Specyfikacja implementacyjna programu *BieszczadyTrip*

Krzysztof Maciejewski

12 sierpnia 2020

Spis treści

1 Wstęp

Dokument ten jest technicznym opisem programu *BieszczadyTrip*. Przybliża on proces tworzenia programu oraz przedstawia zarys stosowanych algorytmów i struktur danych.

2 Środowisko deweloperskie

W tej sekcji przedstawione są elementy środowiska pracy, w którym będzie powstanie program *BieszczadyTrip*.

Parametry komputera:

- Procesor: Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz
- Pamięć RAM: 16 GB
- System operacyjny: Windows 10.0.18362

Pozostałe parametry:

- Język programowania: Java (wersja: "13" 2019-09-17)
- Maszyna wirtualna: Java HotSpot(TM) 64-Bit Server VM (build 13+33, mixed mode, sharing)
- Środowisko programistyczne: Intelij IDEA 2019.2.4 (Community Edition)

3 Zasady wersjonowania

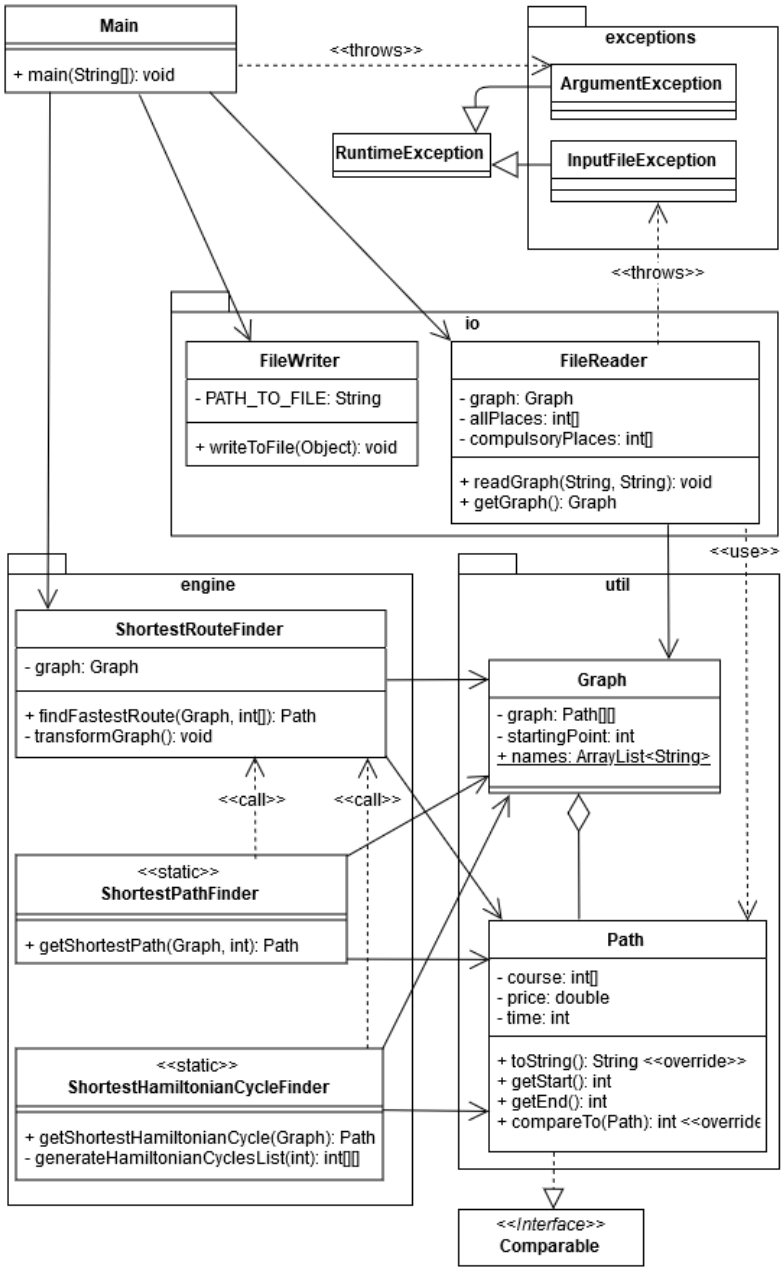
Z uwagi na niewielki rozmiar programu i indywidualną realizację, wszystkie jego wersje będą wrzucane do gałęzi *master*. Każda wrzucana do gałęzi głównej wersja musi się kompilować i poprawnie realizować dodane funkcjonalności, co będzie potwierdzone testami.

Konwencja nazewnicza wersji:

1. Wersje deweloperskie programu, przed uzyskaniem pierwszej wersji stabilnej, będą w formacie: `0.X - dev`, gdzie `X` to numer wersji deweloperskiej.
2. Pierwsza wersja stabilna będzie oznaczona: `1.0`.
3. Poprawki to stabilnej wersji będą oznaczane: `1.Y`, gdzie `Y` to numer poprawionej wersji.

Wiadomości wysyłane do repozytorium nie mają sztywnego szablonu. Powinny one określać co dodano, co zmieniono i co usunięto.

4 Diagram klas



Rysunek 1: Diagram Klas.

- Klasa *Graph* odpowiada za przechowywanie grafu.
- Klasa *Path* odpowiada za przechowywanie ścieżki.
- Klasa *FileWriter* wypisuje metodę *toString()* obiektu do pliku. Wykorzystywana jest do wypisaniq obiektu *Path* przechowującego końcową trasę.
- Klasa *FileReader* czyta dane z pliku wejściowego i tworzy listę wszystkich miejsc oraz tych obowiązkowych. Na podstawie tych list oraz czasów przejścia między miejscami tworzy graf, który możemy potem pobrać przy pomocy metody *getGraph()*.
- Wyjątek *ArguementException* rozszerza *RuntimeException* i będzie rzucany z odpowiednim komunikatem jeżeli program znajdzie błąd w argumentach wejściowych.
- Wyjątek *InputFileException* rozszerza *RuntimeException* i będzie rzucany z odpowiednim komunikatem jeżeli program znajdzie błąd w pliku wejściowym.
- Klasa *ShortestRouteFinder* znajduje najszybszą trasę na podstawie grafu i listy obowiązkowych miejsc.
- Statyczna klasa *ShortestPathFinder* wykorzystuje algorytm Dijkstry w celu znalezienia najkrótszej ścieżki w grafie między dwoma wierzchołkami.
- Statyczna klasa *ShortestHamiltonianCycleFinder* wykorzystuje algorytm Heap'a w celu znalezienia wszystkich cykli Hamiltona w przekształconym algorytmem Dijkstry grafie. Następnie znajduje najkrótszy z nich.

5 Istotne struktury

5.1 Path

Klasa *Path* posiada trzy atrybuty określające ścieżkę. Są to: czas trasy, koszt trasy, przebieg trasy. Przebieg trasy przechowywany jest w tablicy typu `int` i określa kolejne miejsca podróży używając ich numerycznego identyfikatora określonego w obiekcie *Graph*. Klasa *Path* używana jest do przechowywania wszystkich ścieżek — i tych między wierzchołkami grafu, i najkrótszego cyklu będącego wynikiem działania programu. Klasa implementuje interfejs *Comparable*, dzięki czemu jesteśmy w stanie porównać cykle Hamiltona w celu znalezienia tego najkrótszego.

5.2 Graph

Klasa *Graph* zawiera trzy atrybuty. Lista *names* przechowuje miejsca w kolejności wczytania. Każde miejsce ma przypisany numer będący numerem indeksu danego miejsca na tej liście. Graf przechowywany jest w atrybucie *graph*

w postaci dwuwymiarowej tablicy. Numer indeksu w tablicy odpowiada konkretnemu miejscu. Pierwszy indeks określa miejsce początkowe, drugi indeks określa miejsce końcowe. Pola w tablicy są instancjami klasy *Path* i określają ścieżkę pomiędzy dwoma punktami określonymi indeksami tego pola. Ostatnim atrybutem tej klasy jest numeryczny identyfikator punktu startowego.

6 Najważniejsze algorytmy

6.1 Odczytywanie danych z pliku

Program wczytuje dane z plik wiersz po wierszu przy użyciu *BufferedReader*. Jeżeli napotka wiersz z nazwą tabeli *Miejsca Podróży*, to przechodzi do następnego wiersza i przekształca go w listę napisów, gdzie podział przebiega względem znaku kreski pionowej. Program sprawdza czy lista ta zawiera nazwy wszystkich wymaganych kolumn i zapisuje które w kolejności jest dane pole. Następnie wczytuje się kolejne wiersze, dzieli je na listę, sprawdza odpowiednie pola i dodaje wartości do listy *allPlaces*. Działanie to trwa dopóki program nie napotka tabeli *Czas przejścia*, bądź końca pliku. Po wczytaniu wszystkich miejsc tworzony jest graf. Przy napotkaniu tabeli *Czas przejścia*, analogicznie do poprzedniej tabeli, sprawdza odpowiednie wartości kolumn i zapisuje ich kolejność. Następnie, dopóki *BufferedReader* nie napotka końca pliku, tworzone są obiekty *Path* i umieszczane w odpowiednich polach tablicy dwuwymiarowej reprezentującej graf.

6.2 Algorytm Dijkstry

Algorytm Dijkstry obliczania najkrótszych ścieżek i kosztów dojścia w grafie ważonym używany jest, żeby znaleźć najkrótszą ścieżkę między każdymi dwoma miejscami (jeżeli podamy jako dodatkowy argument listę miejsc, to znajdujemy jedynie najkrótsze ścieżki między tymi miejscami). Na tej podstawie tworzony jest nowy graf pełny, gdzie wierzchołki to obowiązkowe miejsca d odwiedzenia, a krawędzie to najkrótsze trasy między danymi dwoma punktami.

Na początku tworzone są dwie listy — jedna posiada wierzchołki przetworzone przez algorytm, a druga czekające na przetworzenie. Tworzone są również dwie tablice. Pierwsza będzie zawierała koszt dojścia do danego wierzchołka grafu od wierzchołka startowego. Druga tablica będzie zawierała numery wierzchołków poprzedników na ścieżce do wierzchołka startowego. Dzięki tej tablicy będziemy w stanie odtworzyć każdą najkrótszą ścieżkę od wierzchołka startowego. Dopóki lista z nieprzetworzonymi wierzchołkami nie jest pusta, w pętli wykonywane są operacje:

1. Wybieramy ze listy wierzchołków nieprzetworzonych wierzchołek u o najmniejszym koszcie dojścia.

2. Wybrany wierzchołek usuwamy z listy wierzchołków nieprzetworzonych i dodajemy do listy wierzchołków przetworzonych.

3. Dla każdego sąsiada tego wierzchołka, który jest wciąż w liście wierzchołków nieprzetworzonych, sprawdzamy, czy `[koszt dojścia do sąsiada]` jest mniejszy od `[koszt dojścia do tego wierzchołka + waga krawędzi wierzchołek-sąsiad]`. Jeśli tak, to wyznaczamy nowy koszt dojścia do sąsiada jako `[koszt dojścia do wierzchołka + waga krawędzi wierzchołek-sąsiad]`. Następnie wierzchołek czynimy poprzednikiem sąsiada.

6.3 Algorytm Heap'a

Algorytm Heap'a wykorzystywany jest żeby znaleźć wszystkie cykle Hamiltona w nowym grafie. Pozwala to sprawdzić długość każdego cyklu i wybrać ten najkrótszy.

Algorytm Heap'a pozwala znaleźć wszystkie permutacje danej tablicy elementów. W każdym kroku algorytm używa `k` elementów początkowej tablicy. Definiujemy funkcję, która przyjmuje liczbę elementów `k` jako argument. Jeżeli `k` jest równe `1`, dodajemy gotową permutację do tablicy wyników. W innym wypadku wywołujemy funkcję rekurencyjnie z argumentem `k-1`. Pod tą rekurencją w funkcji wywołujemy pętlę, w której iterujemy po kolejnych elementach tablicy od pierwszego do `k-2`. Jeżeli `k` jest parzyste zamieniamy każdy kolejny element iteracji z elementem `k-1`. Jeżeli `k` jest nieparzyste zamieniamy pierwszy element tablicy z elementem `k-1`. Po tej instrukcji warunkowej znów wywołujemy funkcję rekurencyjnie z argumentem `k-1`.