Specyfikacja implementacyjna projektu indywidualnego *„Bieszczadzki Komiwojażer”*

Wykonał: Piotr Ferdynus
Sprawdził: mgr inż. Paweł Zawadzki
Data: 13.11.2019

1 Wprowadzenie

Celem specyfikacji jest sprecyzowanie sposobu implementacji funkcjonalności projektu *„Bieszczadzki Komiwojażer”*. W dokumencie zostanie określona logika działania programu, opisane zostaną planowane struktury danych oraz zastosowane algorytmy.

2 Środowisko deweloperskie

Opis charakterystyki sprzętu i oprogramowania, które zostaną użyte podczas pracy nad projektem.

2.1 Parametry sprzętowe

Podczas procesu wytwarzania oprogramowania zostaną wykorzystane dwie stacje robocze o następującej specyfikacji:

Mobilna stacja robocza:

```
Procesor AMD Ryzen 5 2500U
Zintegrowana karta graficzna Radeon Vega 8 Mobile
Pamięć RAM DDR4 8GB
Windows 10 Home wersja 1903
```

Stacjonarna stacja robocza:

```
Procesor Intel Core i5-7400
Karta graficzna NVidia Geforce GTX 1060 3GB
Pamięć RAM DDR4 8GB
Windows 10 Education wersja 1903
```

2.2 Oprogramowanie

Na obu komputerach zostało zainstalowane oprogramowanie pozwalające na pracę w języku programowania Java:

```
SDK Java 11.0.2 2019-01-15 LTS
Java(TM) SE Runtime Environment 18.9
Java HotSpot(TM) 64-Bit Server VM 18.9
IDE Intellij IDEA Ultimate 2019.1.1
```

3 Przebieg wprowadzania zmian

Ustalenie sposobu wprowadzania zmian do projektu.

3.1 Wiadomości do repozytorium

Komentarze zmian wprowadzanych do repozytorium będą realizować szablon: numer wersji + krótki komentarz. Szczegółowy opis numeru wersji znajduje się w sekcji *Numer wersji*.

3.2 Numer wersji

Podczas realizacji projektu zostały przyjęte ogólnie akceptowane zasady wersjonowania projektów informatycznych. Numer wersji występuje w postaci *X.Y.Z*, gdzie X, Y i Z reprezentują liczby naturalne. Człon *X*, indeksowany od zera, to iteracja wydań niekompatybilnych wstecznie lub wnoszących istotne zmiany w funkcjonalności oprogramowania. Człon *Y*, indeksowany od jedynki, reprezentuje mniejszy przyrost funkcjonalności aplikacji. Ostatnia część *Z*, indeksowana od zera, informuje o poprawie błędów w funkcjonowaniu programu.

3.3 Organizacja repozytorium

Repozytorium będzie składać się z gałęzi *master* i *dev*. Główna gałąź (*master*) będzie zawierała stabilną, działającą wersję projektu. Wszelkie zmiany i nowe funkcjonalności będą rozwijane w gałęzi *dev*. Po poprawnym zaimplementowaniu określonej funkcjonalności i uzyskaniu kolejnej iteracji programu, zostanie dokonany merge gałęzi głównej z gałęzią *dev*.

4 Struktura klas programu

Przedstawienie i opis planowanego schematu klas gotowego programu oraz przewidywany sposób działania algorytmu.

4.1 Schemat klas

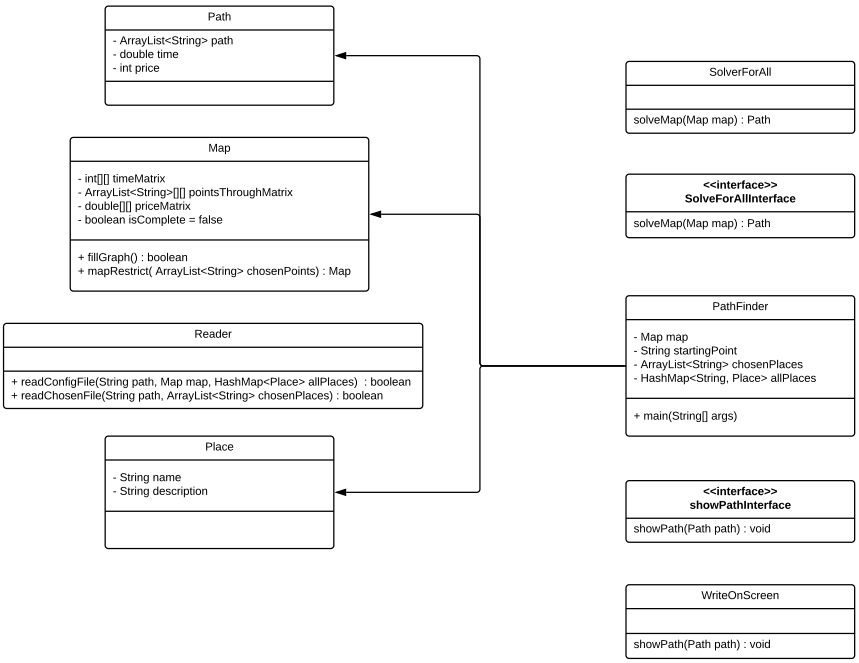

Diagram klas „Bieszczadzki Komiwojażer”

4.2 Opis klas

Przedstawienie funkcji i zadań poszczególnych klas.

4.2.1 PathFinder

Klasa główna, odpowiadająca za obsługiwanie programu. Przyjmuje argumenty wywołania, zarządza przebiegiem rozwiązywania zadanego problemu i komunikacją z użytkownikiem.

4.2.2 SolverForAll

Klasa odpowiada za właściwe wyznaczenie rozwiązania. Szczegółowy opis działania klasy i zastosowanych w niej algorytmów znajduje się w części *Schemat działania* i *Algorytmy*.

4.2.3 Path

Klasa umożliwiająca ustrukturyzowane przechowywanie wyznaczonej optymalnej ścieżki.

4.2.4 Map

Klasa odpowiadająca za przechowywanie danych o mapie Bieszczad i wykonywanie najważniejszych operacji. Wczytane punkty i trasy je łączące zostaną zapisane w postaci macierzy incydencji grafu skierowanego *timeMatrix*, w której w odpowiednich indeksach zostanie zawarta długość trasy pomiędzy punktami według następującego schematu:

ID_miejsca	A	B	C	D	E	F
A	-1	t(A->B)	t(A->C)	t(A->D)	t(A->E)	t(A->F)
B	t(B->A)	-1	t(B->C)	t(B->D)	t(B->E)	t(B->F)
C	t(C->A)	t(C->B)	-1	t(C->D)	t(C->E)	t(C->F)
D	t(D->A)	t(D->B)	t(D->C)	-1	t(D->E)	t(D->F)
E	t(E->A)	t(E->B)	t(E->C)	t(E->D)	-1	t(E->F)
F	t(F->A)	t(F->B)	t(F->C)	t(F->D)	t(F->E)	-1

Schemat macierzy timeMatrix po wczytaniu danych

Jeżeli dwa wierzchołki nie są połączone krawędzią, wartość przechowywana w macierzy wynosi *-1*. Analogicznie uzupełniana jest macierz *priceMatrix*, w której przechowywane wartości odzwierciedlają wysokość opłaty na konkretnym odcinku trasy.

Metoda *fillGraph* jest kluczowa dla działania programu. Wykorzystując algorytm Djikstry, opisany w sekcji *Algorytmy*, uzupełnia ona brakujące krawędzie grafu, wyznaczając najkrótszą drogę pomiędzy wierzchołkami. Jeżeli najkrótsza droga przebiega przez inne wierzchołki, zostają one zapisane w postaci listy liniowej w odpowiednim polu macierzy *pointsThroughMatrix*.

Metoda *mapRestrict* ogranicza mapę tak, by zawierała jedynie te wierzchołki, których *ID_ miejsca* znajduje się w liście *chosenPoints*.

4.2.5 Reader

Klasa odpowiadająca za prawidłowe odczytanie danych z plików wejściowych oraz o odpowiednie zakomunikowanie o występujących w nich błędach. Metody *readConfigFile()* i *readChosenFile()* są odpowiedzialne za, odpowiednio, odczyt pliku konfiguracyjnego i odczyt pliku z wybranymi miejscami.

4.2.6 Place

Klasa ułatwiająca przechowywanie danych o miejscach odczytanych z pliku.

4.2.7 WriteOnScreen

Klasa wypisująca wynik końcowy w sposób zrozumiały dla użytkownika.

4.3 Schemat działania

Klasa *PathFinder* rozpoczyna analizę argumentów wejściowych. Inicjalizuje odczyt plików wywołując odpowiednie metody klasy *reader*. Następnie wywołuje metodę *fillGraph()* klasy *Map*. Szczegółowy opis działania tej metody znajduje się w sekcji *Opis klas*. Jeżeli został podany plik zawierający wybrane miejsca, wywołuje metodę *mapRestrict()*. Tak przygotowane dane zostają przekazane do funkcji *solveMap()*. Otrzymany obiekt typu *Path* zostaje przekazany do metody *showPath* i wypisany w sposób zrozumiały dla użytkownika.

5 Algorytmy

Opis algorytmów wykorzystanych przy implementacji projektu.

5.1 Algorytm Djikstry

5.1.1 Opis

Algorytm służy do znajdowania najkrótszej możliwej drogi pomiędzy wybranym wierzchołkiem a pozostałymi wierzchołkami grafu.

5.1.2 Zastosowanie

Algorytm zostanie zastosowany w metodzie *fillGraph* klasy *Map*, by dopełnić brakujące krawędzie grafu, zastępując je najkrótszą możliwą drogą wiodącą przez jak najmniejszą ilość punktów.

5.1.3 Złożoność

Złożoność czasowa algorytmu w przypadku implementacji przez kopiec to O(E logV), gdzie E reprezentuje liczbę krawędzi, a V liczbę wierzchołków.

5.2 Algorytm Helda–Karpa

5.2.1 Opis

Algorytm służy do znajdowania minimalnego drzewa rozpinającego dla grafu pełnego, to jest drzewa łączącego wszystkie wierzchołki grafu o najmniejszej sumie wag. Wzór algorytmu ma postać rekurencyjną i przedstawia się następująco:

$$\begin{cases} D(S,p) = d_{1,p} & \text{gdy } s = 1 \\ D(S,p) = min_{x \in (S-p)}(D(S-p,x) + d_{x,p}). & \text{gdy } s > 1 \end{cases}$$

S – zbiór wierzchołków grafu

p ∈ S – wierzchołek grafu na którym ma zakończyć się droga

D(S, p) – najkrótsza możliwa droga z wybranego wierzchołka początkowego, przez wszystkie wierzchołki ze zbioru S, kończąca się na wierzchołku p.

5.2.2 Zastosowanie

W programie zostanie wykorzystany do wyznaczenia najkrótszej możliwej drogi zawierającej wszystkie wierzchołki w grafie.

5.2.3 Złożoność

Złożoność czasowa algorytmu wynosi $O(n^2 2^n)$.

6 Struktury danych

Opis wykorzystanych struktur danych i cel ich zastosowania.

6.1 Kopiec

Kopiec zostanie wykorzystany do implementacji kolejki priorytetowej, niezbędnej przy implementacji algorytmu Djikstry, opisanego w sekcji *Algorytmy*.

6.2 Macierz

Macierz zostanie wykorzystana do przechowywania informacji o mapie wczytanej do programu.

6.3 Tablica mieszająca

Tablica mieszająca (z ang. *hash table*) zostanie wykorzystana do przechowywanie szczegółowych danych na temat punktów, jako klucz wykorzystując ich *ID_ miejsca*.

6.4 Lista liniowa

Lista liniowa zaimplementowana w postaci *ArrayList* zostanie wykorzystana do przechowywania listy wybranych miejsc.