Wyszukiwanie najkrótszej trasy przejazdu (algorytm Dijkstry)

DOKUMENTACJA PROJEKTU

Dominik Mańkowski
MAIL: D.MANKOWSKI@STUD.ELKA.PW.EDU.PL

Prowadzący:
DR INŻ. PRZEMYSŁAW KORPAS

Spis treści

1. WSTĘP

- Uszczegółowienie tematu:
  Program wyszukujący najkrótszą trasę pomiędzy dwoma miastami w Polsce za pomocą algorytmu Dijkstry. Zaproponować format przechowywania danych o miastach i drogach pomiędzy nimi. Użytkownik powinien mieć możliwość wskazania dwóch dowolnych miast z bazy, a program powinien wyświetlić optymalną trasę (w sensie wag algorytmu).

- Zakłada funkcjonalność (uzgodniona z Prowadzącym):
  - Program wyświetla najkrótszą ścieżkę pomiędzy dwoma wybranymi przez użytkownika miastami. W przypadku braku połączenia zostaje wyświetlony odpowiedni komunikat.
  - Użytkownik ma możliwość dodawania i usuwania połączeń z bazy, usuwania miasta znajdującego się w bazie.
  - Program umożliwia wyświetlenie istniejących połączeń oraz miast będących w bazie.
  - Połączenia pomiędzy miastami są przechowywane w pliku DataBase.txt.
  - Graf jest skierowany, tj. jeśli można dostać się z miasta A do B, to nie oznacza to, że można dostać się z miasta B do A.

- Modyfikacje w stosunku do zakładanej funkcjonalności projektu:
  Oprócz zakładanej funkcjonalności, możliwe jest również:
  - Zmiana nazwy miasta.
  - Zmiana odległości pomiędzy miastami w istniejącym połączeniu.

2. ZARYS DZIAŁANIA PROGRAMU

Po uruchomieniu programu pierwszą czynnością, jaka zostaje jest wykonana jest inicjalizacja grafu oraz wczytanie istniejących połączeń z pliku DataBase.txt. Wczytywanie się nie powiedzie, jeśli plik nie istnieje lub jest źle wypełniony. W takiej sytuacji użytkownik będzie musiał dodać połączenia z poziomu menu programu. Następnie zostaje wyświetlone menu programu, gdzie użytkownik ma do wyboru jedną z następujących opcji:

1) „Find the shortest path" – użytkownik podaje nazwy dwóch miast (punkt startowy i początkowy), a następnie program wywołuje funkcję wyszukującą najkrótsze połączenie. Jeśli istnieje ścieżka pomiędzy podanymi miastami, to program wypisze jej przebieg. W przeciwnym razie zostanie wyświetlona odpowiednia informacja o braku połączenia pomiędzy miastami.
2) „Add a connection" – użytkownik podaje nazwy dwóch miast oraz odległość pomiędzy nimi. Następnie połączenie zostaje dodane do grafu w taki sposób, aby zachować kolejność alfabetyczną (rosnącą).
3) „Remove a connection" – użytkownik podaje nazwy dwóch miast i połączenie zostaje usunięte z grafu. Jeśli to było jedyne połączenie dla któregoś z (lub obu) miast, to zostaje/ą ono/e usunięte całkowicie z grafu.
4) „Remove a city" – użytkownik podaje nazwę miasta i zostaje ono (wraz ze wszystkimi istniejącymi z i do niego połączeniami) usunięte z grafu.
5) „Rename a city" – użytkownik podaje nazwę miasta i zostaje zmieniona jego nazwa (w taki sposób, aby zachować kolejność alfabetyczną w liście połączeń). Funkcja ta korzysta z dwóch wcześniejszych funkcji – na początku kopiuje nazwy połączonych miast, a następnie uruchamia funkcję „Remov e city" oraz „Add connection" – przy czym to drugie zostaje uruchomione dla każdego ze skopiowanych miast.
6) „Change a distance" – użytkownik podaje nazwy dwóch miast, a następnie wprowadza nową odległość pomiędzy miastami.
7) „Show list of cities" – program wyświetla istniejące miasta w grafie (w kolejności alfabetycznej).
8) „Show connections" – program wyświetla istniejące połączenia pomiędzy miastami, jakie są w grafie.
9) „Author" – program wyświetla informacje o autorze.
10) „Exit the program" – wywołana zostaje funkcja aktualizującą plik DataBase.txt, a następnie zostaje zamknięty program.

3. Reprezentacja grafu

W momencie uruchomienia programu liczba wierzchołków grafu (tj. liczba miast w bazie) nie jest definitywnie ustalona (tj. może zostać zmieniona), także musi on mieć możliwość dynamicznej zmiany swojego rozmiaru. Rozwiązaniem tego problemu jest lista, której każdy węzeł zawiera głowę do listy sąsiedztwa. Struktura „Graph" zawiera wskaźnik na pierwszy element głównej listy.

```
typedef struct Graph
{
   struct AdjacencyList *head;
}Graph_t;
```

Wystarczy jedynie wskaźnik do pierwszego węzła, aby wykonać wszelkie operacje na liście (typu dodaj/usuń połączenie, itd.). Struktura tworzącą główną listę jest zdefiniowana w następujący sposób:

```
typedef struct AdjacencyList
{
   char *from;
   struct AdjacencyList *next;
   struct AdjacencyListNode *head;
   unsigned int ordinal_number;
}AdjacencyList_t;
```

Każdy węzeł „głównej" listy zawiera:

- *from – łańcuch znaków będący nazwą miasta, od którego będą wychodzić połączenia w danej liście sąsiedztwa.
- *next – wskaźnik do następnej listy sąsiedztwa.
- *head – wskaźnik do pierwszego elementu wierzchołka listy sąsiedztwa.
- ordinal_number – liczba porządkowa danego miasta w grafie, tj. listy są ułożone w porządku alfabetycznym rosnącym i miasto pierwsze w kolejności alfabetycznej będzie miało liczbę porządkową równą 0, kolejne 1, itd.

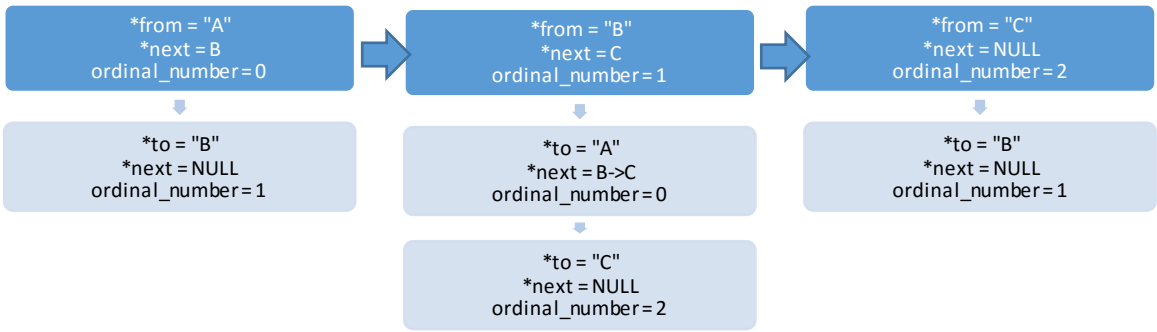
Węzły listy sąsiedztwa zdefiniowane są w następujący sposób:

```
typedef struct AdjacencyListNode
{
   struct AdjacencyListNode *next;
   char *to;
   double distance;
   unsigned int ordinal_number;
}AdjacencyListNode_t;
```

Opis elementów struktury:

- *next – wskaźnik do następnego elementu listy.
- *to – łańcuch znaków zawierający nazwę miasta, do którego jest połączenie.
- distance – odległość pomiędzy połączonymi miastami.
- ordinal_number – numer porządkowy miasta odpowiadający numerowi porządkowemu z „głównej” listy.

Przykładowa struktura list wykorzystana w programie jest zaprezentowana na rysunku 1.



Rysunek 1

*next = B->C oznacza, że *next* wskazuje na węzeł znajdujący się w liście sąsiedztwa „B” (kolor szarawy), a nie na element „C” będący w głównej liście (tj. w kolorze niebieskim).

4. WCZYTYWANIE GRAFU Z PLIKU

Po uruchomieniu programu następuje próba wczytania danych z pliku DataBase.txt. Algorytm jest następujący:

1) Uruchom funkcję **GraphCreation()** z pliku GraphCreation.c.
2) Na początku funkcja ta wywoła funkcję **CreateLists()** z pliku ListCreation.c, która ma za zadanie wypełnienie list **connections_list** i **unique_cities_list**. W tej funkcji odbędą się poniższe czynności:
3) Otwórz plik.
4) Pobierz dane w formacie „miasto1 miasto2 odległość".
5) Wczytane połączenie jest dodawane do listy **connections_list.**
6) Wczytanie połączeń do grafu nie odbędzie się, jeśli:
    a. Dane w pliku są w złym formacie.
    b. Nazwa miasta jest nieprawidłowa (zawiera niedozwolone znaki).
    c. „miasto1" i „miasto2" nazywają się tak samo.
    d. Odległość pomiędzy miastami jest mniejsza lub równa 0.
    e. Połączenie jest już na liście.
    f. Odległość od A do B jest inna niż B do A (w przypadku połączenia skierowanego w obydwie strony).
    g. Liczba połączeń jest równa 0.
7) Jeśli wystąpił błąd, to wtedy wyświetl odpowiednią informację o błędzie i powiedz użytkownikowi, że ma dodać połączenia ręcznie. Wróć do głównego menu.
8) Jeśli nie wystąpił błąd, to uruchom funkcję **AddUniqueCitiesToTheList()**, która ma za zadanie stworzenie listy **unique_cities_list** -będzie ona zawierać nazwy unikalnych miast posortowanych w porządku alfabetycznym malejącym – czyli na początku listy będą miasta zaczynające się na litery z końca alfabetu. Funkcja **AddUniqueCitiesToTheList()** oraz **CreateLists()** kończą swoje działanie. Następuje powrót do funkcji **GraphCreation()**.
9) Na podstawie list **unique_cities_list** oraz **connections_list** zostają dodane połączenia do grafu w kolejności alfabetycznej (funkcja **AddEdgesToGraph()**).
10) Usunięcie dynamicznych list z pamięci.
11) Powrót do głównego menu.

5. INTERFEJS PROGRAMU

Logika aplikacji oraz interfejs są od siebie oddzielone. Funkcje odpowiadające za wyświetlanie informacji oraz pobieranie danych od użytkownika są zdefiniowane w pliku ProgramInterface.c. Funkcje odpowiedzialne za wyświetlanie błędów są zdefiniowane w pliku ErrorsMessages.c – błędy wyświetlane są na standardowym strumieniu błędów (stderr).

W pliku ProgramInterface.c zdefiniowane są funkcje:

- void clearBuffer(void)
  Funkcja czyści bufor wejściowy.
- void getCitysName(manage_cities, char *)
  Funkcja wyświetla informację w zależności od wartości zmiennej typu manage_cities, a następnie wczytuje nazwę miasta podaną przez użytkownika do zmiennej typu char *.
- void getDistanceBetweenCities(double *)
  Funkcja pobiera dystans pomiędzy dwoma miastami i zapisuje wartość do zmiennej typu double *.
- void getUsersChoice(char *)
  Funkcja pobiera wybór użytkownika w głównym menu.
- void showAuthor(void)
  Funkcja wyświetla informacje o autorze programu.
- void showCityList(const Graph_t *)
  Funkcja wyświetla listę miast (w kolejności alfabetycznej), które są w grafie.
- void showConnections(const Graph_t *)
  Funkcja wyświetla połączenia połączenia pomiędzy miastami, jakie są w grafie.
- void showMenuOptions(unsigned int)
  Co 3 złe wpisanie opcji wyboru w głównym menu, funkcja wyświetla główne menu. Dla reszty przypadków zostaje wyświetlona informacja: *„Type in the correct choice.”.*
- void showThePath(shortest_path, const char *, const char *, unsigned int, double *)
  Funkcja wyświetla przebieg najkrótszej trasy pomiędzy miastami wybranymi przez użytkownika. Jeśli nie ma połączenia pomiędzy tymi miastami (tj. jeśli w tablicy kosztów dojścia odległość od źródła jest równa nieskończoność) to zostaje wyświetlona odpowiednia informacja.

W pliku ErrorsMessages.c zdefiniowane są następujące funkcje:

- void printDataBaseError(programError_t)
  Funkcja ta wyświetla komunikat błędu na podstawie wartości zmiennej typu programError_t. Błędy te dotyczą wypełnienia pliku DataBase.txt.
- void printSystemError(systemError_t)
  Funkcja ta wyświetla komunikat błędu na podstawie wartości zmiennej typu systemError_t. Błędy te dotyczącą problemów na poziomie systemu, a nie aplikacji (np. brak wolnej pamięci).
- void printErrorUsesrsAction(programError_t)
  Funkcja ta wyświetla komunikat błędu na podstawie wartośći zmiennej typu programError_t. Błędy te są spowodowane akcją użytkownika (np. wpisanie złej nazwy miasta, podanie niepoprawnej odległości pomiędzy miastami, itd.).

Wartości typu programError_t są typem wyliczeniowym i są zdefiniowane w pliku ErrorCodes.h. Oznaczają one:

- *ALREADY_ON_THE_LIST* – połączenie już jest na liście.
- *BAD_FORMAT* – zły format danych.
- *CANT_OPEN_FILE* – nie można otworzyć pliku DataBase.txt.
- *CANT_UPDATE_FILE* - nie można uaktualnić pliku DataBase.txt.
- *CITY_ALREADY_IN_GRAPH* – miasto już jest w grafie.
- *CONNECTION_ALREADY_IN_GRAPH* – połączenie już jest w grafie.
- *DIFFERENT_DISTANCE* – inna odległość z miasta A do B niż z B do A.
- *DIRECTED_CONNECTION* – jest połączenie z miasta A do B, ale nie ma z B do A.
- *INCORRECT_DISTANCE* – niepoprawna odległość (np. mniejsza od 0).
- *NO_SUCH_CITY* – brak takiego miasta w grafie (np. przy wybieraniu miasta do usunięcia z grafu).
- *NO_SUCH_CONNECTION* – brak takiego połączenia w grafie.
- *NO_CONNECTIONS* – brak połączeń w pliku DataBase.txt.
- *NO_DIGIT* – podana dana nie jest cyfrą (np. przy wyborze opcji z głównego menu).
- *NO_ERROR* – brak błędu, nie wyświetla żadnej wiadomości.
- *SAME_NAME* – drugie miasto nazywa się tak samo jak pierwsze.
- *TOO_MANY_SIGNS* – podano za dużo znaków przy wyborze opcji z głównego menu (więcej niż jeden znak).

Wartości typu systemError_t są typem wyliczeniowym i są również zdefiniowane w pliku ErrorCodes.h. Oznaczają one:

- *CANT_ALLOCATE_MEMORY* – nie można zaalokować pamięci.

6. Opis algorytmu wyszukiwania najkrótszej ścieżki

Po wybraniu przez użytkownika opcji „*Find shortest path*” zostaje uruchomiona funkcja **findShortestPathMenu(Graph_t *graph)**, która wykonuje następujące czynności:

1) Pobranie nazwy pierwszego miasta od użytkownika.
2) Pobranie nazwy drugiego miasta od użytkownika.
3) Uruchomienie funkcji **updateOrdinalNumbers()**, która aktualizuje liczby porządkowe miast w grafie (patrz Rysunek 1).
4) Wywołanie funkcji **findShortestPath()**, która wykonuje niezbędne akcje przed wywołaniem algorytmu Dijkstry.

Funkcja **void findShortestPath(const Graph_t *graph, const char *source, const char *destination)**

Działa w następujący sposób:

1) Znajdź liczbę wierzchołków w grafie (czyli liczbę miast) oraz ustal, jaki numer porządkowy ma miasto początkowe (***source***) oraz miasto końcowe (***destination***).
2) Aby było łatwiej potem wypisać nazwy miast stanowiące najkrótszą ścieżkę, zapamiętaj nazwy miast w tablicy ****cities_names**.
3) Utwórz tablicę kosztów dojścia od wierzchołka początkowego do i-tego wierzchołka w grafie i ustal wartości na LLONG_MAX (co jest równoważne nieskończoności).
4) Ustal koszt dojścia do wierzchołka początkowego na 0.
5) Stwórz tablicę ***previous_vertex_array**, która będzie pamiętała poprzedni wierzchołek przed dojściem do i-tego (co będzie stanowić najkrótszą ścieżkę). Domyślnie ustal wszystkie wartości na -1. Przykładowo, jeśli najkrótsza ścieżka będzie wyglądać tak: 0->3->4, to **pva[0]** = -1, **pva[3]** = 0, **pva[4]** = 3. Dzięki tej tablicy będziemy mogli odtworzyć najkrótszą ścieżkę.
6) Utwórz kolejkę priorytetową **priority_queue.** Kolejka ma następującą reprezentację:

```
typedef struct PQNode
{
   unsigned int city_number;
   double key;
}PQNode_t;
```

7) Jest to jednowymiarowa tablica zawierająca **number_of_vertices** (czyli liczba miast) struktur typu **PQNode_t**. Kolejka jest realizowana za pomocą **kopca binarnego** typu **min**. **Key** to jest klucz wg którego będzie posortowana kolejka priorytetowa, a **city_number** mówi o tym, jakie miasto odpowiada dla danego klucza.
8) Początkowo każdy **key** jest ustawiony na LLONG_MAX, a **city_number** na od 0 do **numer_of_vertices**.
9) Ustaw minimalny klucz (czyli najwyższy priorytet) dla wierzchołka startowego i odbuduj kopiec.
10) Uruchom funkcję **DijkstrasAlgorithm()**.

Funkcja:

`void DijkstrasAlgorithm(PQNode_t **priority_queue, size_t number_of_vertices, unsigned int destination_number, const Graph_t *graph, double *distance_array, int *previous_vertex_array)`

wykonuje algorytm Dijkstry dla wprowadzonych danych. Na początek należy wyjaśnić dane wejściowe:

Nazwa zmiennej	Typ zmiennej	Opis zmiennej
priority_queue	PQNode_t **	Kolejka priorytetowa.
number_of_vertices	size_t	Liczba wierzchołków w kolejce priorytetowej.
destination_number	unsigned int	Liczba porządkowa miasta docelowego.
graph	const Graph_t *	Graf zawierający połączenia pomiędzy miastami.
distance_array	double *	Tablica kosztów dojść od miasta początkowego do i-tego.
previous_vertex_array	int *	Tablica zawierająca dane potrzebne do dojścia od początku do końca ścieżki.

procedura Algorytm Dijkstry:
dopóki priority_queue nie jest pusta **wykonaj**
    zaktualizuj distance_array oraz previous_vertex_array
    jeżeli priority_queue[0] jest miastem docelowym **to**
        wyjdź
    usuń pierwszy element **priority_queue** i **zastąp** go ostatnim
    odbuduj kopiec
    zaktualizuj klucze w priority_queue
    odbuduj kopiec

Funkcją odpowiedzialną za aktualizację tablicy kosztów dojść i tablicy poprzednich wierzchołków jest:

```
void updateDistanceArray(unsigned int citys_number, double *distance_array, const Graph_t graph, int *previous_vertex_array)
```

Funkcja sprawdza, czy koszt dojścia od wierzchołka początkowego do jednego z sąsiadów wierzchołka w którym obecnie algorytm się znajduje jest większy czy mniejszy od kosztu dojścia z tablicy **distance_array**. Jeśli jest mniejszy, to koszt dojścia zostaje zaktualizowany, tak samo jak **previous_vertex_array**.

Funkcją która usuwa pierwszy element kolejki priorytetowej (tj. korzeń drzewa binarnego) jest:

```
void extractMin(PQNode_t **priority_queue, size_t *number_of_vertices)
```

Jej przebieg jest następujący:

procedura extractMin
    zamień pierwszy element priority_queue z ostatnim
    usuń ostatni element priority_queue
    odbuduj korzeń

Funkcją usuwającą ostatni element kolejki priorytetowej jest:

```
void removeLastPQ(PQNode_t **priority_queue, size_t *number_of_vertices)
```

Zmniejsza ona również **number_of_vertices** (czyli liczbę wierzchołków które są w kolejce priorytetowej) o 1.

Po zamienieniu korzenia z ostatnim elementem kolejki priorytetowej należy odbudować kopiec, ponieważ został on naruszony. Realizuje to funkcja:

void heapifyRoot(PQNode_t **priority_queue, size_t *number_of_vertices, unsigned int parent)

```
procedura heapifyRoot
        left := parent * 2 + 1          // lewe dziecko
        right := left + 1               // prawe dziecko
        jeżeli left ≥ number_of_vertices
                wyjdź                   // jeżeli już nie ma lewego dziecka
        dmin := wartość klucza w kolejce dla lewego dziecka
        pmin := left                    // parent min
                jeżeli right < number_of_vertices
                jeżeli dmin > wartość klucza w kolejce dla prawego dziecka
                        dmin := wartość klucza w kolejce dla prawego dziecka
                        pmin := right
        jeżeli wartość klucza w kolejce dla parent ≤ dmin
                wyjdź
        zamień elementy w kolejce(parent, pmin)
        parent := pmin
        powtórz heapifyRoot
```

Funkcją realizującą zamianę elementów w kolejce jest:

void swapPQNodes(PQNode_t **priority_queue, int minimal, int current)

Ostatnią fazą realizacji algorytmu Dijkstry jest uaktualnienie kluczy w kolejce i odbudowa kopca. Funkcją odpowiadającą za to jest:

void decreaseKeys(PQNode_t **priority_queue, double *distance_array, size_t *number_of_vertices)

```
procedura decreaseKeys
        dla każdego i ∈ [0, number_of_vertices − 1]
                zaktualizuj wartość klucza dla i-tego elementu w kolejce priorytetowej
                odbuduj kopiec
```

W czasie tej operacji naruszona zostaje struktura kopca, dlatego trzeba go odbudować. Z racji, że nie jest to korzeń kopca należy użyć bardziej ogólnej funkcji niż **heapifyRoot()**. Realizuje to funkcja:

`void heapify(PQNode_t **priority_queue, size_t number_of_vertices, int child)`

procedura heapify
    jeżeli child jest równy 0
        wyjdź
    parent := $\lfloor \frac{child-1}{2} \rfloor$
    jeżeli wartość klucza **parent** ≤ wartość klucza **child** w kolejce priorytetowej
        wyjdź
    zamień elementy w kolejce(parent, child)
    child := parent
    powtórz heapify

Po dojściu algorytmu Dijkstry do wierzchołka końcowego następuje wyjście z algorytmu, usunięcie kolejki priorytetowej z pamięci, a następnie powrót do funkcji **findShortestPath()**. Zostają uruchomione dwie funkcje:

`void setPath(shortest_path *path, unsigned int destination_number, const int *previous_vertex_array, char **cities_names)`

Funkcja ta tworzy listę, która zawiera miasta tworzące najkrótszą ścieżkę od miasta początkowego do końcowego.

Na koniec zostaje uruchomiona funkcja:

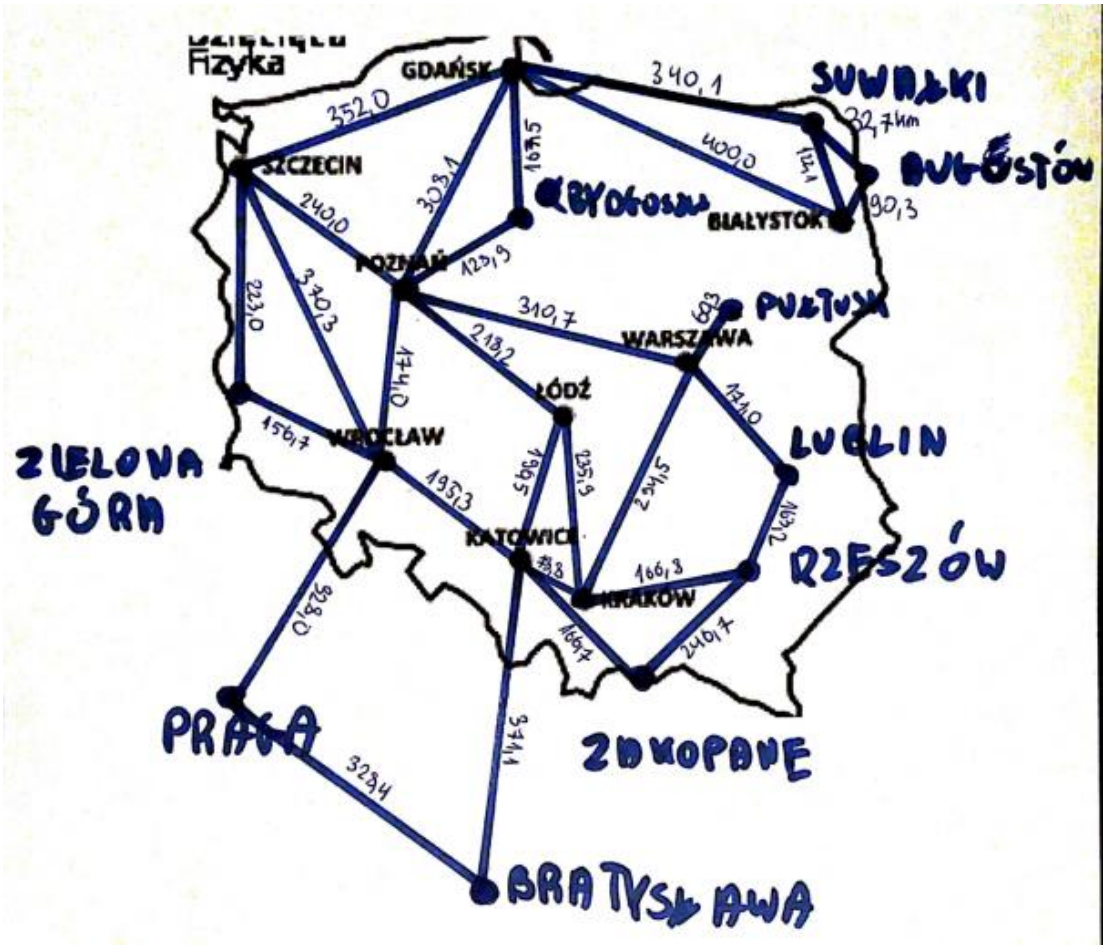
`void showThePath(shortest_path path, const char *destination, const char *source, unsigned int destination_number, double *distance_array)`

Wyświetla ona przebieg ścieżki od miasta początkowego do końcowego. W przypadku braku połączenia (tj. gdy wartość **distance_array[destination_number]** będzie równa LLONG_MAX) zostanie wyświetlona odpowiednia informacja. Następnie następuje powrót do głównego menu.

7. TESTY

Zostały przeprowadzone testy jedynie dla poprawności działania algorytmu – szybkość algorytmu dla dużych grafów nie została sprawdzona, co wynika z fakt, że:
1) ciężko wygenerować graf o zadanej gęstości przy obecnym formacie wprowadzania danych,
2) ten projekt nie miał na celu sprawdzenia wydajności algorytmu, a jedynie poprawność jego implementacji.

Wg teorii złożoność czasowa algorytmu Dijkstry z użyciem kolejki priorytetowej zaimplementowanej w postaci kopca wynosi $O(E \log v)$, gdzie E - liczba krawędzi, v - liczba wierzchołków. Program został skompilowany i sprawdzony przy użyciu GCC 6.2.0 oraz clang 800.0.38 (LLVM 8.0.0). W ramach testów do bazy danych zostały wgrane następujące połączenia (graf nieskierowany):



Przykładowo, dla połączenia pomiędzy Lublinem a Suwałkami program znalazł następującą ścieżkę:

```
The shortest path from Lublin to Suwalki is:
Lublin -> Warszawa -> Poznan -> Bydgoszcz -> Gdansk -> Suwalki
Distance from Lublin to Suwalki: 1118.20km.
```

Jak łatwo sprawdzić, jest to rzeczywiście najkrótsze połączenie w danym grafie. Podczas testowania kolejnych połączeń program nie wskazał błędnej ścieżki. Nie zostały znalezione również żadne błędy podczas działania programu. Nie było również żadnych wycieków pamięci, co zostało sprawdzone z wykorzystaniem narzędzia Valgrind w wersji 3.11.0.