Politechnika Śląska w Gliwicach
Wydział Automatyki, Elektroniki i Informatyki

Podstawy Programowania Komputerów

Mapa

autor Bartosz Siwiaszczyk

rok akademicki 2022/2023

kierunek Teleinformatyka

semestr 1

grupa 3

sekcja 6

1. Treść zadania

Napisać program, który umożliwia znalezienie najkrótszej trasy między dwoma miastami. Miasta połączone są drogami o pewnej długości. Drogi są jednokierunkowe. Plik mapy dróg ma określoną postać. W każdej linii podana jest jedna droga. Drugim plikiem wejściowym jest plik z trasami do wyznaczenia. Każda linia pliku zawiera jedną trasę w określonej postaci. Wynikiem działania programu jest plik wyjściowy z wyznaczonymi trasami, tzn. podana jest nazwa trasy, całkowita długość, a następnie poszczególne odcinki z długościami. Program uruchamiany jest z linii poleceń z wykorzystaniem następujących przełączników:

-d plik wejściowy z drogami

-t plik wejściowy z trasami do wyznaczenia

-o plik wynikowy z wyznaczony trasami

2. Analiza zadania

Zagadnienie przedstawia problem znajdowania najmniejszej odległości między dwoma punktami w grafie skierowanym za pomocą algorytmu Dijkstry.

2.1 Struktury danych

W programie wykorzystywane są struktury danych takie jak wektor, para, mapa oraz kolejka. Wybór takich struktur uzasadniony jest ze względu na ich funkcjonalność – m. in. dynamiczny rozmiar czy możliwość skorzystania z klucza.

2.2 Algorytmy

Program stosuje wersję algorytmu Dijkstry. Algorytm stosuje pętlę sprawdzając czy dotarł do miasta docelowego przerywając działanie w przypadku twierdzącym lub kontynuuje działanie w przypadku przeciwnym, w ten sposób znajdując trasę.

3. Specyfikacja zewnętrzna

Program jest uruchamiany z linii poleceń. Należy przekazać do programu nazwy plików: wejściowego i wyjściowego po odpowiednich przełącznikach

(odpowiednio: -d i -t dla plików wejściowych i -o dla pliku wyjściowego), np.

-d trasy.txt

-t sprawdz_trasy.txt

-o wynik_trasy.txt

```
Katowice Krakow 70
 Krakow Tarnow 70
 Tarnow Jaslo 50
 Katowice Gliwice 22
 Lodz Poznan 205
 Gliwice Katowice 22
 Katowice Czestochowa 70
 Czestochowa Lodz 120
 Lodz Torun 165
 Krakow Katowice 70
 Gliwice Wroclaw 180
```

Rysunek 1

```
Katowice Torun
 Krakow Poznan
 Tarnow Wroclaw
```

Rysunek 2

Rysunek 1 oraz 2 przedstawia format przykładowego pliku zawierającego drogi, oraz trasy do wyznaczenia przez program.

Uruchomienie programu bez żadnego parametru powoduje wyświetlenie komunikatu:

„Wprowadz w argumentach programu lokalizacje plikow zawierajace: -d (plik z drogami) -t (plik z trasami) -o (plik wyjsciowy)”

Nieznalezienie pliku powoduje wyświetlenie odpowiedniego komunikatu:

„Nie udalo sie otworzyc pliku […].”

4. Specyfikacja wewnętrzna

Program został zrealizowany zgodnie z paradygmatem strukturalnym. W programie rozdzielono interfejs od logiki aplikacji.

4.1 Typy zdefiniowane w programie

W programie zdefiniowano następujący typ:

```
struct Road {
    string city1;
    string city2;
    int distance;
};
```

Typ ten służy do przechowywania informacji o drogach – zawiera miasto początkowe, końcowe oraz odległość między nimi.

4.2 Ogólna struktura programu

W funkcji głównej (*main*) zawarte są: część odpowiedzialna za przełączniki, oraz wywołanie funkcji odpowiadających za główny algorytm programu oraz czytanie i zapisywanie do pliku.

```
string roadsFilename, routesFilename, outputFilename;
if (argc == 1)
    cout << "[...];
else {
    for (int i = 1; i < argc; i++) {
        if (string(argv[i]) == "-d") {
            roadsFilename = argv[i + 1];
        }
        else if (string(argv[i]) == "-t") {
            routesFilename = argv[i + 1];
        }
        else if (string(argv[i]) == "-o") {
            outputFilename = argv[i + 1];
        }
    }
}
```

Instrukcja warunkowa *if* sprawdza, czy użytkownik uruchomił program przy użyciu wymaganych przełączników, w przypadku twierdzącym przypisuje pliki do odpowiadających im zmiennym, w przypadku przeczącym informuje użytkownika o konieczności użycia przełączników.

```
auto roads = readRoads(roadsFilename);
auto routes = readRoutes(routesFilename);
```

Następnie wywoływane są dwie funkcje odpowiadające za odczytanie dróg oraz tras do wyznaczenia z plików przy użyciu wcześniej zlokalizowanych w przełącznikach zmiennych lokalizacji pliku.

```
unordered_map <string, vector <Road>> cityToRoads;
for (auto road : roads) {
    cityToRoads[road.city1].push_back(road);
}
```

W kolejnym kroku typ mapa konieczny do użycia głównego algorytmu instrukcją for iteruje wcześniej zaimportowane drogi w wektorze *Road* zapisując je w sobie.

```
vector<pair<pair<string, string>, vector<Road>>> results = shortestPath(routes, cityToRoads);
writeResults(results, outputFilename);

return 0;
```

Później, wywoływany jest wektor wyników *results* oraz funkcja *shortestPath* będące główną częścią programu.

Na samym końcu wywoływana jest funkcja *writeResults* zapisująca ostateczne wyniki do pliku.

Funkcja *main* zwraca wartość „0”.

4.3 Szczegółowy opis implementacji funkcji

```
vector<Road> readRoads(string filename) {
    vector<Road> roads;
    ifstream input(filename);
    if (!input.is_open()) {
        cerr << "Nie udalo sie otworzyc pliku z drogami." << endl;
        return roads;
    }

    string city1, city2;
    int distance;
    while (input >> city1 >> city2 >> distance) {
        roads.push_back({ city1, city2, distance });
    }

    return roads;
}
```

Funkcja *readRoads* służy do odczytania z pliku dróg jego zawartości. Jej parametrem jest typ string *filename* będący nazwą tego pliku. Na początku zdefiniowany jest wektor struktury Road *roads* oraz otwarty zostaje strumień pliku wejściowego.

Następnie funkcja sprawdza, czy plik wejściowy jest otwarty, w przeciwnym przypadku informuje o tym użytkownika. Następnie zdefiniowane są typy string *city1*, *city2* oraz int *distance* będące miastami początkowymi, oraz końcowymi i długością drogi między nimi. Następnie funkcja instrukcją *push_back* przypisuje wartości z pliku do wektora *roads*. Na końcu funkcja ten wektor zwraca.

```
vector<pair<string, string>> readRoutes(string filename) {
    vector<pair<string, string>> routes;
    ifstream input(filename);
    if (!input.is_open()) {
        cerr << "Nie udalo sie otworzyc pliku z trasami." << endl;
        return routes;
    }

    string city1, city2;
    while (input >> city1 >> city2) {
        routes.push_back({ city1, city2 });
    }

    return routes;
}
```

Funkcja *readRoutes* służy do odczytania z pliku tras jego zawartości. Jej parametrem jest typ string *filename* będący nazwą tego pliku. Na początku zdefiniowany jest wektor pary typów string *routes* oraz otwarty zostaje strumień pliku wejściowego.

Następnie funkcja sprawdza, czy plik wejściowy jest otwarty, w przeciwnym przypadku informuje o tym użytkownika. Następnie zdefiniowane są typy string *city1, city2* będące miastem początkowym oraz końcowym. Następnie funkcja instrukcją *push_back* przypisuje wartości z pliku do wektora *routes*. Na końcu funkcja ten wektor zwraca.

```
void writeResults(vector<pair<pair<string, string>, vector<Road>>> results, string filename) {
    ofstream output(filename);
    if (!output.is_open()) {
        cerr << "Nie udalo sie otworzyc pliku wyjsciowego." << endl;
        return;
    }
    for (auto result : results) {
        auto route = result.first;
        auto roads = result.second;
        if (roads.empty()) {
            output << "Nie znaleziono trasy dla trasy: " << route.first << " --> " << route.second << endl;
        }
        else {
            output << "Trasa: " << route.first << " --> " << route.second << " (" << accumulate(roads.begin(), roads.end(), 0, [](int sum, Road road) { return sum + road.distance; }) << " km):" << endl;
            for (auto road : result.second) {
                output << road.city1 << " ---> " << road.city2 << " ---> " << road.distance << "km" << endl;
            }
        }
    }
}
```

Funkcja *writeResults* służy do zapisywania skutków działania programu do pliku. Jej argumentami są: wektor *pary par typów string results* oraz wektora *typu Road*, a także typ string *filename* będący lokalizacją pliku wyjściowego. Na początku otwarty zostaje strumień pliku, później funkcja sprawdza, czy został otwarty plik – w przeciwnym przypadku użytkownik zostaje o tym poinformowany.

Następnie pętla *for* dla wektora *results* przechodzi przez każdy jego element. Definiowana jest trasa jako pierwszy element tego wektora, oraz droga jako drugi jego element. Następnie instrukcja warunkowa *if* sprawdza, czy udało się znaleźć trasę poprzez sprawdzenie czy drugi element wektora jest pusty. W przypadku twierdzącym w piku zapisana jest informacja o niemożliwości zapisania trasy oraz miasta między którymi ta próba się nie udała. W przypadku przeciwnym do pliku wyjściowego zapisywane są informacje o wyznaczonej trasie: jej początek, jej koniec, jej długość oraz każde z miast na jej przebiegu.

```
vector<pair<pair<string, string>, vector<Road>>> shortestPath(vector<pair<string, string>> routes, unordered_map<string, vector<Road>> cityToRoads) {
    vector<pair<pair<string, string>, vector<Road>>> results;
    for (auto route : routes) {
        unordered_map<string, int> distance;
        unordered_map<string, string> predecessor;
        priority_queue<pair<int, string>, vector<pair<int, string>>, greater<pair<int, string>>> queue;

        distance[route.first] = 0;
        queue.push({ 0, route.first });

        // Dopóki kolejka nie jest pusta:
        while (!queue.empty()) {
            auto top = queue.top();
            queue.pop();


            auto city = top.second;

            for (auto road : cityToRoads[city]) {
                // Jeśli dotarliśmy do końcowego miasta, zapamiętaj wynik
                if (road.city2 == route.second) {
                    distance[route.second] = distance[city] + road.distance;
                    predecessor[route.second] = city;
                    break;
                }

                // Jeśli jeszcze nie odwiedziliśmy danego miasta:
                if (distance.count(road.city2) == 0) {
                    distance[road.city2] = distance[city] + road.distance;
                    predecessor[road.city2] = city;
                    queue.push({ distance[road.city2], road.city2 });
                }
            }
        }

        // Jeśli udało się znaleźć trasę do końcowego miasta:
        if (predecessor.count(route.second) > 0) {
            vector<Road> result;
            auto currentCity = route.second;
            while (currentCity != route.first) {
                result.push_back({ predecessor[currentCity], currentCity, distance[currentCity] - distance[predecessor[currentCity]] });
                currentCity = predecessor[currentCity];
            }
            reverse(result.begin(), result.end());
            results.push_back({ route, result });
        }
        else {
            results.push_back({ route, {}});
        }
    }
    return results;
}
```

Funkcja *shortestPath* przyjmuje dwa parametry: wektor par (nazwa miasta początkowego, nazwa miasta końcowego) oraz mapę miast do list dróg.

Na początku, pętla for iteruje przez każdą trasę z wektora *routes*. Następnie, trzy unordered_map są tworzone: *distance, predecessor, oraz priority_queue. distance* mapuje nazwę miasta do odległości od miasta początkowego, *predecessor* mapuje nazwę miasta do poprzedniego miasta na drodze (w celu odtworzenia całej trasy), a *priority_queue* jest używana do przechowywania miast do odwiedzenia.

Następnie, odległość do miasta początkowego jest ustawiona na zero i miasto początkowe jest dodawane do kolejki. Pętla while jest używana, aby przetwarzać miasta w kolejce. W każdej iteracji, miasto o najniższej odległości jest pobierane z kolejki i przetwarzane. Następnie, dla każdej drogi z mapy *cityToRoads* odwiedzającej to miasto, sprawdzane jest, czy dotarliśmy do końcowego miasta. Jeśli tak, wynik jest zapisywany i pętla przestaje działać. W przeciwnym razie, jeśli miasto jest odwiedzane po raz pierwszy, jego odległość jest aktualizowana i miasto jest dodawane do kolejki.

Po wyjściu z pętli while, jeśli trasa do końcowego miasta zostanie znaleziona, jest ona rekonstruowana poprzez odtwarzanie krok po kroku od końcowego miasta do początkowego miasta, używając mapy *predecessor*. Wynik jest zapisywany jako wektor dróg. Jeśli trasa nie zostanie znaleziona, wynik jest pusty. Po zakończeniu iteracji przez wszystkie trasy, funkcja zwraca wektor par.

5. Testowanie

Program testowany był na wielu rodzajach plików oraz różnych znakach znajdujących się w tych plikach. Brak pliku powoduje poinformowanie użytkownika o tym fakcie. W trakcie testów nie stwierdzono występowania poważnych błędów.

6. Wnioski

Program mapa jest programem składającym się z wielu mniejszych funkcji. Najbardziej wymagającym zadaniem było zaimplementowanie algorytmu Dijkstry szukającego tras. Trudne okazało się także zapisywanie wyników działania programu do pliku z powodu skomplikowanej natury argumentów funkcji *shortestPath*.