileReader.getNodes();

int[][] connectionsMatrix = fileReader.getInputMatrix();
```

- na podstawie analizy tablicy przejść connectionMatrix, uzupełniane są mapy children dla każdego z wierzchołków wczytanych z pliku (List<Node> nodes):

```
for (int i = 0; i < nodes.size(); i++) {
    for (int j = 0; j < nodes.size(); j++) {
        if (connectionsMatrix[i][j] != Integer.MAX_VALUE &&
                                       connectionsMatrix[i][j] != 0) {
            nodes.get(i).addChildren(
                nodes.get(j),
                connectionsMatrix[i][j]
            );
        }
    }
}
```

- tworzona jest i uzupełniana tablica ścieżek między punktami (LinkedList<Node>[][] paths). Jeśli wierzchołek startowy jest równy wierzchołkowi końcowemu, do tablicy ścieżek dodawany jest tylko ten wierzchołek. Jeśli są to dwa różne wierzchołki, a odległość między nimi jest mniejsza od Integer.MAX_VALUE (bezpośrednia krawędź istnieje), ścieżkę stanowią te dwa wierzchołki. W pozostałych przypadkach (brakujące krawędzie), macierz przejść jest uzupełniana przez wywoływanie algorytmu A*. Jako parametry przekazywana jest tablica connectionMatrix w aktualnej postaci oraz wierzchołki, dla których krawędź chcemy uzyskać. Po uzyskaniu ścieżki z algorytmu A*, jej długość jest dodawana do tablicy connectionMatrix, uzupełniana jest mapa children węzła początkowego, a znaleziona ścieżka jest dodawana do tablicy paths:

```
LinkedList<Node>[][] paths = new LinkedList[nodes.size()][nodes.size()];
for (int i = 0; i < nodes.size(); i++) {
    for (int j = 0; j < nodes.size(); j++) {
        LinkedList<Node> path = new LinkedList<>();
        if (i == j) {
            path.add(new Node(nodes.get(i)));
        } else if (connectionsMatrix[i][j] < Integer.MAX_VALUE) {
            path.add(new Node(nodes.get(i)));
            path.add(new Node(nodes.get(j)));
        } else {
            path = (LinkedList<Node>) (new Astar(
                nodes,
                nodes.get(i),
                nodes.get(j)
            )).getPath();
            connectionsMatrix[i][j] = path.get(path.size() - 1)
                .getDistanceFromStart();
            nodes.get(i)
                .addChildren(nodes.get(j), connectionsMatrix[i][j]);
        }
        paths[i][j] = path;
    }
}
```

- uzupełniona tablica connectionMatrix oraz lista wierzchołków nodes zapisywane są do plików:

```
FileWriter.writeMatrixToFile(connectionsMatrix);
FileWriter.writeNodeListToFile(nodes);
```

- uruchamiany jest algorytm GeneticAlgorithm, przyjmując jako parametry pola klasy Main oraz uzupełnioną tablicę połączeń connectionsMatrix. Wynik działania algorytmu (osobnik z najlepszym dopasowaniem, czyli najkrótszą ścieżką), jest zapisywany do zmiennej path:

```
GeneticAlgorithm geneticAlgorithm = new GeneticAlgorithm(
        POPULATION,
        MAX_GENERATIONS_COUNT,
        MAX_GENERATIONS_WITH_NO_IMPROVEMENT,
        CROSS_FACTOR,
        MUTATE_FACTOR,
        connectionsMatrix
);
Individual path = geneticAlgorithm.start();
```

- znaleziona ścieżka składa się z połączeń między wierzchołkami niezależnie czy wierzchołki te oryginalnie były połączone pojedynczą krawędzią, czy są połączone ścieżką znalezioną w wyniku działania algorytmu A*. W tym kroku ścieżka jest rozwijana z uwzględnieniem rzeczywistych ścieżek między wierzchołkami, pobranych z tablicy ścieżek paths:

```
List<Node> bestPath = new LinkedList<>();
for (int i = 1; i < path.getChromosome().length; i++) {
    int node1id = path.getChromosome()[i - 1];
    int node2id = path.getChromosome()[i];
    bestPath.addAll(paths[node1id][node2id]);
    bestPath.remove(bestPath.size() - 1);
}
```

- znaleziona ścieżka jest wypisywana na ekran oraz zapisywana do pliku. W trakcie tworzenia Stringa, który ją opisuje, obliczana jest również jej długość:

```
System.out.println("Best path found:");
StringBuilder pathAsString = new StringBuilder();
int pathLength = 0;
Node previousNode = bestPath.get(0);
Node temp;
for (int i = 1; i < bestPath.size(); i++) {
    temp = bestPath.get(i);
    pathAsString.append(temp.getName())
            .append(" -> ");
    pathLength += connectionsMatrix[previousNode.getId()][temp.getId()];
    previousNode = temp;
}
pathLength += connectionsMatrix[bestPath.get(bestPath.size() -
1).getId()][bestPath.get(0).getId()];
pathAsString.append(bestPath.get(0).getName())
        .append("\n")
        .append("Path length = ")
        .append(pathLength / 60)
        .append(" h ")
        .append(pathLength % 60)
        .append(" min");
System.out.println(pathAsString.toString());
FileWriter.writePathToFile(pathAsString.toString());
```

4.3. Wczytywanie danych z plików

Odbywa się w klasie FileReader. Nazwy plików są zawarte w dwóch polach statycznych klasy – MATRIX_FILE („resources\input_connection_matrix.txt”) oraz NODE_NAMES_FILE („resources\input_node_names.txt”). Wczytywanie odbywa się w dwóch metodach, wywoływanych w konstruktorze klasy. Wczytane dane są przechowywane w polach

```
private int[][] inputMatrix;
private List<Node> nodes;
```

z których mogą być pobrane za pomocą getterów. Poniżej opisałem metody wczytujące dane:

- wczytywanie wierzchołków (metoda readNodes()) – następuje otwarcie pliku, lub wyświetlenie informacji o błędzie i zakończenie działania. Pierwsza linia pliku (nagłówek) jest ignorowana (następuje jej wczytanie bez przypisania do żadnej zmiennej). Następnie, dopóki w pliku istnieją kolejne dane, wczytywane są po 4 kolejne linie. Pierwsza z nich stanowi nazwę wierzchołka, druga (po konwersji na integer) – indeks wierzchołka, a dwie kolejne (również po konwersji na integer) – to kolejno współrzędne x i y. Po wczytaniu tych danych, do listy nodes dodawany jest nowoutworzony obiekt klasy Node.

```
private void readNodes() {
    File file = null;
    Scanner reader;
    try {
        file = new File(NODE_NAMES_FILE);
        reader = new Scanner(file);
    }
    catch(IOException e) {
        System.out.println("File " + file.getName() + " couldn't be
opened!");
        return;
    }
    reader.nextLine();  // header
    while (reader.hasNext()) {
        String nodeName = reader.nextLine();
        int nodeId = Integer.valueOf(reader.nextLine());
        int nodeXpos = Integer.valueOf(reader.nextLine());
        int nodeYpos = Integer.valueOf(reader.nextLine());
        nodes.add(new Node(nodeId, nodeName, nodeXpos, nodeYpos));
    }
    reader.close();
}
```

- wczytanie macierzy połączeń (metoda matrixFileReader(matrixSize)) – jako argument przyjmuje wielkość tablicy (ilość wierzchołków) – jej ustalenie następuje na podstawie odczytania rozmiaru listy nodes. Następuje otwarcie pliku, lub wyświetlenie informacji o błędzie i zakończenie działania. W pierwszej kolejności tworzona jest dwuwymiarowa tablica input, a następnie wypełniana zerami (na głównej przekątnej) oraz wartościami Integer.MAX_VALUE (na pozostałych pozycjach). Po tym następuje wczytywanie danych z pliku. Pierwsza linia pliku (nagłówek) jest ignorowana (następuje jej wczytanie bez przypisania do żadnej zmiennej). Następnie, dopóki plik zawiera kolejne dane), wczytywane są po 3 kolejne wartości integer – dwie pierwsze określają miejsce do zapisu w tablicy input (indeksy wierzchołków), a ostatnia to wartość do zapisania (waga krawędzi).

```
private int[][] matrixFileReader(int matrixSize) {
    File file = null;
    Scanner reader;
    try {
        file = new File(MATRIX_FILE);
        reader = new Scanner(file);
    } catch (IOException e) {
        System.out.println("File " + file.getName() + " couldn't be
opened!");
        return new int[0][0];
```

```
    }
    int[][] input = new int[matrixSize][matrixSize];
    for (int i = 0; i < matrixSize; i++) {
        for (int j = 0; j < matrixSize; j++) {
            if (i == j) {
                input[i][j] = 0;
            } else {
                input[i][j] = Integer.MAX_VALUE;
            }
        }
    }
    reader.nextLine();  // header
    while (reader.hasNext()) {
        int from = reader.nextInt();
        int to = reader.nextInt();
        int distance = reader.nextInt();
        input[from][to] = distance;
    }
    reader.close();
    return input;
}
```

4.4. Zapis danych do pliku

Zapis danych do plików następuje w klasie FileWriter. Wszystkie jej pola i metody są statyczne. Nazwy plików do zapisu są określone w polach klasy:

```
private final static String MATRIX_OUTPUT_FILE =
"full_connection_matrix.txt";
private final static String NODE_LIST_FILE = "full_node_list.txt";
private final static String BEST_PATH = "best_path.txt";
```

Do zapisu służą metody writePathToFile, writeNodeListToFile, writeMatrixToFile. Pierwsza jako argument przyjmuje gotowy do zapisu String, pozostałe dwie konwertują dane do pojedynczych Stringów za pomocą metod nodesToString oraz matrixToString. Ze względu na prostotę działania pominę opisywanie tych metod.

4.5. Algorytm A*

Algorytm A* zaimplementowany jest w klasie Astar. Klasa zawiera cztery pola, inicjalizowane w konstruktorze klasy:

```
private List<Node> allNodes;
private Set<Node> nodesToVisit;
private Node start;
private Node goal;
```

pole	opis
allNodes	zawiera listę wszystkich dostępnych wierzchołków – przekazywana jako parametr konstruktora klasy
nodesToVisit	wierzchołki dostępne do odwiedzenia z odwiedzonych do tej pory wierzchołków – zbiór jest modyfikowany w ramach działania algorytmu
start	wierzchołek początkowy – przekazywany jako parametr konstruktora klasy
goal	wierzchołek końcowy – przekazywany jako parametr konstruktora klasy

W konstruktorze, poza zapisaniem argumentów do pól klasy oraz zainicjalizowaniem zbioru nodesToVisit, dla każdego wierzchołka obliczana jest jego odległość heurystyczna od wierzchołka końcowego (metoda calculateHeuristicDistance(goal)):

```
public Astar(List<Node> nodes, Node start, Node goal) {
    allNodes = nodes;
```

```
    this.start = start;
    this.goal = goal;
    nodesToVisit = new HashSet<>();
    for (Node node : allNodes) {
        node.calculateHeuristicDistance(goal);
    }
}
```

Wywołanie algorytmu następuje przez metodę getPath(). Za aktualny wierzchołek (zmienna currentNode) przyjmowany jest wierzchołek początkowy. Jego odległość od startu (distanceFromStart) ustawiana jest na zero. Do listy wierzchołków dostępnych do odwiedzenia (nodesToVisit) dodawane są wierzchołki dostępne z wierzchołka początkowego (pobrane z jego mapy children). Jest to wykonywane w opisanej dalej metodzie addNewToVisitFromNodeNeigbors. Następnie, do momentu osiągnięcia wierzchołka końcowego (lub braku kolejnych wierzchołków do odwiedzenia), wykonywana jest pętla, w której

- usuwany jest aktualny wierzchołek z listy nodesToVisit
- do listy nodesToVisit dodawane są wierzchołki z mapy children tego wierzchołka (metoda addNewToVisitFromNodeNeighbors)
- jako aktualny wierzchołek ustawiany jest wierzchołek o najlepszej wartości, znaleziony w opisanej dalej metodzie findBestNodeToVisit

Po zakończeniu ścieżki wywoływana jest metoda reconstructPath, do której jako argument przekazywany jest aktualny wierzchołek (czyli wierzchołek końcowy). Wynik działania metody reconstructPath jest również wynikiem metody getPath:

```
public List<Node> getPath() {
    Node currentNode = new Node(start);
    currentNode.setDistanceFromStart(0);
    addNewToVisitFromNodeNeighbors(currentNode);
    while (currentNode.getId() != goal.getId() && !nodesToVisit.isEmpty())
    {
        nodesToVisit.remove(currentNode);
        addNewToVisitFromNodeNeighbors(currentNode);
        currentNode = new Node(findBestNodeToVisit());
    }
    return reconstructPath(currentNode);
}
```

Metoda addNewToVisitFromNodeNeighbors – jako parameter przyjmuje wierzchołek, z którego mapy children zostaną pobrane wierzchołki do dodania. Dla każdego elementu mapy z listy wszystkich wierzchołków pobierany jest wierzchołek o odpowiednim indeksie. W jego polu cameFrom zapisywany jest wierzchołek będący argumentem metody (pozwala to na późniejszą rekonstrukcję ścieżki). Jako odległość od początku ścieżki zapisywana jest wartość pobrana z argumentu metody powiększona o odległość między tymi dwoma wierzchołkami. Następnie, uzupełniony w ten sposób wierzchołek, jest przekazywany do metody addNodeToNodesToVisit, w której podejmowana jest decyzja o dodaniu go do listy wierzchołków do odwiedzenia, lub odrzuceniu.

```
private void addNewToVisitFromNodeNeighbors(Node node) {
    for (Map.Entry<Integer, Integer> child : node.getChildren().entrySet())
{
        Node nodeToAdd = new Node(allNodes.get(child.getKey()));
        nodeToAdd.setCameFrom(node);
        nodeToAdd.setDistanceFromStart(node.getDistanceFromStart() +
child.getValue());
        addNodeToNodesToVisit(nodeToAdd);
    }
}
```

Metoda addNodeToNodesToVisit – w pierwszej kolejności metoda sprawdza, czy wierzchołek już znajduje się w zbiorze nodesToVisit. Jeśli zbiór nie zawiera tego wierzchołka, jest do niego dodawany, a metoda kończy działanie. Jeśli zbiór już zawiera ten wierzchołek, jest on odnajdywany w zbiorze i zapisywany do zmiennej nodeAlreadyAdded. W kolejnym kroku

porównywane są odległości obu wierzchołków od początku ścieżki – jeśli nowy wierzchołek ma tę wartość mniejszą, ze zbioru jest usuwany stary wierzchołek, a dodawany nowy. W przeciwnym wypadku dodanie wierzchołka jest pomijane.

```
private void addNodeToNodesToVisit(Node nodeToAdd) {
    if (!nodesToVisit.contains(nodeToAdd)) {
        nodesToVisit.add(nodeToAdd);
    } else {
        Node nodeAlreadyAdded = null;
        for (Node n : nodesToVisit) {
            if (n.getId() == nodeToAdd.getId()) {
                nodeAlreadyAdded = n;
                break;
            }
        }
        assert nodeAlreadyAdded != null;
        if (nodeToAdd.getDistanceFromStart() <
nodeAlreadyAdded.getDistanceFromStart()) {
            nodesToVisit.remove(nodeAlreadyAdded);
            nodesToVisit.add(nodeToAdd);
        }
    }
}
```

Metoda findBestNodeToVisit – metoda iteruje po wszystkich wierzchołkach dostępnych do odwiedzenia (zbiór nodesToVisit), wyszukując wierzchołka o najniższej wartości sumy odległości od początku ścieżki oraz wartości heurystycznej tego wierzchołka. Wierzchołek z najniższą wartością zostaje zwrócony jako najkorzystniejszy do odwiedzenia w tym kroku.

```
private Node findBestNodeToVisit() {
    int minValue = Integer.MAX_VALUE;
    Node bestNodeToVisit = null;
    for (Node node : nodesToVisit) {
        int pathHeuristicValue = node.getDistanceFromStart() +
node.getHeuristicDistance();
        if (pathHeuristicValue < minValue) {
            minValue = pathHeuristicValue;
            bestNodeToVisit = node;
        }
    }
    return bestNodeToVisit;
}
```

Metoda recounstructPath – jako argument przyjmuje wierzchołek końcowy, od którego zaczyna rekonstrukcję. Do momentu osiągnięcia wierzchołka początkowego wykonuje pętlę wstawiającą aktualny wierzchołek na zerowe miejsce na liście path, a następnie przyjmującą za aktualny wierzchołek rodzica (cameFrom) dodanego do listy wierzchołka.

```
private List<Node> reconstructPath(Node goal) {
    List<Node> path = new LinkedList<>();
    Node currentNode = goal;
    while (currentNode.getId() != start.getId()) {
        path.add(0, new Node(currentNode));
        currentNode = currentNode.getCameFrom();
    }
    path.add(0, new Node(currentNode));
    return path;
}
```

4.6. Algorytm genetyczny

Algorytm genetyczny został zaimplementowany w klasie GeneticAlgorithm. Klasa zawiera następujące pola:

```
public static final Random GENERATOR = new Random();

private final int population;
private final int maxGenerationsCount;
private final double crossFactor;
private final double mutateFactor;

private final List<List<Individual>> generations;
private int firstGenerationOfBestFit = 1;
private final Set<Integer> nodes;
private int[][] connectionsMatrix;
private int maxGenerationsWithNoImprovement;

private Individual bestFit;
private int generationsWithNoImprovement = 0;
```

pole	opis
GENERATOR	obiekt klasy Random, służący do generowania liczb pseudolosowych, wykorzystywany w tej klasie oraz klasie Individual
population	liczba osobników w generacji, przekazywane przez konstruktor klasy
maxGenerationsCount	liczba generacji, po której osiągnięciu algorytm zakończy działanie, przekazywane przez konstruktor klasy
crossFactor	współczynnik krzyżowania, przekazywane przez konstruktor klasy
mutateFactor	współczynnik mutacji, przekazywane przez konstruktor klasy
generations	lista generacji – każda generacja to lista osobników
firstGenerationOfBestFit	pierwsza generacja, w której wystąpiło najlepsze, do tej pory, dopasowanie
nodes	zbiór wierzchołków (same indeksy)
connectionsMatrix	tablica krawędzi, przekazywane przez konstruktor klasy
maxGenerationsWithNoImprovement	liczba generacji bez poprawy najlepszego dopasowania, po której osiągnięciu algorytm zakończy działanie, przekazywane przez konstruktor klasy
bestFit	najlepsze, do tej pory, dopasowanie
generationsWithNoImprovement	aktualna liczba generacji, w których nie nastąpiła poprawa najlepszego dopasowania

W konstruktorze klasy, poza inicjalizacją zmiennych, tworzony jest zbiór wierzchołków – są do niego dodawane kolejne liczby typu integer, od zera do maksymalnego indeksu wierzchołka (ustalonego na podstawie rozmiaru tablicy connectionsMatrix).

```
public GeneticAlgorithm(
        int population,
        int maxGenerationsCount,
        int maxGenerationsWithNoImprovement,
        double crossFactor,
        double mutateFactor,
        int[][] connectionsMatrix
) {
    this.population = population;
    this.maxGenerationsCount = maxGenerationsCount;
    this.crossFactor = crossFactor;
    this.mutateFactor = mutateFactor;
    this.generations = new ArrayList<>();
    this.connectionsMatrix = connectionsMatrix;
    this.maxGenerationsWithNoImprovement = maxGenerationsWithNoImprovement;
    nodes = new HashSet<>();
    for (int i = 0; i < connectionsMatrix.length; i++) {
        nodes.add(i);
    }
}
```

Uruchomienie algorytmu odbywa się przez metodę start. Wygenerowanie pierwszej generacji odbywa się w metodzie setFirstGeneration. Metoda ta dodaje nową generację do listy generations oraz zwraca osobnika z najlepszym dostosowaniem w utworzonej generacji. Algorytm jest wykonywany w pętli while do momentu osiągnięcia liczby generacji ustawionej w polu maxGenerationsCount (przy wartości -1 parametr ten nie jest brany pod uwagę) lub jeśli w liczbie generacji ustawionej w polu maxGenerationsWithNoImprovement nie wystąpi osobnik z lepszym dostosowaniem niż najlepsze osiągnięte do tej pory (podobnie jak w przypadku maxGenerationsCount – przy wartości -1 ten parametr nie jest brany pod uwagę). W pętli wywoływana jest metoda makeNewGeneration, która podobnie jak setFirstGeneration zwraca najlepiej dostosowanego osobnika. Osobnik ten jest porównywany z dotychczasowym najlepszym wynikiem, zapisanym w bestFit. Jeśli jego dostosowanie jest lepsze od dotychczas najlepszego, jest przypisywany do zmiennej bestFit, a zmienna generationsWithNoImprovement jest zerowana. Jeśli w nowej generacji nie wystąpił osobnik z lepszym dostosowaniem niż dotychczasowe najlepsze dostosowanie – współczynnik generationsWithNoImprovement jest inkrementowany. Po zakończeniu pętli wyświetlany jest chromosom najlepiej dostosowanego osobnika, wraz z wartością dostosowania (długością ścieżki) oraz informacją w którym pokoleniu osobnik wystąpił.

```
public Individual start() {
    bestFit = setFirstGeneration();
    printGeneration(1, bestFit);
    while (
            (generations.size() < maxGenerationsCount ||
maxGenerationsCount == -1)
                    && (generationsWithNoImprovement <=
maxGenerationsWithNoImprovement ||
                    maxGenerationsWithNoImprovement == -1)
    ) {
        Individual bestFitInGeneration = makeNewGeneration();
        printGeneration(generations.size(), bestFitInGeneration);
        if (bestFitInGeneration.compareTo(bestFit) > 0) {
            generationsWithNoImprovement = 0;
            bestFit = bestFitInGeneration;
            firstGenerationOfBestFit = generations.size();
        } else {
            generationsWithNoImprovement++;
        }
    }
    System.out.println("---------------------------------------");
    System.out.println("> BEST FIT FOUND IN GENERATION " +
firstGenerationOfBestFit);
    System.out.println("> " + bestFit);
    System.out.println("> FITNESS = " + bestFit.getFitness() / 60 + " h " +
bestFit.getFitness() % 60);
    System.out.println("---------------------------------------\n");
    return bestFit;
}
```

Metoda setFirstGeneration – tworzy liczbę osobników równą współczynnikowi population, a następnie dodaje tak utworzoną listę do listy generations. Nowe osobniki są tworzone przez konstruktor klasy Individual (nowe osobniki z losowymi chromosomami). W trakcie tworzenia kolejnych osobników wyszukiwany jest osobnik z najlepszym współczynnikiem dostosowania, który potem jest zwracany jako wynik działania metody.

```
private Individual setFirstGeneration() {
    Set<Individual> firstGeneration = new HashSet<>();
    Individual bestFitInGeneration = null;
    for (int i = 0; i < population; i++) {
        Individual individual = new Individual(
            mutateFactor, nodes.size(), connectionsMatrix
        );
        if (bestFitInGeneration == null
                || individual.compareTo(bestFitInGeneration) > 0) {
            bestFitInGeneration = individual;
        }
```

```
            firstGeneration.add(individual);
        }
        generations.add(new ArrayList<>(firstGeneration));
        return bestFitInGeneration;
    }
```

Metoda makeNewGeneration – metoda tworzy nową generację bazując na ostatniej generacji z listy generations. Działanie metody rozpoczyna się od zainicjalizowania listy newGeneration, do której będą dodawani nowi osobnicy, pojedynczego osobnika bestFitInGeneration, w którym będzie przechowywany osobnik z najlepszym dostosowaniem oraz pobrania ostatniej generacji z listy generations. Po pobraniu generacji, jest ona sortowana opisaną później metodą sortGeneration.

```
private Individual makeNewGeneration() {
    List<Individual> newGeneration = new ArrayList<>();
    Individual bestFitInGeneration = null;
    List<Individual> sortedLastGeneration =
            sortGeneration(generations.size() - 1);
```

Następnie, na podstawie współczynników dostosowania każdego osobnika, wyliczane są szansy z jakimi osobniki powinny być losowane do nowej generacji. Ze względu na małe różnice między osobnikami (różnice rzędu kilku tysięcy przy wartościach dostosowania rzędu kilkuset tysięcy), współczynniki są zmniejszane o najniższy współczynnik dostosowania w tej populacji. Na początek wszystkie wartości dostosowania są sumowane. Następnie pobierany jest najniższy współczynnik dostosowania w tej populacji, a suma jest pomniejszana o ten współczynnik pomnożony przez wielkość populacji. Potem, w pętli for, dla każdego osobnika obliczany jest jego udział w sumie współczynników dostosowania (przeskalowane o 100000 – standardowe współczynniki procentowe nie sprawdziłyby się przy bardzo dużych populacjach). Ze względu na to, że im mniejsze dostosowanie, tym szansa na wylosowanie osobnika powinna być większa, otrzymana wartość jest odejmowana od 100000.

```
int generationFitnessSum = sortedLastGeneration.stream()
        .map(Individual::getFitness)
        .reduce(Integer::sum)
        .orElse(0);

int oldGenerationMinFitness = sortedLastGeneration
        .get(sortedLastGeneration.size() - 1)
        .getFitness();
generationFitnessSum -= (oldGenerationMinFitness * population);
int[] oldGenerationIndividualsFitness = new int[population];
int partialSum = 0;
for (int i = 0; i < population; i++) {
    if (generationFitnessSum > 0) {
        int fitness = 100000 - (
            100000 * (
                        sortedLastGeneration.get(i).getFitness()
                        - oldGenerationMinFitness
                    ) / generationFitnessSum
            );
        oldGenerationIndividualsFitness[i] = partialSum + fitness;
        partialSum += fitness;
    } else {
        oldGenerationIndividualsFitness[i] = 100000 * (i + 1) / population;
    }
}
```

W kolejnym kroku, dla każdego osobnika z nowej populacji losowana jest liczba z przedziału <0, 100000000>. Na podstawie wylosowanej liczby wybierany jest osobnik ze starej populacji i dodawany do nowej. Następnie dla każdego osobnika o nieparzystym indeksie losowana jest liczba typu double (z przedziału <0, 1>) i porównywana ze współczynnikiem krzyżowania crossFactor. Jeśli wylosowana liczba jest mniejsza, następuje krzyżowanie osobnika o aktualnym indeksie z osobnikiem z indeksem o jeden niższym. Krzyżowanie odbywa się w metodzie cross z klasy Individual. Na każdym z potomków powstałych w wyniku krzyżowania wywoływana jest metoda mutate klasy Individual mutująca pojedyncze geny. W trakcie tworzenia

nowych osobników wyszukiwany jest osobnik z najlepszym dostosowaniem, który zostanie zwrócony jako wynik działania metody. Nowo utworzona generacja jest dodawana do listy generations.

```
for (int i = 0; i < population; i++) {
    int random = GENERATOR.nextInt(100000000);
    int newIndividualNumber = 0;
    for (int j = 0; j < population; j++) {
        if (random < oldGenerationIndividualsFitness[j]) {
            break;
        } else {
            newIndividualNumber = j;
        }
    }
    newGeneration.add(sortedLastGeneration.get(newIndividualNumber));
    if (i % 2 == 1 && GENERATOR.nextDouble() < crossFactor) {
        Individual child1 = newGeneration.get(i - 1)
                .cross(newGeneration.get(i)).mutate();
        Individual child2 = newGeneration.get(i)
                .cross(newGeneration.get(i - 1)).mutate();
        newGeneration.set(i - 1, child1);
        newGeneration.set(i, child2);
    }
    if (bestFitInGeneration == null ||
newGeneration.get(i).compareTo(bestFitInGeneration) > 0) {
        bestFitInGeneration = newGeneration.get(i);
    }
}
generations.add(new ArrayList<>(newGeneration));
generations.add(generations.size() - 2, new ArrayList<>());
return bestFitInGeneration;
```

Metoda sortGeneration – metoda jako parameter przyjmuje indeks generacji, która ma zostać posortowana. Generacja ta jest pobierana z listy generations. Sortowanie odbywa się za pomocą metody sort klasy List, jako operator porównania przyjmując metodę compareTo klasy Individual. Posortowana lista osobników jest zwracana jako wynik działania metody.

```
private List<Individual> sortGeneration(int generationNumber) {
    List<Individual> list = new ArrayList<>(
            generations.get(generationNumber)
    );
    list.sort(Individual::compareTo);
    return list;
}
```

4.6.1. Osobnik – klasa Individual

Obiekty klasy Individual reprezentują pojedyncze osobniki w algorytmie genetycznym. Klasa zawiera następujące pola:

```
private final double mutateFactor;
private final int[][] connectionMatrix;

private int[] chromosome;
private int fitness;
```

pole	opis
mutateFactor	współczynnik mutacji, wykorzystywany po zakończeniu krzyżowania osobników, przekazywany przez konstruktor klasy
connectionMatrix	tablica krawędzi, wykorzystywana do obliczania współczynnika dopasowania osobnika, przekazywana przez konstruktor klasy
chromosome	chromosom osobnika, powstaje losowo przy tworzeniu nowego osobnika lub w wyniku krzyżowania dwóch osobników

fitness	współczynnik dostosowania osobnika będący sumą odległości między kolejnymi wierzchołkami wchodzącymi w skład chromosomu (ostatni wierzchołek łączy się z pierwszym). Jest obliczany w metodzie calculateFitness wywoływanej przy tworzeniu nowego osobnika.

Klasa Individual zawiera dwa konstruktory – pierwszy jest publiczny i jest wykorzystywany do tworzenia nowego, losowego osobnika. Chromosom jest tworzony przez losowanie kolejnych indeksów wierzchołków z utworzonego wcześniej zbioru dostępnych wierzchołków, dodawaniu ich do chromosomu oraz usuwaniu ze zbioru. Czynność jest powtarzana do zapełnienia chromosomu indeksami wierzchołków. Dzięki takiemu podejściu, składowe chromosomu nie powtarzają się.

```
public Individual(double mutateFactor, int genesCount, int[][]
connectionMatrix) {
    List<Integer> genesToUse = new ArrayList<>();
    for (int i = 0; i < genesCount; i++) {
        genesToUse.add(i);
    }
    chromosome = new int[genesCount];
    for (int i = 0; i < genesCount; i++) {
        int index = GENERATOR.nextInt(genesToUse.size());
        chromosome[i] = genesToUse.get(index);
        genesToUse.remove(index);
    }
    this.mutateFactor = mutateFactor;
    this.connectionMatrix = connectionMatrix;
    this.fitness = calculateFitness(chromosome);
}
```

Drugi konstruktor jest prywatny i służy do tworzenia nowego osobnika w wyniku krzyżowania – wymaga przekazania w parametrze gotowego chromosomu.

```
private Individual(double mutateFactor, int[][] connectionMatrix, int[]
chromosome) {
    this.chromosome = chromosome;
    this.mutateFactor = mutateFactor;
    this.connectionMatrix = connectionMatrix;
    this.fitness = calculateFitness(chromosome);
}
```

4.6.1.1. Krzyżowanie osobników

Do krzyżowania wykorzystywana jest metoda cross osobnika Individual, przyjmująca jako parametr (pair) drugiego osobnika. Do krzyżowania wykorzystano Operator EX (Edge Crossover) w wersji Edge-3 (Whitley, 2000). Krzyżowanie rozpoczyna się od zainicjalizowania struktur danych: childChromosome (zwykła tablica integer), do którego będzie zapisywany utworzony chromosom oraz lista krawędzi edges, w której indeks to numer wierzchołka, będąca listą map o kluczu integer (wierzchołek do którego prowadzi krawędź w chromosomie jednego z rodziców) oraz wartości boolean (false – krawędź występuje tylko u jednego z rodziców, true – krawędź występuje u obojga rodziców).

```
int[] childChromosome = new int[pair.chromosome.length];

List<Map<Integer, Boolean>> edges = new ArrayList<>();
for (int i = 0; i < chromosome.length; i++) {
    edges.add(new HashMap<>());
}
```

Następnie lista edges jest wypełniana krawędziami występującymi w chromosomach rodziców. W pierwszej kolejności brane są krawędzie wychodzące z wierzchołka o indeksie zero:

```
int parent1gene = this.chromosome[0];
int parent1neighbour1 = this.chromosome[chromosome.length - 1];
int parent1neighbour2 = this.chromosome[1];
mapEdges(edges, parent1gene, parent1neighbour1, parent1neighbour2);
int parent2gene = pair.chromosome[0];
```

```
int parent2neighbour1 = pair.chromosome[chromosome.length - 1];
int parent2neighbour2 = pair.chromosome[1];
mapEdges(edges, parent2gene, parent2neighbour1, parent2neighbour2);
```

Dodawanie krawędzi odbywa się w metodzie mapEdges. Jako parametry przekazywana jest lista, wierzchołek dla którego ma się odbyć mapowanie, oraz wierzchołki z nim sąsiadujące. Metoda sprawdza czy mapa pod odpowiednim indeksem zawiera już daną krawędź. Jeśli zawiera – krawędź jest usuwana, a następnie dodawana ponownie, tym razem z wartością true. Jeśli krawędź do tej pory nie została dodana – jest dodawana z wartością false.

```
private void mapEdges(
        List<Map<Integer, Boolean>> edges,
        int parentGene,
        int parentGeneNeighbour1,
        int parentGeneNeighbour2
) {
    if (edges.get(parentGene).containsKey(parentGeneNeighbour1)) {
        edges.get(parentGene).remove(parentGeneNeighbour1);
        edges.get(parentGene).put(parentGeneNeighbour1, true);
    } else {
        edges.get(parentGene).put(parentGeneNeighbour1, false);
    }
    if (edges.get(parentGene).containsKey(parentGeneNeighbour2)) {
        edges.get(parentGene).remove(parentGeneNeighbour2);
        edges.get(parentGene).put(parentGeneNeighbour2, true);
    } else {
        edges.get(parentGene).put(parentGeneNeighbour2, false);
    }
}
```

Następnie dodawane są krawędzie dla wierzchołków z pozycji od 1 do przedostatniej chromosomów każdego rodziców:

```
for (int i = 1; i < chromosome.length - 1; i++) {
    parent1gene = this.chromosome[i];
    parent1neighbour1 = this.chromosome[i - 1];
    parent1neighbour2 = this.chromosome[i + 1];
    mapEdges(edges, parent1gene, parent1neighbour1, parent1neighbour2);
    parent2gene = pair.chromosome[i];
    parent2neighbour1 = pair.chromosome[i - 1];
    parent2neighbour2 = pair.chromosome[i + 1];
    mapEdges(edges, parent2gene, parent2neighbour1, parent2neighbour2);
}
```

Na koniec dodawane są krawędzie dla ostatnich wierzchołków w chromosomach rodziców:

```
parent1gene = this.chromosome[chromosome.length - 1];
parent1neighbour1 = this.chromosome[chromosome.length - 2];
parent1neighbour2 = this.chromosome[0];
mapEdges(edges, parent1gene, parent1neighbour1, parent1neighbour2);
parent2gene = pair.chromosome[chromosome.length - 1];
parent2neighbour1 = pair.chromosome[chromosome.length - 2];
parent2neighbour2 = pair.chromosome[0];
mapEdges(edges, parent2gene, parent2neighbour1, parent2neighbour2);
```

Po zmapowaniu krawędzi, tworzony jest zbiór zawierający możliwe do wykorzystania wierzchołki, mający zapewnić brak powtarzalności dodawanych do chromosomu genów:

```
Set<Integer> genesToUse = new HashSet<>();
for (int i = 0; i < chromosome.length; i++) {
    genesToUse.add(i);
}
```

W kolejnym kroku, dla zerowego indeksu nowego chromosomu, wybierany jest losowy gen. Po jego wylosowaniu z listy edges, usuwane są wszystkie prowadzące do niego krawędzie:

```
int index = 0;
int v = getRandomGeneFromSet(genesToUse);
childChromosome[index] = v;
for (Map<Integer, Boolean> integerBooleanMap : edges) {
    integerBooleanMap.remove(v);
}
```

Losowanie genów odbywa się w metodzie setRandomGeneFromSet. Metoda ta jako parametr przyjmuje zbiór możliwych do wykorzystania genów. Po wylosowaniu jednego z genów z tego zbioru, gen jest usuwany ze zbioru i zwracany jako wynik metody:

```
private int getRandomGeneFromSet(Set<Integer> genesToUse) {
    int randomIndex = GENERATOR.nextInt(genesToUse.size());
    int i = 0;
    for (Integer gene : genesToUse) {
        if (i == randomIndex) {
            genesToUse.remove(gene);
            return gene;
        }
        i++;
    }
    return -1;
}
```

Dalsza część krzyżowania odbywa się w pętli while działającej dopóki w zbiorze genesToUse znajdują się jeszcze jakieś geny. Jako kolejny gen do chromosomu dodawany jest jeden z wierzchołków pobranych z listy krawędzi edges. Jeśli jedna z krawędzi występuje u obojga rodziców, wybierany jest wierzchołek leżący na jej końcu. Jeśli nie ma takich krawędzi, wybierany jest wierzchołek, który ma najmniejszy zbiór jeszcze niewykorzystanych krawędzi (ale nie zerowy). Jeśli w dwóch pierwszych krokach nie zostanie wybrany kolejny wierzchołek, jest on wybierany losowo. Po wybraniu kolejnego wierzchołka, prowadzące do niego krawędzie są usuwane ze zbioru edges, sam wierzchołek jest usuwany ze zbioru genów do wykorzystania, a pętla while wykonuje się ponownie.

```
while (!genesToUse.isEmpty()) {
    index++;
    if (index == childChromosome.length) {
        break;
    }
    int nextEdge = -1;
    for (Map.Entry<Integer, Boolean> edge : edges.get(v).entrySet()) {
        if (edge.getValue()) {
            nextEdge = edge.getKey();
        }
    }
    if (nextEdge < 0) {
        int minEdges = Integer.MAX_VALUE;
        for (Map.Entry<Integer, Boolean> edge : edges.get(v).entrySet()) {
            if (edges.get(edge.getKey()).size() < minEdges
                    && !edges.get(edge.getKey()).isEmpty()) {
                minEdges = edges.get(edge.getKey()).size();
                nextEdge = edge.getKey();
            }
        }
    }
    if (nextEdge < 0) {
        nextEdge = getRandomGeneFromSet(genesToUse);
    }
    v = nextEdge;
```

```
    for (Map<Integer, Boolean> integerBooleanMap : edges) {
        integerBooleanMap.remove(v);
    }
    genesToUse.remove(v);
    childChromosome[index] = v;
}
```

Po uzupełnieniu wszystkich pozycji chromosomu nowego osobnika, tworzony jest nowy obiekt Individual, zwracany jako wynik działania metody.

```
return new Individual(mutateFactor, connectionMatrix, childChromosome);
```

4.6.1.2. Mutacja chromosomu

Mutacja odbywa się w metodzie mutate klasy Individual. Dla każdego genu w chromosomie losowana jest liczba typu double. Liczba ta jest porównywana ze współczynnikiem mutacji mutateFactor, jeśli jest niższa następuje mutacja, polegająca na zamianie danego genu z genem o indeksie niższym o 1. Po zakończeniu mutowania, obliczany jest współczynnik dopasowania dla nowego chromosomu, a osobnik ze zmienionym chromosomem jest zwracany jako wynik działania metody mutate:

```
public Individual mutate() {
    for (int i = 1; i < chromosome.length; i++) {
        if (GENERATOR.nextDouble() < mutateFactor) {
            int temp = chromosome[i];
            chromosome[i] = chromosome[i - 1];
            chromosome[i - 1] = temp;
        }
    }
    fitness = calculateFitness(chromosome);
    return this;
}
```

4.6.1.3. Współczynnik dopasowania i porównywanie osobników

Obliczanie współczynnika dopasowania odbywa się w metodzie calculateFitness klasy Individual. Metoda jest wywołana po utworzeniu nowego osobnika – zarówno z losowym chromosomem jak i w wyniku krzyżowania. Dodatkowo jest wywoływana na koniec mutowania chromosomu w metodzie mutate. Polega na przejściu po każdym genie z chromosomu osobnika i zsumowaniu wszystkich odległości przejść między wierzchołkami zawartymi w chromosomie, wraz z odległością od wierzchołka ostatniego do wierzchołka z indeksu 0.

```
private int calculateFitness(int[] chromosome) {
    int fitness = 0;
    for (int i = 1; i < chromosome.length; i++) {
        fitness += connectionMatrix[chromosome[i - 1]][chromosome[i]];
    }
    fitness += connectionMatrix[
                    chromosome[chromosome.length - 1]
            ][
                    chromosome[0]
            ];
    return fitness;
}
```

Klasa Individual implementuje interfejs Comparable, co pozwala na proste porównywanie osobników.

```
public class Individual implements Comparable<Individual>
```

Aby było to możliwe, w klasie nadpisano metodę compareTo interfejsu Comparable. Metoda zwraca różnicę współczynników dopasowania osobnika będącego parametrem oraz osobnika, na którym metoda została wywołana. Jeśli porównujemy dwa osobniki – A i B, za pomocą wywołania A.compareTo(B):

- osobnik A jest lepiej dopasowany (ma niższy współczynnik fitness) – zwracana jest wartość dodatnia
- osobnik B jest lepiej dopasowany – zwracana jest wartość ujemna
- oba osobniki mają taki sam współczynnik fitness – zwracane jest zero (osobnik są sobie równe)

```
@Override
public int compareTo(Individual o) {
    return o.fitness - this.fitness;
}
```

5. Prezentacja wyników

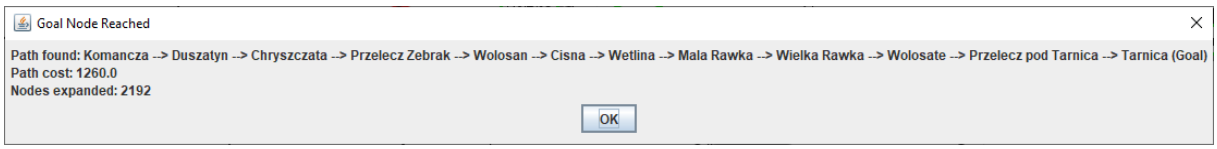
5.1. Wypełnienie macierzy połączeń

Poniżej widok macierzy połączeń po uzupełnieniu brakujących krawędzi przy użyciu algorytmu A*:

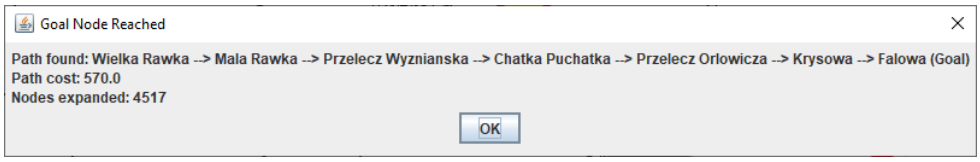
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
0	0	810	855	990	420	735	690	390	60	795	810	975	870	180	915	1185	735	510	465	600	930	645	825	1125	720	765	1110	720	600	390	75	1050	1275	300	315	1035	1290	750	630	450	210	855	660	615	810	885	525	285	870	945	615
1	720	0	195	375	735	90	780	480	675	525	900	315	480	900	465	525	285	600	315	690	1020	300	165	675	270	210	450	630	255	375	795	435	825	1020	510	375	630	90	180	1170	525	390	150	705	150	225	525	600	405	495	195
2	810	240	0	285	825	315	870	570	765	750	990	225	705	990	480	435	375	690	540	780	1110	390	75	900	495	300	360	855	345	465	1035	345	1050	1110	540	285	540	180	405	1260	615	405	240	795	60	135	615	690	420	720	420
3	810	420	285	0	825	480	870	570	765	915	990	375	870	990	195	585	180	690	690	780	1110	300	225	1065	660	255	510	900	345	465	885	60	1215	1110	540	435	690	360	570	1260	615	120	405	795	240	120	570	690	135	885	585
4	330	735	780	915	0	660	615	315	285	720	735	900	795	510	840	1110	660	435	390	525	855	570	750	1050	645	690	1035	645	525	315	405	975	1200	630	240	960	1215	675	555	780	135	780	585	540	735	810	450	90	795	870	540
5	630	75	255	435	645	0	690	390	585	435	810	375	390	810	480	585	345	510	225	600	930	210	240	585	180	270	510	540	165	285	870	495	735	930	420	435	690	150	90	1080	435	420	60	615	210	285	570	510	435	405	105
6	735	930	975	1110	750	855	0	345	690	465	120	1095	540	915	1035	1305	855	180	735	270	240	765	945	1125	720	885	1230	390	720	510	810	1170	1275	1035	645	1155	1410	870	825	1185	540	975	780	60	930	1005	795	615	990	945	840
7	390	585	630	765	405	510	300	0	345	405	420	750	480	570	690	960	510	120	390	210	540	420	600	1050	645	540	885	330	375	165	465	825	1200	690	300	810	1065	525	555	840	195	630	435	225	585	660	450	270	645	870	540
8	45	750	795	930	360	675	630	330	0	735	750	915	810	225	855	1125	675	450	405	540	870	585	765	1065	660	705	1050	660	540	330	120	990	1215	345	255	975	1230	690	570	495	150	795	600	555	750	825	465	225	810	885	555
9	870	540	720	900	885	465	495	480	825	0	615	840	75	1050	945	1050	810	315	495	180	735	675	705	660	255	735	975	105	630	645	945	960	810	1170	690	900	1155	615	360	1320	675	885	525	420	675	750	900	750	900	480	375
10	885	1080	1125	1260	900	1005	150	495	840	615	0	1245	690	1065	1185	1455	1005	330	885	420	120	915	1095	1275	870	1035	1380	540	870	660	960	1320	1425	1185	795	1305	1560	1020	975	1335	690	1125	930	210	1080	1155	945	765	1140	1095	990
11	975	405	270	420	990	480	1035	840	930	915	1155	0	870	1155	615	210	540	855	705	945	1275	555	210	1065	660	465	135	1020	510	630	1200	480	1215	1275	705	60	315	345	570	1425	780	540	405	960	225	270	780	855	555	885	585
12	885	465	645	825	900	390	540	525	840	45	660	765	0	1065	870	975	735	360	420	225	780	600	630	585	180	660	900	150	555	675	990	885	735	1185	615	825	1080	540	285	1335	690	810	450	465	600	675	825	765	825	405	300
13	135	945	990	1125	555	870	825	525	195	930	945	1110	1005	0	1050	1320	870	645	600	735	1065	780	960	1260	855	900	1245	855	735	525	210	1185	1410	120	450	1170	1425	885	765	270	345	990	795	750	945	1020	660	420	1005	1080	750
14	765	435	480	210	780	435	825	525	720	870	945	585	825	945	0	795	150	645	645	735	1065	255	435	1020	615	210	720	855	300	420	990	270	1170	1065	495	645	900	375	525	1215	570	75	360	750	435	330	525	645	90	840	540
15	1170	600	465	615	1185	675	1230	1035	1125	1110	1350	195	1065	1350	810	0	735	1050	900	1140	1470	750	405	1260	855	660	225	1215	705	825	1395	675	1410	1470	900	150	105	540	765	1620	975	735	600	1155	420	465	975	1050	750	1080	780
16	630	300	345	255	645	300	690	390	585	735	810	465	690	810	180	675	0	510	510	600	930	120	315	885	480	75	600	720	165	285	855	315	1035	930	360	525	780	240	390	1080	435	120	225	615	300	375	390	510	135	705	405
17	555	750	795	930	570	675	180	165	510	285	300	915	360	735	855	1125	675	0	555	90	420	585	765	945	540	705	1050	210	540	330	630	990	1095	855	465	975	1230	690	645	1005	360	795	600	105	750	825	615	435	810	765	660
18	465	345	525	705	480	270	675	450	420	510	795	645	465	645	750	855	615	570	0	585	915	480	510	660	255	540	780	615	435	270	540	765	810	765	195	705	960	420	165	915	270	690	330	600	480	555	405	345	705	480	150
19	690	735	915	1065	705	660	315	300	645	195	435	1035	270	870	990	1245	810	135	690	0	555	720	900	855	450	840	1170	120	675	465	765	1125	1005	990	600	1095	1350	810	555	1140	495	930	720	240	870	945	750	570	945	675	570
20	1005	1200	1245	1380	1020	1125	270	615	960	735	120	1365	810	1185	1305	1575	1125	450	1005	540	0	1035	1215	1395	990	1155	1500	660	990	780	1080	1440	1545	1305	915	1425	1680	1140	1095	1455	810	1245	1050	330	1200	1275	1065	885	1260	1215	1110
21	510	255	300	345	525	180	570	270	465	615	690	420	570	690	270	630	90	390	390	480	810	0	270	765	360	120	555	600	45	165	735	405	915	810	240	480	735	195	270	960	315	210	105	495	255	330	435	390	225	585	285
22	765	195	60	210	780	270	825	630	720	705	945	150	660	945	405	360	330	645	495	735	1065	345	0	855	450	255	285	810	300	420	990	270	1005	1065	495	210	465	135	360	1215	570	330	195	750	15	60	570	645	345	675	375
23	1140	720	900	1080	1155	645	1185	1035	1095	690	1305	1020	645	1320	1125	1230	990	1005	675	870	1425	855	885	0	435	915	1155	795	810	930	1215	1140	150	1440	870	1080	1335	795	540	1590	945	1065	705	1110	855	930	1080	1020	1080	195	555
24	705	285	465	645	720	210	750	600	660	255	870	585	210	885	690	795	555	570	240	435	990	420	450	405	0	480	720	360	375	495	780	705	555	1005	435	645	900	360	105	1155	510	630	270	675	420	495	645	585	645	225	120
25	705	225	270	315	720	300	765	465	660	735	885	390	690	885	255	600	75	585	525	675	1005	210	240	885	480	0	525	795	240	360	930	375	1035	1005	435	450	705	165	390	1155	510	180	225	690	225	300	315	585	195	705	405
26	1125	555	420	570	1140	630	1185	990	1080	1065	1305	150	1020	1305	765	255	690	1005	855	1095	1425	705	360	1215	810	615	0	1170	660	780	1350	630	1365	1425	855	105	360	495	720	1575	930	690	555	1110	375	420	930	1005	705	1035	735
27	765	615	795	975	780	540	390	375	720	75	510	915	150	945	1020	1125	885	210	570	75	630	750	780	735	330	810	1050	0	705	540	840	1035	885	1065	675	975	1230	690	435	1215	570	960	600	315	750	825	825	645	975	555	450
28	465	210	255	390	480	135	525	225	420	570	645	375	525	645	315	585	135	345	345	435	765	45	225	720	315	165	510	555	0	120	690	450	870	765	195	435	690	150	225	915	270	255	60	450	210	285	405	345	270	540	240
29	345	420	465	600	360	345	405	105	300	510	525	585	585	525	525	795	345	225	225	315	645	255	435	885	480	375	720	435	210	0	570	660	1035	645	75	645	900	360	390	795	150	465	270	330	420	495	285	225	480	705	375
30	90	900	945	1080	510	825	780	480	150	885	900	1065	960	270	1005	1275	825	600	555	690	1020	735	915	1215	810	855	1200	810	690	480	0	1140	1365	390	405	1125	1380	840	720	540	300	945	750	705	900	975	615	375	960	1035	705
31	900	510	375	90	915	570	960	660	855	1005	1080	465	960	1080	285	675	270	780	780	870	1200	390	315	1155	750	345	600	990	435	555	975	0	1305	1200	630	525	780	450	660	1350	705	210	495	885	330	210	660	780	225	975	675
32	1290	870	1050	1230	1305	795	1335	1185	1245	840	1455	1170	795	1470	1275	1380	1095	1155	825	1020	1575	1005	1035	150	585	1065	1305	945	960	1080	1365	1290	0	1590	1020	1230	1485	945	690	1740	1095	1215	855	1260	1005	1080	1230	1170	1230	345	705
33	285	1095	1140	1275	705	1020	975	675	345	1080	1095	1260	1155	150	1200	1470	1020	795	750	885	1215	930	1110	1410	1005	1050	1395	1005	885	675	360	1335	1560	0	600	1320	1575	1035	915	150	495	1140	945	900	1095	1170	810	570	1155	1230	900
34	270	495	540	675	285	420	480	255	225	585	600	660	615	450	600	870	420	375	150	390	720	330	510	810	405	450	795	510	285	75	345	735	960	570	0	720	975	435	315	720	75	540	345	405	495	570	210	150	555	630	300
35	1020	450	315	465	1035	525	1080	885	975	960	1200	45	915	1200	660	150	585	900	750	990	1320	600	255	1110	705	510	75	1065	555	675	1095	525	1260	1320	750	0	255	390	615	1470	825	585	450	1005	270	315	825	900	600	930	630
36	1290	720	585	735	1305	795	1350	1155	1245	1230	1470	315	1185	1470	930	120	855	1170	1020	1260	1590	870	525	1380	975	780	345	1335	825	945	1365	795	1530	1590	1020	270	0	660	885	1740	1095	855	720	1275	540	585	1095	1170	870	1200	900
37	630	60	105	285	645	135	690	390	585	570	810	225	525	810	375	435	195	510	360	600	930	210	75	720	315	120	360	675	165	285	705	345	870	930	360	285	540	0	225	1080	435	300	60	615	60	135	435	510	315	540	240
38	600	180	360	540	615	105	795	495	555	345	915	480	300	780	585	690	405	615	135	525	1035	315	330	495	90	375	615	450	270	390	675	600	645	900	330	540	795	255	0	1050	405	525	165	720	315	390	540	480	540	315	15
39	465	1275	1320	1455	885	1200	1155	855	525	1260	1275	1440	1335	330	1380	1650	1200	975	930	1065	1395	1110	1290	1590	1185	1230	1575	1185	1065	855	540	1515	1740	180	780	1500	1755	1215	1095	0	675	1320	1125	1080	1275	1350	990	750	1335	1410	1080
40	195	600	645	780	210	525	480	180	150	585	600	765	660	375	705	975	525	300	255	390	720	435	615	915	510	555	900	510	390	180	270	840	1065	495	105	825	1080	540	420	645	0	645	450	405	600	675	315	75	660	735	405
41	690	360	405	135	705	360	750	450	645	795	870	510	750	870	75	720	60	570	570	660	990	180	360	945	540	135	645	780	225	345	765	195	1095	990	420	570	825	300	450	1140	495	0	285	675	360	255	450	570	15	765	465
42	570	150	195	375	585	75	630	330	525	510	750	315	465	750	420	525	240	450	300	540	870	150	165	660	255	210	450	615	105	225	645	435	810	870	300	375	630	90	165	1020	375	360	0	555	150	225	510	450	375	480	180
43	675	870	915	1050	690	795	75	285	630	405	195	1035	480	855	975	1245	795	120	675	210	315	705	885	1065	660	825	1170	330	660	450	750	1110	1215	975	585	1095	1350	810	765	1125	480	915	720	0	870	945	735	555	930	885	780
44	750	180	45	225	765	255	810	510	705	690	930	165	645	930	420	375	315	630	480	720	1050	330	15	840	435	240	300	795	285	405	825	285	990	1050	480	225	480	120	345	1200	555	345	180	735	0	75	555	630	360	660	360
45	870	300	165	150	885	375	930	630	825	810	1050	255	765	1050	345	465	330	750	600	840	1170	450	105	960	555	360	390	915	405	525	945	210	1110	1170	600	315	570	240	465	1320	675	270	300	855	120	0	675	750	285	780	480
46	480	525	570	615	495	600	690	465	435	795	810	690	825	660	555	900	375	585	360	600	930	495	540	1020	615	300	825	720	495	285	555	675	1170	780	210	750	1005	465	525	930	285	480	525	615	525	600	0	360	495	840	510
47	240	645	690	825	135	570	525	225	195	630	645	810	705	420	750	1020	570	345	300	435	765	480	660	960	555	600	945	555	435	225	315	885	1110	540	150	870	1125	585	465	690	45	690	495	450	645	720	360	0	705	780	450
48	705	375	420	150	720	375	765	465	660	810	885	525	765	885	90	735	75	585	585	675	1005	195	375	960	555	150	660	795	240	360	780	210	1110	1005	435	585	840	315	465	1155	510	15	300	690	375	270	465	585	0	780	480
49	945	525	705	885	960	450	990	840	900	495	1110	825	450	1125	930	1035	750	810	480	675	1230	660	675	180	240	720	960	600	615	735	1020	945	330	1245	675	885	1140	600	345	1395	750	870	510	915	660	735	885	825	885	0	360
50	585	195	375	555	600	120	795	510	540	360	915	495	315	765	600	705	420	630	120	540	1035	330	345	510	105	390	630	465	285	390	660	615	660	885	315	555	810	270	15	1035	390	540	180	720	330	405	525	465	555	330	0

W celu potwierdzenia poprawności, przeprowadzono kilka wyszukiwań ścieżek za pomocą programu Search, wykorzystywanego w ćwiczeniu 2 „Strategie przeszukiwania”. We wszystkich przypadkach wyszukiwania dały takie same wyniki:

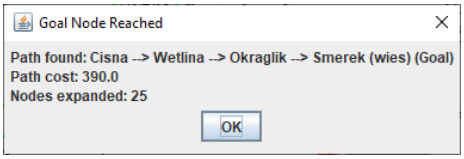
- koszt ścieżki z wierzchołka 20 (Komańcza) do wierzchołka 48 (Tarnica): 1260



- koszt ścieżki z wierzchołka 21 (Wielka Rawka) do wierzchołka 12 (Falowa): 570

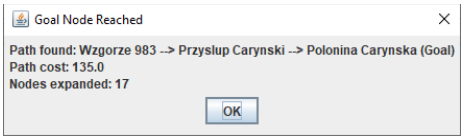


- koszt ścieżki z wierzchołka 7 (Cisna) do wierzchołka 18 (Smerek (Wieś)): 390



- koszt ścieżki z wierzchołka 41 (Przełęcz pod Tarnicą) do wierzchołka 5 (Chatka Puchatka): 360

- koszt ścieżki z wierzchołka 22 (Wzgórze 983) do wierzchołka 37 (Połonina Caryńska): 135



5.2. Wyszukanie najkrótszej ścieżki

Przeprowadziłem wyszukiwania najkrótszej ścieżki dla różnych ustawień parametrów algorytmu genetycznego. W poniższej tabeli zebrałem część wyników wykonania algorytmu, we wszystkich przypadkach algorytm działał do momentu osiągnięcia 1000000 generacji bez poprawienia najlepszego dostosowania. Wyjątkiem było pierwsze wykonanie algorytmu (liczność populacji 1000) – zanim warunek został osiągnięty, została wykorzystana cała pamięć dostępna wirtualnej maszynie Java.

liczność populacji	współczynnik krzyżowania	współczynnik mutacji	długość odnalezionej ścieżki	generacja, w której odnaleziono najkrótszą ścieżkę
1000	0.9	0.01	412 godz. 45 min.	93327
100	0.9	0.01	349 godz. 45 min.	18639
10	0.9	0.01	329 godz.	733669
10	0.95	0.01	328 godz. 15 min.	2207591
10	1	0.01	341 godz. 15 min.	1998691
10	**0.95**	**0**	**527 godz. 15 min.**	**3**
10	0.95	0.1	192 godz. 30 min.	724779
10	0.95	0.2	234 godz. 30 min.	941269
10	0.95	0.15	214 godz. 45 min.	2603421
10	**0.99**	**0.1**	**175 godz. 45 min.**	**1593837**
100	0.99	0.1	224 godz.	245169
100	0.95	0.1	232 godz. 15 min.	514153
10	**0.8**	**0.1**	**175 godz. 15 min.**	**4102077**
10	0.8	0.2	220 godz. 30 min.	2162039
10	0.5	0.2	203 godz. 45 min.	2412433
10	0.5	0.1	227 godz. 15 min.	2607645

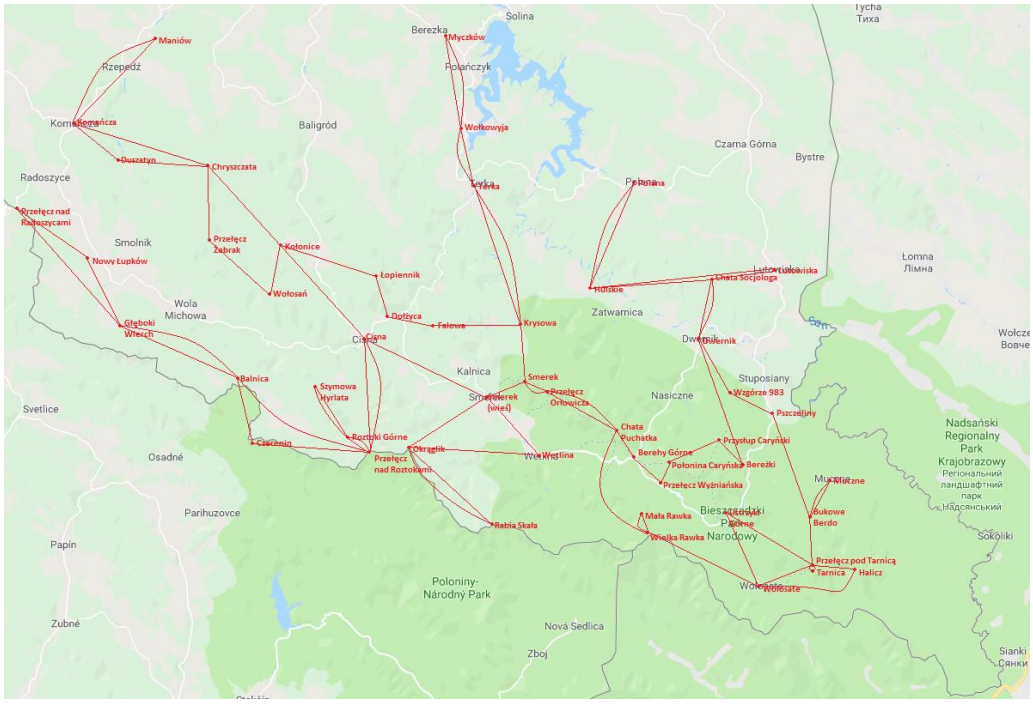
W tabeli pogrubiłem wyniki trzech wykonań algorytmu – najgorszego oraz najlepszych (najlepszy wynik padł przy dwóch różnych ustawieniach parametrów). Ze względu na dużą losowość algorytmu genetycznego, dla zaznaczonych parametrów przeprowadzono po kilka dodatkowych wykonań algorytmu (tym razem liczba generacji bez poprawienia najlepszego dostosowania została ustawiona na wartość 5 razy wyższą – 5000000), wyniki zebrano w poniższej tabeli:

Parametry (populacja / współczynnik krzyżowania / współczynnik mutacji)	**wykonanie**	**długość odnalezionej ścieżki**	**generacja, w której odnaleziono najkrótszą ścieżkę**
10 / 0.95 / 0	1	527 godz. 15 min.	3
	2	525 godz. 30 min.	1
	3	504 godz. 30 min.	3
	4	524 godz. 30 min.	3
	5	534 godz. 30 min.	1
10 / 0.99 / 0.1	1	175 godz. 45 min.	1593837
	2	179 godz. 45 min.	2108251
	3	171 godz. 15 min.	1223579
	4	177 godz. 15 min.	388935
	5	172 godz.	4024807
10 / 0.8 / 0.1	1	175 godz. 45 min.	4102077
	2	183 godz. 45 min.	1164207
	3	205 godz. 15 min.	440157
	4	174 godz. 30 min.	2878027
	5	171 godz. 15 min.	2233897
	6	156 godz.	6441329

Najkrótsza znaleziona ścieżka (ostatnia w tabeli):

Przysłup Caryński -> Bereżki -> Dwernik -> Chata Socjologa -> Lutowiska -> Chata Socjologa -> Hulskie -> Polana -> Hulskie -> Chata Socjologa -> Dwernik -> Wzgórze 983 -> Pszczeliny -> Bukowe Berdo -> Muczne -> Bukowe Berdo -> Przełęcz pod Tarnicą -> Tarnica -> Przełęcz pod Tarnicą -> Halicz -> Wołosate -> Przełęcz pod Tarnicą -> Ustrzyki Górne -> Wołosate -> Wielka Rawka -> Mała Rawka -> Wielka Rawka -> Chata Puchatka -> Przełęcz Orłowicza -> Smerek -> Krysowa -> Terka -> Wołkowyja -> Myczków -> Wołkowyja -> Terka -> Krysowa -> Falowa -> Dołżyca -> Łopiennik -> Kołonice -> Wołosań -> Przełęcz Żebrak -> Chryszczata -> Duszatyn -> Komańcza -> Maniów -> Komańcza -> Chryszczata -> Kołonice -> Cisna -> Przełęcz nad Roztokami -> Roztoki Górne -> Szymowa Hyrlata -> Roztoki Górne -> Przełęcz nad Roztokami -> Czerenin -> Balnica -> Głęboki Wierch -> Przełęcz nad Radoszycami -> Nowy Łupków -> Głęboki Wierch -> Balnica -> Przełęcz nad Roztokami -> Cisna -> Smerek (wieś) -> Smerek -> Przełęcz Orłowicza -> Chata Puchatka -> Berehy Górne -> Przełęcz Wyżniańska -> Połonina Caryńska -> Przysłup Caryński

Widok najkrótszej ścieżki na mapie:



6. Wnioski

6.1. Algorytm A*

Porównania wyników wykonania mojej implementacji algorytmu A* z wynikami z programu Search pozwala stwierdzić, że implementacja została przeprowadzona prawidłowo. Dużym zaskoczeniem było dla mnie to, że wyniki zostały odnalezione tak szybko – 2431 wyszukiwań zajęło ułamek sekundy. Wyszukiwania połączeń między niektórymi z tych wierzchołków w programie Search nie zostały ukończone po kilkunastu godzinach.

6.2. Algorytm genetyczny

Implementując algorytm brałem pod uwagę wykorzystywanie jak najliczniejszych pokoleń w trakcie jego wykonywania. Pierwsze uruchomienia algorytmu prowadziłem z ustawieniami 1000 – 10000 osobników w pokoleniu. Ograniczenie liczności generacji do 100, a potem do 10, pozwoliło znacznie poprawić wyniki. Niestety nie jestem w stanie stwierdzić czy wynika to z błędów implementacji, czy z przyczyn niezależnych od tego.

Kolejne testy pokazały, że większy wpływ na jakość wyniku ma odpowiedni dobór współczynnika mutacji niż współczynnika krzyżowania. Algorytm dawał podobne wyniki przy ustawieniach współczynnika krzyżowania 0.8 jak i 0.99. W przypadku współczynnika mutacji najbardziej optymalnym ustawieniem okazało się 0.1 – zarówno mniejsza, jak i większa wartość powodowała pogorszenie jakości znajdowanych rozwiązań.

Dzięki przeprowadzonym próbom można stwierdzić, że działanie algorytmu nie jest całkowicie losowe (czego się obawiałem w trakcie jego implementacji) – gdyby tak było, zwiększanie współczynnika mutacji nie powodowałoby takiego pogarszania wyników.

Można też zauważyć, że im dłuższe wykonywanie algorytmu, tym lepszej jakości rozwiązania są odnajdywane. Wyjątkiem były uruchomienia z ustawieniem współczynnika mutacji na 0 – rozwiązania odnalezione w jednej z pierwszych generacji pozostawały najlepszymi.

6.3. Najlepsze odnalezione rozwiązanie

W trakcie nanoszenia ścieżki na mapę moją uwagę zwrócił jeden jej fragment:

Przysłup Caryński -> Bereżki -> Dwernik -> Chata Socjologa -> Lutowiska -> Chata Socjologa -> Hulskie -> Polana -> Hulskie -> Chata Socjologa -> Dwernik -> Wzgórze 983 -> Pszczeliny -> Bukowe Berdo -> Muczne -> Bukowe Berdo -> Przełęcz pod Tarnicą -> Tarnica -> Przełęcz pod Tarnicą -> Halicz -> **Wołosate -> Przełęcz pod Tarnicą -> Ustrzyki Górne -> Wołosate** -> Wielka Rawka -> Mała Rawka -> Wielka Rawka -> Chata Puchatka -> Przełęcz Orłowicza -> Smerek -> Krysowa -> Terka -> Wołkowyja -> Myczków -> Wołkowyja -> Terka -> Krysowa -> Falowa -> Dołżyca -> Łopiennik -> Kołonice -> Wołosań -> Przełęcz Żebrak -> Chryszczata -> Duszatyn -> Komańcza -> Maniów -> Komańcza -> Chryszczata -> Kołonice -> Cisna -> Przełęcz nad Roztokami -> Roztoki Górne -> Szymowa Hyrlata -> Roztoki Górne -> Przełęcz nad Roztokami -> Czerenin -> Balnica -> Głęboki Wierch -> Przełęcz nad Radoszycami -> Nowy Łupków -> Głęboki Wierch -> Balnica -> Przełęcz nad Roztokami -> Cisna -> Smerek (wieś) -> Smerek -> Przełęcz Orłowicza -> Chata Puchatka -> Berehy Górne -> Przełęcz Wyżniańska -> Połonina Caryńska -> Przysłup Caryński

Przełęcz pod Tarnicą była już odwiedzona, a ze względu na znajomość tej okolicy wydało mi się, że szybszym rozwiązaniem byłoby przejście bezpośrednio z Wołosatego do Ustrzyk Górnych i z powrotem. Po sprawdzeniu kosztów przejść między tymi punktami (według danych wejściowych):

Wołosate – Przełęcz pod Tarnicą: 120 min.

Przełęcz pod Tarnicą – Ustrzyki Górne: 135 min.

Wołosate – Ustrzyki Górne: 75 min.

Ustrzyki Górne – Wołosate: 75 min.

Spostrzeżenie okazało się prawidłowe – pominięcie powrotu na Przełęcz pod Tarnicą pozwoliłoby skrócić ścieżkę o dodatkowe 3 godziny. Nie jest jednak wykluczone, że dalsze działanie algorytmu pozwoliłoby mu znaleźć również takie rozwiązanie.

Odnalezienie przedstawionej wyżej ścieżki zajęło kilkanaście minut – opracowanie takiej trasy bez użycia rozwiązań sztucznej inteligencji zajęłoby znacznie dłużej (o ile w ogóle byłoby możliwe).

7. Lista załączonych plików

'**bieszczadzkie_szlaki_turystyczne.xml**' – dane wejściowe, plik w formacie do programu Search (bez zaznaczonego wierzchołka początkowego i końcowego – wszystkie wierzchołki mają typ „REGULAR")

'**dane_wejściowe.xlsx**' – dane z pliku '**bieszczadzkie_szlaki_turystyczne.xml**' przeniesione do arkusza kalkulacyjnego

'**uzupełniona_tablica.xlsx**' – tablica przejść po uzupełnieniu brakujących połączeń przy wykorzystaniu algorytmu A*

folder '**program**' – zawiera kod źródłowy stworzonego programu

- główna klasa programu znajduje się w pliku '**program/src/main/java/ai_project/Main.java**' – edytując go można zmienić ustawienia algorytmu genetycznego
- dane wejściowe znajdują się w plikach '**input_connection_matrix.txt**' oraz '**input_node_names.txt**' w katalogu '**program/resources**'
- kompilacja i uruchomienie programu jest możliwe poprzez uruchomienie skryptu '**START.bat**' z głównego katalogu programu
- dane wyjściowe są zapisywane do plików w głównym katalogu programu