Politechnika Śląska w Gliwicach
Wydział Informatyki, Elektroniki i Informatyki

Podstawy Programowania Komputerów

Spedycja

autor	Patryk Sobieraj
prowadzący	mgr inż. Marek Kokot
rok akademicki	2015/2016
kierunek	Teleinformatyka
rodzaj studiów	SSI
semestr	1
grupa	1

1. Treść zadania

Firma spedycyjna przyjmuje towary w centrali, skąd rozsyła do odbiorców w całym kraju. Trasy między miastami są zapisane w pliku w następujący sposób:

<miasto> <miasto> <odległość>

Przykładowy plik:

```
Szczecin Poznan 220
Szczecin Koszalin 110
Poznan Bytom 300
Poznan Lodz 130
Lodz Katowice 170
Bytom Katowice 15
Bytom Wroclaw 180
```

W wyniki działania programu zostanie tworzony plik z trasami spedycyjnymi w następującej postaci (przykład dla centrali w Poznaniu):

```
Poznan -> Bytom: 300
Poznan -> Lodz -> Katowice: 300
Poznan -> Szczecin -> Koszalin: 330
Poznan -> Lodz: 130
Poznan -> Szczecin: 220
Poznan -> Bytom -> Wroclaw: 480
```

Trasy spedycyjne mają najkrótsze możliwe długości. Należy wykorzystać algorytm Dijkstry.

Program uruchamiany jest z linii poleceń z wykorzystaniem następujących przełączników:

- `-i` plik wejściowy z drogami
- `-o` plik wyjściowy z trasami spedycyjnymi
- `-s` nazwa miasta startowego, gdzie znajduje się centrala

2. Analiza zadania

Zagadnienie przedstawia problem znajdywania najkrótszej trasy z zadanego miasta do wszystkich pozostałych, podając dokładny przebieg każdej z tras oraz jej całkowitą długość. W programie należy dynamicznie alokować i zwalniać pamięć dla różnych typów struktur danych oraz operować na plikach wejściowych i wyjściowych.

2.1 Struktury danych

W programie wykorzystano drzewo binarne. W jego węzłach przechowywane są nazwy miast oraz przypisany im indeks w tablicy dynamicznej. Drugą strukturą jest tablica dynamiczna, reprezentująca graf, w której przechowywane są informacje o mieście – wskaźnik na odpowiadający mu węzeł drzewa binarnego oraz lista jego sąsiadów. Każdy z elementów listy sąsiadów zawiera informację o indeksie sąsiadującego miasta oraz długość drogi do niego.

2.2 Algorytmy

Program wczytuje podane w pliku wejściowym połączenia międzymiastowe do tablicy dynamicznej reprezentującej graf połączeń międzymiastowych. Jeśli podanego miasta nie ma jeszcze w drzewie binarnym, jest ono do niego dodawane przed utworzeniem połączenia. Drzewo binarne sortowane jest na bieżąco alfabetycznie według nazwy miasta. Następnie program wyznacza wszystkie możliwe trasy z danego miasta początkowego za pomocą algorytmu Dijkstry. Dzięki temu możliwe jest wypisanie do pliku wyjściowego najkrótszych ścieżek łączących podane miasto z pozostałymi.

3. Specyfikacja zewnętrzna

Program jest uruchamiany z linii poleceń. Należy przekazać do programu nazwę pliku wejściowego z drogami między miastami, nazwę pliku wyjściowego z wyznaczonymi trasami oraz nazwę miasta startowego po odpowiednich przełącznikach (odpowiednio `-i`, `-o` i `-s`), np.

```
program.exe -i drogi.txt -o trasy.txt -s Poznan
```

Pliki są plikami tekstowymi, ale mogą mieć dowolne rozszerzenie. Przełączniki mogą być podane w dowolnej kolejności. Uruchomienie programu bez żadnego parametru, bez wszystkich parametrów lub z niewłaściwymi parametrami powoduje wyświetlenie pomocy, np.

```
program.exe -d drogi.txt -t trasy.txt
```

4. Specyfikacja wewnętrzna

W programie każda z używanych funkcji została napisana w osobnym pliku projektu. Program nie komunikuje się z użytkownikiem (poza komunikatami pomocy i błędów), a jedynie operuje na plikach wejściowych i wyjściowych.

4.1 Typy zdefiniowane w programie

W programie zdefiniowano następujące typy:

```
struct Wezel {
	string NazwaMiasta;
	int Indeks;
	Wezel* lewy;
	Wezel* prawy;
	Wezel(string Miasto, int Numer);
};

struct Sasiad {
	int IndeksSasiada;
	int Koszt;
	Sasiad* nastepny;
	Sasiad(int Indeks, int Droga);
};
```

```
struct Graf {
	Wezel* Miasto;
	Sasiad* ListaSasiadow;
	Graf();
};
```

Typ `Wezel` reprezentuje węzeł drzewa binarnego. Zawiera nazwę miasta, jego indeks w tablicy reprezentującej graf oraz wskaźniki na lewą i prawą gałąź. Typ `Sasiad` jest reprezentacją elementu listy miast sąsiednich. Zawiera indeks sąsiadującego miasta, dystans do niego oraz wskaźnik na następny element listy. Typ `Graf` reprezentuje węzeł grafu połączeń miast w tablicy dynamicznej. Zawiera wskaźnik na węzeł drzewa binarnego danego miasta oraz wskaźnik na listę sąsiadów danego miasta.

4.2 Ogólna struktura programu ze szczegółowymi opisami funkcji

W funkcji `main` wywoływana jest funkcja

```
bool WczytajPrzelaczniki(int &argc, char* argv[], string &NazwaDrogi, string &NazwaTrasy, string &NazwaStart)
```

której zadaniem jest wczytanie parametrów wiersza poleceń do programu. Jeśli parametry są niepoprawne, program wyświetli pomoc za pomocą funkcji

```
void WyswietlPomoc()
```

i zakończy się.

Jeśli podane parametry były poprawne, wywoła się funkcja

```
bool WczytajDane(string &NazwaDrogi, Wezel* &Drzewo, Graf Lista[], int &LiczbaMiast)
```

wczytująca dane z pliku wejściowego do odpowiednich struktur danych. Pobierze ona także informację o ilości miast w bazie. Korzysta ona z funkcji

```
void DodajDroge(string &sMiastoPocz, string &sMiastoKonc, int &Koszt, Wezel* &Drzewo, Graf Lista[], int &LiczbaMiast)
```

która dodaje każde z podanych w pliku połączeń do bazy. Wewnątrz funkcji, jeśli podane miasto nie istnieje jeszcze w programie, zostanie ono dodane za pomocą funkcji

```
Wezel* DodajMiasto(string &Miasto, Wezel* &Drzewo, Graf Lista[], int &LiczbaMiast)
```

Do sprawdzenia, czy dane miasto zostało już wczytane, używana jest funkcja

```
Wezel* ZnajdzMiasto(string &Miasto, Wezel* &Drzewo)
```

która zwraca wskaźnik na dane miasto w drzewie binarnym.

Następnie dla obu z miast drugie z nich zostanie oznaczone jako jego sąsiad za pomocą funkcji

```
void DodajSasiada(Wezel* &wMiastoPocz, Wezel* &wMiastoKonc, int &Koszt, Graf Lista[])
```

Jeśli wczytywanie danych z pliku nie uda się, program wyświetli komunikat o błędzie i zakończy się.

Jeśli do tej pory wszystko przebiegło pomyślnie, zostanie wywołana funkcja

```
void Dijkstra(string &NazwaStart, Wezel* &Drzewo, Graf Lista[], int Odleglosci[], int Poprzednicy[], int &LiczbaMiast)
```

będąca implementacją algorytmu Dijkstry.

Na jej działanie składają się trzy funkcje składowe. Pierwsza z nich

```
void InicjujTablice(string &NazwaStart, Wezel* &Drzewo, int Odleglosci[], int Poprzednicy[], int &LiczbaMiast)
```

zapewnia odpowiednie warunki startowe (wartości odległości i indeksy miast poprzedzających) dla działania właściwej części algorytmu. Druga z funkcji

```
void UtworzKolejke(string &NazwaStart, int Kolejka[], Wezel* &Drzewo, int Odleglosci[], int &LiczbaMiast)
```

tworzy kolejkę priorytetową, której algorytm Dijkstry używa do określenia, które z miast będzie następnym, z którego wyznaczane będą odległości do jego sąsiadów. Elementami kolejki priorytetowej są indeksy miast, natomiast priorytetem odległość danego miasta od miasta początkowego. Ostatnia z funkcji

```
void PoliczOdleglosci(int Kolejka[], Graf Lista[], int Odleglosci[], int Poprzednicy[], int &LiczbaMiast)
```

jest odpowiedzialna za obliczenie dla każdego z miast odległości od miasta startowego i wyznaczenie miasta poprzedzającego dane miasto na najkrótszej trasie do niego z miasta startowego. Do sortowania kolejki priorytetowej używana jest funkcja

```
void SortujKolejke(int Kolejka[], int Odleglosci[], int &LiczbaMiast)
```

Po wykonaniu algorytmu Dijkstry wywołana zostaje funkcja

```
void WyswietlTrasy(string &NazwaTrasy, Graf Lista[], int Odleglosci[], int Poprzednicy[], int &LiczbaMiast)
```

której zadaniem jest zapisanie do pliku wyjściowego wyznaczonych tras, co było celem zadania. Korzysta ona z funkcji rekurencyjnej

```
string WyswietlTrase(int &Miasto, Graf Lista[], int Odleglosci[], int Poprzednicy[])
```

aby wyświetlić trasy w poprawnej kolejności (od miasta początkowego do końcowego).

W następnym kroku, również w przypadku, gdy algorytm nie będzie się wykonywał

w wyniku wymienionych na początku błędów wejścia, wywołana zostanie funkcja

```
void ZwolnijPamiec(Wezel* &Drzewo, Graf Lista[], int Odleglosci[], int Poprzednicy[], int &LiczbaMiast)
```

odpowiedzialna za zwolnienie pamięci po dynamicznych strukturach danych. Korzysta ona z rekurencyjnych funkcji pomocniczych

```
void ZwolnijPamiecDrzewo(Wezel* &Drzewo)
void ZwolnijPamiecSasiad(Sasiad* &ListaSasiadow)
```

które odpowiadają za zwolnienie pamięci po odpowiednim typie struktury dynamicznej.

5. Testowanie

Program został przetestowany na różnych kombinacjach przełączników. Przy niepoprawnych kombinacja wyświetla odpowiedni do błędu komunikat, podobnie dzieje się przy źle podanej nazwie pliku wejściowego. Program był testowany też na różnych plikach wejściowych, które umożliwiały przetestowanie poprawności jego działania w charakterystycznych sytuacjach (ścieżki między miastami zapętlające się, brak połączeń między miastami itp.), dzięki którym można by zobaczyć ewentualne nieprawidłowości w wykonanych obliczeniach. W każdej z testowanych sytuacji program zwrócił prawidłowy wynik.

Program został sprawdzony pod kątem wycieków pamięci.

6. Wnioski

Program wyznaczający trasy spedycyjne jest programem ciekawym w realizacji. Łatwo można odnieść go do życia codziennego, a przez to też zauważyć jego zastosowanie praktyczne. Algorytm Dijkstry, mimo prostoty zasady działania, okazał się dość trudny w realizacji. Jego implementacja wymusza na programiście dobrą znajomość posługiwania się różnymi typami danych. Podczas pisania należy też zwrócić szczególną uwagę na dynamiczne zarządzanie pamięcią, aby nie dopuścić ani do wycieków, ani do używania nieprzydzielonej pamięci. Trudnym zadaniem było też wypisanie trasy spedycyjnej w kolejności od węzła początkowego, gdyż algorytm Dijkstry generuje trasy w odwrotną stronę.