

# Bieszczadzki Tour

Specyfikacja implementacyjna

Maciej Czarkowski, 18.11.2019

## Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
<b>2</b>	<b>Środowisko deweloperskie</b>	<b>2</b>
<b>3</b>	<b>Zasady wersjonowania</b>	<b>3</b>
3.1	Współpraca z systemem kontroli wersji . . . . .	3
3.2	Schemat wiadomości . . . . .	3
<b>4</b>	<b>Diagram klas i struktura programu</b>	<b>4</b>
4.1	DataReader . . . . .	4
4.2	Map . . . . .	4
4.3	Place . . . . .	4
4.4	MainAlgorithm . . . . .	5
4.5	DijkstraAlgorithm . . . . .	5
4.6	DataWriter . . . . .	5
<b>5</b>	<b>Rozwiązany problem</b>	<b>5</b>
<b>6</b>	<b>Wykorzystane algorytmy</b>	<b>6</b>
<b>7</b>	<b>Istotne struktury danych</b>	<b>7</b>

## 1 Wstęp

Niniejszy dokument, będący specyfikacją implementacyjną projektu „*Bieszczadzki Tour*”, ma za zadanie możliwie najlepiej przybliżyć, osobom odpowiedzialnym za jego implementację, sposoby oraz metody prowadzące do stworzenia wydajnego i poprawnie działającego kodu. Program ma rozwiązywać problem odnalezienia optymalnej ścieżki pomiędzy zestawem zadanych punktów, w taki sposób, aby trasa była najkrótsza oraz możliwie najtańsza. Zgodnie z informacjami zawartymi w specyfikacji funkcjonalnej projektu, program do działania wykorzystuje pliki wejściowe, których konfiguracja powinna być zgodna z wyżej wymienionym dokumentem. Pożądanym efektem działania programu jest plik wynikowy, informujący użytkownika, którą trasą się udać, aby droga była optymalna.

## 2 Środowisko deweloperskie

Implementacja programu będzie odbywała się na komputerze *Dell Vostro 3578*, z 4-rdzeniowym procesorem *Intel Core i5-8250U*, korzystającym z systemu *Windows 10 Pro* w wersji 64-bitowej *10.0.18362*. Program zaimplementowany będzie w języku *Java* w wersji 8. Implementacja będzie odbywała się w środowisku programistycznym *IntelliJ IDEA 2018.3 (Community Edition) Build #IC-183.4284.148*, wydanym 21 listopada 2018 roku, z wykorzystaniem narzędzi deweloperskich z pakietu *OpenJDK 64-Bit Server VM by JetBrains s.r.o Windows 10 10.0*. Środowiskiem uruchomieniowym dla kodu będzie maszyna wirtualna *Javy* w wersji *1.8.0\_152-release-1343-b15 amd64*.

## 3 Zasady wersjonowania

### 3.1 Współpraca z systemem kontroli wersji

Projekt będzie przechowywany na zdalnym repozytorium, przygotowanym do realizacji projektu indywidualnego z Algorytmów i Struktur Danych. Kolejne funkcjonalności będą realizowane na osobnych gałęziach, po czym, po ich pełnym wykonaniu, będą scalane z gałęzią *master* repozytorium.

### 3.2 Schemat wiadomości

Każda aktualizacja zawartości repozytorium (*commit*), która będzie przekazywana na odpowiednią gałąź w repozytorium, będzie opatrzona odpowiednią wiadomością, zgodną ze schematem dotyczącym pierwszego słowa w wiadomości, które identyfikowało będzie poczynioną w kodzie modyfikację:

- „add” w przypadku dodania nowego elementu do kodu;
- „delete” w przypadku usunięcia określonego fragmentu kodu;
- „fix” w przypadku naprawiania niedziałających segmentów kodu;
- „modify” w przypadku drobnych modyfikacji w kodzie;
- „refactor” w przypadku znaczących zmian w kodzie, dotyczących większego fragmentu kodu;
- „approve v. x.x.”, gdzie x.x. to wartości od 0.1 do 1.0 (wersji ostatecznej), w przypadku zatwierdzenia kolejnej wersji do wydania — dodania na gałąź *master*.

## 4 Diagram klas i struktura programu

Niniejszy rozdział opisuje strukturę programu, wyróżniając kolejne wykorzystywane klasy.

### 4.1 DataReader

Jest to klasa odpowiedzialna za walidację danych wejściowych do programu. Analizuje ona otrzymane argumenty, a jeśli którykolwiek z nich nie jest zgodny z przyjętym formatem, informuje użytkownika o błędzie. Zczytane z pliku wejściowego miejsca zostaną umieszczone na liście, po czym po zakończeniu jej wypełniania utworzona zostanie tablica, która przechowywała będzie czasy przejść pomiędzy miejscami. Dzięki metodzie *fillTimesVector* ID miejsc są konwertowane na unikalną wartość liczbową, która będzie identyfikowała dane miejsce w programie.

### 4.2 Map

Jest to klasa reprezentująca strukturę mapy punktów, które rozważamy przy tworzeniu optymalnej ścieżki. Mapa punktów przedstawiona jest w postaci tablicy dwuwymiarowej przechowującej czasy przejść pomiędzy miejscami na mapie.

### 4.3 Place

Klasa ta zawiera model pojedynczego elementu mapy. Atrybutami tej klasy są ID miejsca, pełna jego nazwa oraz numeryczne, unikalne ID, niezbędne, aby można było identyfikować punkty w tablicy czasów przejść.

## 4.4 MainAlgorithm

Jest to klasa realizująca główną logikę programu. W przypadku pojawienia się na wejściu pliku `wishlist` klasa ta przekazuje mapę miejsc do modyfikacji metodom z klasy *DijkstraAlgorithm*. Metoda *calculateWholePath* zwraca tablicę numerycznych ID miejsc w kolejności, w której powinny one zostać odwiedzone.

## 4.5 DijkstraAlgorithm

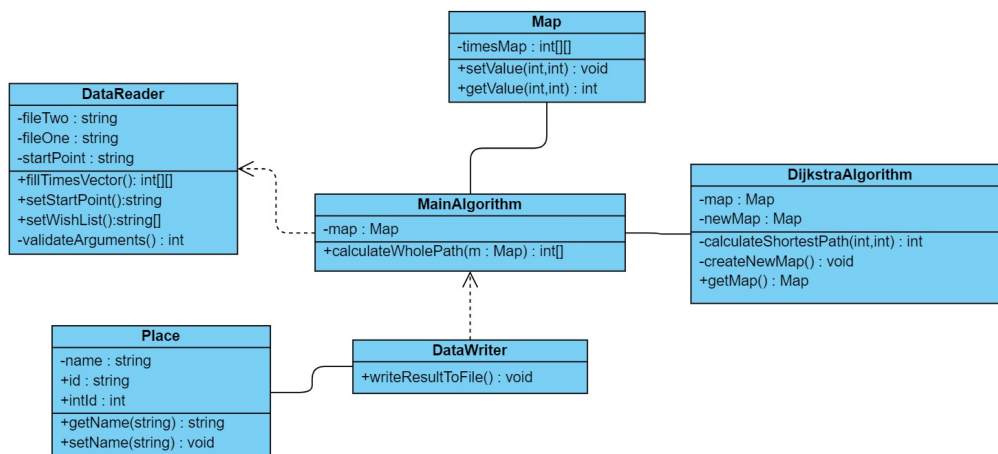
Klasa tworząca, na podstawie otrzymanej mapy punktów i czasów przejść pomiędzy nimi, nową mapę, uwzględniającą jedynie miejsca, które pojawiły się w pliku `wishlist`.

## 4.6 DataWriter

Klasa wypisująca otrzymany rezultat działania algorytmu do pliku wynikowego `result.txt`. Na podstawie otrzymanego od klasy *MainAlgorithm* wektora, klasa odtwarza nazwy miejsc i wypisuje je do pliku wyjściowego.

# 5 Rozwiązywany problem

Rozwiązywanym przez program problemem jest zagadnienie optymalizacyjne polegające na znalezieniu cyklu zamkniętego w grafie ważonym o możliwie najmniejszym koszcie, nazywane *Problemem komiwojażera*. Jest to algorytm NP-trudny, charakteryzujący się dużą złożonością obliczeniową przy kierowaniu się metodą brute force — przy sprawdzaniu wszystkich możliwych połączeń pomiędzy wszystkimi węzłami otrzymujemy złożoność obliczeniową rzędu  $n!$ , co już przy niewielu węzłach (+20) powoduje praktyczny brak



Rysunek 1: Diagram klas programu

możliwość realizacji programu poprzez niezwykle długi czas wykonania. W tej sytuacji stosownym rozwiązaniem jest zastosowanie algorytmów przybliżających optymalną trasę, jednakże działających w możliwym do zaakceptowania czasie.

## 6 Wykorzystane algorytmy

Program opiera się na wykorzystaniu algorytmu Dijkstry oraz algorytmu najbliższego sąsiada. Pierwszy z wymienionych algorytmów wykorzystany zostanie przy tworzeniu pomniejszonej mapy punktów, które chcemy odwiedzić. Odnajduje on najkrótszą możliwą ścieżkę pomiędzy dwoma określonymi węzłami w grafie. Algorytm najbliższego sąsiada pozwoli odnaleźć estymowaną optymalną trasę pomiędzy wybranymi punktami, bądź pomiędzy wszystkimi, jeśli plik `wishlist` nie został podany jako argument. Jest to algorytm zachłanny, wybierający jako ścieżki przejścia pomiędzy węzłami ścieżki o możliwie najmniejszej wadze, jednakże biorąc pod uwagę, że rozwią-

zywany problem jest NP-trudny, taki algorytm zagwarantuje wystarczająco przybliżone rozwiązania do optymalnego — wyznaczone ścieżki powinny być maksymalnie 25% gorsze od optymalnej.

## 7 Istotne struktury danych

Główną strukturą danych, którą będzie wykorzystywał program jest dwuwymiarowa tablica wartości całkowitych, identyfikująca czasy przejść w minutach pomiędzy poszczególnymi miejscami. Wykorzystana zostanie również lista liniowa, która pozwoli na wejściu programu zebrać wszystkie dane, które następnie zostaną wpisane odpowiednio do tablicy.