4. Grafy. Znajdowanie najkrótszej drogi

Zapewne zdarza ci się korzystać z map internetowych. Pozwalają one m.in. znaleźć najszybszą lub najkrótszą drogę między dwoma miejscami. Jakie algorytmy to umożliwiają i jakie struktury danych te algorytmy wykorzystują? W tym temacie odpowiemy na te pytania. Zajmiemy się strukturą danych nazwaną grafem oraz podstawowymi algorytmami operującymi na grafach.

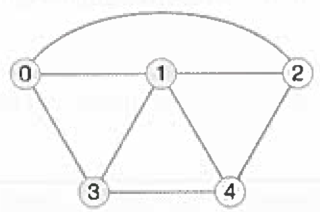
Cele lekcji

- Dowiesz się, czym jest graf, i poznasz sposoby jego reprezentowania.
- Przejrzysz wszystkie wierzchołki grafu, stosując różne algorytmy.
- Znajdziesz najkrótszą drogę pomiędzy dwoma wierzchołkami grafu.

4.1. Czym jest graf?

Graf

Graf (ang. *graph*) to struktura danych składająca się z niepustego zbioru wierzchołków i zbioru połączeń między nimi, czyli krawędzi. Rysunek 4.1 przedstawia przykład grafu złożonego z pięciu wierzchołków (ponumerowanych od 0 do 4) oraz ośmiu krawędzi łączących pary wierzchołków: 0 i 1, 0 i 2, 0 i 3, 1 i 2, 1 i 3, 1 i 4, 2 i 4 oraz 3 i 4.



Rys. 4.1. Przykład grafu złożonego z pięciu wierzchołków i ośmiu krawędzi

Sąsiednie wierzchołki grafu

Wierzchołki połączone krawędzią nazywamy **sąsiednimi wierzchołkami grafu**. W odniesieniu do mapy wierzchołki możemy interpretować jako miejscowości, a krawędzie jako drogi pomiędzy miejscowościami.

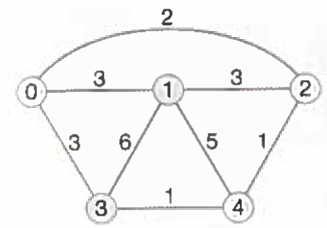
Graf prosty

Wierzchołek grafu może nie być połączony krawędzią z żadnym innym wierzchołkiem lub może być połączony krawędzią z samym sobą. Dwa wierzchołki mogą być też połączone więcej niż jedną krawędzią. W temacie zajmiemy się **grafami prostymi**, czyli takimi, w których dwa wierzchołki może łączyć co najwyżej jedna krawędź oraz z żadnego wierzchołka nie prowadzi krawędź do niego samego.

Waga krawędzi grafu

Graf ważony

Krawędziom grafu mogą być przypisane wartości zwane **wagami**. W odniesieniu do mapy wagi mogą określać np. odległości pomiędzy miejscowościami lub czasy przejazdu. Rysunek 4.2 przedstawia przykład grafu z wagami krawędzi. Taki graf nazywamy **grafem ważonym**.



Rys. 4.2. Przykład grafu ważonego

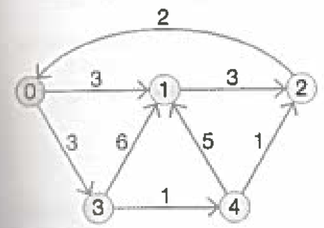
Ćwiczenie 1

Narysuj na kartce papieru przykład grafu ważonego. Wykorzystaj część sieci dróg poprowadzonych między miejscowością, w której mieszkasz, a kilkoma sąsiednimi miejscowościami.

Graf nieskierowany

Graf skierowany

Graf nieskierowany to graf, w którym przejście wzdłuż krawędzi jest możliwe w obu kierunkach. Przykładami grafów nieskierowanych są grafy na rysunkach 4.1 i 4.2. **Graf skierowany** to graf, w którym po krawędzi można przejść w określonym kierunku. Rysunek 4.3 przedstawia przykład grafu skierowanego. Kierunek, w którym można przejść po krawędzi, oznaczony jest strzałką.



Rys. 4.3. Przykład grafu skierowanego

Graf spójny

Graf, w którym istnieje droga z każdego wierzchołka do każdego, nazywamy **grafem spójnym**. Grafy przedstawione na rysunkach 4.1, 4.2 i 4.3 są spójne.

Zapamiętaj

Graf składa się z niepustego zbioru wierzchołków i zbioru krawędzi, czyli połączeń między wierzchołkami. W grafach nieskierowanych przejście między dwoma wierzchołkami połączonymi krawędzią jest możliwe w obu kierunkach. W grafach skierowanych między wierzchołkami można przechodzić tylko zgodnie z zaznaczonym kierunkiem. Jeśli krawędziom grafu przyporządkowane są wagi, to taki graf nazywamy grafem ważonym. Graf spójny to graf, w którym istnieje droga z każdego wierzchołka do każdego.

4.2. Reprezentacja grafu

Do reprezentacji grafu wykorzystuje się różne struktury danych i różne sposoby. Część z nich opiera się na rozwiązaniach, które już znasz. Pokażemy m.in., jak wykorzystać do reprezentacji grafu typ **vector** z biblioteki STL.

Macierz sąsiedztwa

Do przechowywania informacji o grafie można wykorzystać tablicę dwuwymiarową $n \times n$, gdzie n oznacza liczbę wierzchołków grafu. Element `(i,j)` tablicy informuje, czy istnieje krawędź od wierzchołka `i` do wierzchołka `j`. Gdy graf nie jest grafem ważonym, informacja, czy krawędź między wierzchołkami istnieje czy nie, może być informacją logiczną (wartości typu **bool**: **true** i **false** lub typu **int**: `0` i `1`). Dla grafu ważonego z dodatnimi wagami elementem tablicy może być waga krawędzi oraz wartość 0, gdy krawędź nie istnieje. Jeśli graf jest nieskierowany, to wartości elementów `(i,j)` oraz `(j,i)` są takie same. Jeśli graf jest skierowany, a przejście między wierzchołkami `i` oraz `j` jest możliwe tylko w jednym kierunku, to wartość jednego z elementów `(i,j)` lub `(j,i)` będzie równa 0.

	0	1	2	3	4
0	0	3	0	3	0
1	0	0	3	0	0
2	2	0	0	0	0
3	0	6	0	0	1
4	0	5	1	0	0

Rys. 4.4. Reprezentacja grafu z rysunku 4.3 w postaci macierzy sąsiedztwa

Macierz sąsiedztwa

Opisaną powyżej reprezentację grafu nazywamy **macierzą sąsiedztwa**. Rysunek 4.4 przedstawia przykład reprezentacji grafu z rysunku 4.3 ze s. 63 w postaci macierzy sąsiedztwa.

Warto wiedzieć

Z macierzy sąsiedztwa można odczytać wiele informacji dotyczących grafu, np. które wierzchołki sąsiadują z danym wierzchołkiem.

Ćwiczenie 2

Zapisz w postaci macierzy sąsiedztwa reprezentacje grafów z rysunków 4.1 i 4.2 ze s. 62 i 63.

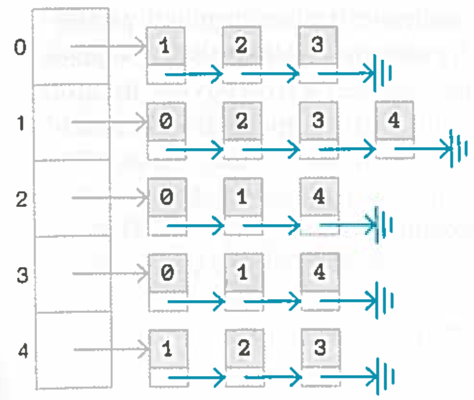
Na rysunku 4.4 widać, że wiele elementów tablicy jest niewykorzystanych (wypełnionych zerami, ponieważ krawędzie między danymi wierzchołkami nie istnieją). Złożoność pamięciowa i złożoność czasowa algorytmów operujących na takiej reprezentacji grafu wynosi co najmniej $O(n^2)$.

Listy sąsiedztwa

Listy sąsiedztwa

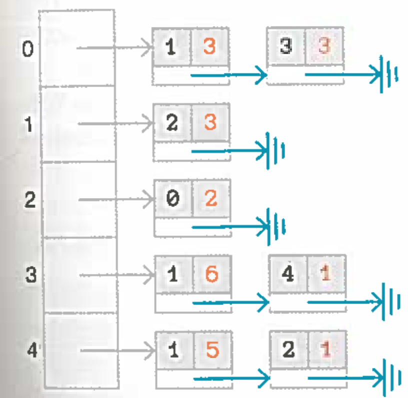
Inną formą reprezentacji grafu są **listy sąsiedztwa**. W tym przypadku dla każdego wierzchołka pamięta się listę sąsiadujących z nim wierzchołków (połączonych krawędzią z tym wierzchołkiem). Jeśli graf nie jest grafem ważonym, elementami danej listy będą numery sąsiednich wierzchołków.

Rysunek 4.5 przedstawia reprezentację grafu z rysunku 4.1 ze s. 62 w postaci list sąsiedztwa.



Rys. 4.5. Reprezentacja grafu z rysunku 4.1 w postaci list sąsiedztwa

W przypadku grafu ważonego dla każdego wierzchołka należy pamiętać dwie wartości: numer sąsiedniego wierzchołka oraz wagę krawędzi, która do niego prowadzi. Rysunek 4.6 przedstawia reprezentację grafu z rysunku 4.3 ze s. 63 w postaci list sąsiedztwa. Wagi krawędzi są zaznaczone na pomarańczowo.



Rys. 4.6. Reprezentacja grafu z rysunku 4.3 w postaci list sąsiedztwa

Pierwsze algorytmy związane z grafami, którymi będziemy się zajmować, będą dotyczyć grafu nieważonego. Dlatego omówimy najpierw możliwości reprezentacji takiego właśnie grafu w postaci list sąsiedztwa.

W języku C++ listy sąsiedztwa dla grafu nieważonego można przedstawić w postaci tablicy n-elementowej, gdzie n jest liczbą wierzchołków grafu. Każdy element tej tablicy odpowiada jednemu z wierzchołków i jest **listą** zawierającą numery wierzchołków sąsiadujących z tym wierzchołkiem. Przykład deklaracji tablicy `Graf`, służącej do przechowywania list sąsiedztwa dla grafu z pięcioma wierzchołkami, wygląda następująco: Lista, s. 47

```
const int N=5;
list<int> Graf[N];
```

Przedstawimy jeszcze jedną możliwość deklaracji grafu w postaci list sąsiedztwa. Wykorzystuje ona typ **vector** z biblioteki STL. Do przeglądania elementów listy nie musimy wówczas wykorzystywać **iteratora**.

Iterator, s. 51

Tablica dynamiczna

Typ `vector` jest w pewnym sensie odpowiednikiem **tablicy dynamicznej** (ang. *dynamic array*), czyli takiej, której rozmiar (liczbę elementów) można określać i zmieniać podczas działania programu. Do zmiennej typu **vector**, podobnie jak do zmiennej typu **list**, można wstawić w dowolne miejsce nowy element. Można też usunąć z niej dowolny element (zmieniając w ten sposób także rozmiar). Ogólna **deklaracja zmiennej typu vector** jest następująca:

Deklaracja zmiennej typu vector

```
vector<typ elementów wektora> nazwa_wektora;
```

Żeby korzystać z typu **vector**, należy za pomocą dyrektywy `#include <vector>` dołączyć do programu bibliotekę vector.

Do elementu typu **vector** odwołujemy się tak samo jak do elementu w tablicy – podając w nawiasach kwadratowych indeks elementu. Rozmiar zmiennej typu **vector** można ustalić za pomocą **metody resize**, której parametrem jest liczba elementów. W programie można jej użyć wielokrotnie i w ten sposób zwiększać lub zmniejszać liczbę elementów. Metoda **resize** może także mieć drugi parametr, który określa wartość początkową elementów zmiennej typu **vector**. Aktualny rozmiar tej zmiennej zwraca bezparametrowa **metoda size**.

Metoda resize dla klasy vector

Metoda size dla klasy vector

Poniższy fragment kodu źródłowego programu tworzy zmienną Tab typu **vector** o elementach całkowitych, następnie wczytuje z klawiatury rozmiar zmiennej i wypełnia ją losowymi liczbami.

Fragment kodu źródłowego programu tworzącego zmienną typu vector, której elementy są losowymi liczbami

```
vector<int> Tab;
int n;
cin>>n;
Tab.resize(n);
for (int i=0;i<Tab.size();i++)
    Tab[i]=rand()/100;
```

Dobra rada

Wstawianie elementów do struktury danych typu **vector** oraz usuwanie ich z tej struktury wymaga często przesunięcia w pamięci całej struktury danych (alokacji przydzielonej pamięci). Zwiększa to czas wykonywania tych operacji. Jeśli w programie, który piszesz, elementy będą często wstawiane bądź usuwane, lepiej użyj typu **list**.

Ćwiczenie 3

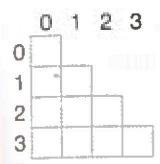
Napisz program, który utworzy zmienną typu **vector**, wczyta jej rozmiar z klawiatury, a następnie wypełni tę zmienną losowymi liczbami całkowitymi i ją wypisze.

Elementem zmiennej typu **vector** może być kolejna zmienna typu **vector**. Otrzymamy w ten sposób odpowiednik tablicy dwuwymiarowej, w której każdy wiersz może mieć różną liczbę elementów. Do poszczególnych elementów takiej zmiennej odwołujemy się jak do elementów tablicy dwuwymiarowej, podając w drugiej parze nawiasów kwadratowych drugi indeks.

Poniższy fragment kodu źródłowego programu dla $n = 4$ utworzy tablicę Tab przedstawioną na rysunku 4.7.

Fragment kodu źródłowego programu tworzącego tablicę o różnej liczbie elementów w wierszach

```
vector<vector<int> > Tab;
int n;
cin>>n;
Tab.resize(n);
for (int i=0;i<Tab.size();i++)
    Tab[i].resize(i+1);
```



Rys. 4.7. Przykład tablicy o różnej liczbie elementów w wierszach

Warto wiedzieć

Elementy zmiennej typu **vector** można sortować za pomocą funkcji **sort** z biblioteki STL. Parametrami funkcji są wówczas wartości zwracane przez iteratory **begin** i **end**. Na przykład zapis funkcji sortującej wartości zmiennej Tab będzie wyglądał następująco:

```
sort(Tab.begin(),
    Tab.end());
```

Ćwiczenie 4

Napisz program, który utworzy tablicę taką jak na rysunku 4.7, wypełni ją losowymi liczbami całkowitymi i wypisze elementy tablicy.

Do reprezentacji grafu w postaci list sąsiedztwa można wykorzystać zmienną typu **vector**, której elementy są też typu **vector**. Oto deklaracja zmiennej `Graf` reprezentującej graf, w którym wierzchołki oznaczone są liczbami całkowitymi:

```
vector<vector<int> > Graf;
```

Można zdefiniować własną nazwę dla tak określonego typu. Jest to szczególnie przydatne, gdy zmienna reprezentująca graf będzie parametrem funkcji. Do **definiowania własnej nazwy typu zmiennych** służy słowo kluczowe **typedef**:

Definicja własnej nazwy typu zmiennych

```
typedef<definicja typu> nazwa_typu;
```

Definicja typu `tgraf` reprezentującego graf jako listy sąsiedztwa oraz deklaracja zmiennej `Graf` tego typu przyjmą postać:

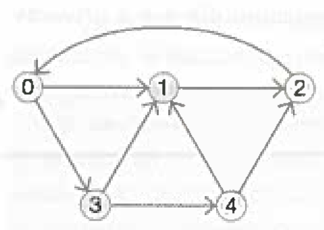
```
typedef vector<vector<int> > tgraf;
tgraf Graf;
```

Dobra rada

Dwa znaki > zapisane obok siebie kompilator może uznać za operator wczytywania danych. Dlatego przy deklaracji zmiennych z wykorzystaniem szablonów typów z biblioteki STL oddziel te znaki spacją.

Opis grafu wygodnie jest przygotować w pliku tekstowym. Pierwszy wiersz pliku będzie zawierał dwie liczby całkowite dodatnie określające odpowiednio liczbę wierzchołków grafu i liczbę krawędzi. Każdy kolejny wiersz będzie opisem jednej krawędzi w postaci numeru wierzchołka początkowego i numeru wierzchołka końcowego.

Rysunek 4.8 na s. 68 przedstawia przykład grafu skierowanego i jego opis w pliku tekstowym.




```
5 8
0 1
0 3
1 2
2 0
3 1
3 4
4 1
4 2
```

Rys. 4.8. Przykład grafu skierowanego i jego opis w pliku tekstowym

Kod źródłowy funkcji wczytującej opis grafu z pliku tekstowego i tworzącej zmienną `Graf` typu `tgraf` może być następujący:

Kod źródłowy funkcji `Czytaj`, wczytującej z pliku tekstowego opis grafu nieważonego

```
1. void Czytaj(tgraf &Graf)
2. {
3.     int n, m, w1, w2;
4.     ifstream we("graf.txt");
5.     we>>n>>m;
6.     Graf.resize(n);
7.     for (int i=0;i<m;i++)
8.     {
9.         we>>w1>>w2;
10.        Graf[w1].push_back(w2);
11.    }
12.    we.close();
13. }
```

W linii 5 z pliku tekstowego do zmiennych `n` i `m` są wczytywane informacje o liczbie wierzchołków i liczbie krawędzi grafu. Instrukcja w linii 6 tworzy graf o `n` wierzchołkach. W pętli w liniach 7–11 odczytywane są informacje o kolejnych krawędziach grafu (linia 9) i dodawane są one do struktury grafu (linia 10). W zmiennej `w1` zapisywany jest numer wierzchołka będącego początkiem krawędzi, a w zmiennej `w2` – numer wierzchołka będącego końcem krawędzi. Metoda `push_back` dodaje nowy element na końcu zmiennej typu `vector`.

Metoda `push_back` dla klasy `vector`

Zapamiętaj

Graf można reprezentować m.in. w postaci macierzy sąsiedztwa lub list sąsiedztwa. W przypadku macierzy sąsiedztwa informacje o grafie przechowuje się w tablicy dwuwymiarowej *n* × *n*, gdzie *n* oznacza liczbę wierzchołków grafu. Element (`i`,`j`) tablicy informuje, czy istnieje krawędź od wierzchołka `i` do `j`. W reprezentacji grafu za pomocą list sąsiedztwa dla każdego wierzchołka pamięta się listę krawędzi z niego wychodzących. Do implementacji list sąsiedztwa można użyć typu `vector`.

4.3. Przeszukiwanie grafu w głąb

Zajmiemy się teraz problemem przejrzenia w usystematyzowany sposób wszystkich wierzchołków grafu. Załóżymy, że **graf** jest **skierowany** i **spójny**. Dla grafu nieskierowanego krawędzie w opisie występowałyby dwukrotnie. Oto specyfikacja problemu:

Graf skierowany, graf spójny, s. 63

Specyfikacja

Dane: `n` – liczba wierzchołków grafu,
`m` – liczba krawędzi grafu,
`Graf` – reprezentacja spójnego grafu skierowanego w postaci list sąsiedztwa,
`w1` – numer wierzchołka, od którego zaczynamy przeglądanie grafu.

Wynik: numery kolejno odwiedzonych wierzchołków.

Algorytm przeglądania grafu, który teraz omówimy, nosi nazwę **przeszukiwania w głąb** (**DFS**, od ang. *depth-first search*). Opiera się na podobnej zasadzie co **rekurencyjny algorytm znajdowania wyjścia z labiryntu**. Zapiszemy go także rekurencyjnie.

Algorytm przeszukiwania w głąb (DFS)

Rekurencyjny algorytm znajdowania wyjścia z labiryntu, s. 31

Wprowadzimy dodatkową strukturę danych, w której zapiszemy informację, czy dany wierzchołek został już odwiedzony. Będzie to tablica wartości logicznych `Odwiedzone` o rozmiarze zgodnym z liczbą wierzchołków grafu. Gdybyśmy nie zaznaczyli, które wierzchołki odwiedziliśmy, moglibyśmy wielokrotnie sprawdzać sąsiadów tego samego wierzchołka (wywoływać funkcję rekurencyjną dla tych samych parametrów, co prowadziłoby do rekurencji nieskończonej).

Oznaczymy wierzchołek początkowy jako odwiedzony, a następnie dla sąsiadów tego wierzchołka będziemy wywoływać funkcję rekurencyjną, tak jakby dany sąsiad wierzchołka był wierzchołkiem początkowym. Wywołanie rekurencyjne nastąpi pod warunkiem, że dany wierzchołek nie został jeszcze odwiedzony. Zapis algorytmu w pseudokodzie może być następujący:

```
dla i ← 0, 1, …, n – 1 wykonuj Odwiedzone[i] ← fałsz
funkcja DFS(w1)
   wypisz w1
   Odwiedzone[w1] ← prawda
   dla i ← 0, 1, …, (liczba krawędzi w1) – 1 wykonuj
      w2 ← Graf[w1][i]
      jeśli nie Odwiedzone[w2] to DFS(w2)
```

Zwróć uwagę, że tablica `Odwiedzone` (pierwsza linia pseudokodu) musi być inicjowana poza funkcją rekurencyjną. Zmienna pomocnicza `w2` przechowuje numer wierzchołka będącego końcem analizowanej krawędzi. Jej wartość jest parametrem wywołania rekurencyjnego. Kod źródłowy funkcji DFS, realizującej algorytm przedstawiony w pseudokodzie, oraz funkcji `main` jest następujący:

Fragment kodu źródłowego programu realizującego algorytm przeszukiwania grafu w głąb – funkcja `DFS`, realizująca ten algorytm, oraz funkcja `main`

```
1.  void DFS(int w1, tgraf &Graf, vector<bool> &Odwiedzone)
2.  {
3.      cout<<"Odwiedzony wierzcholek: "<<w1<<endl;
4.      Odwiedzone[w1]=true;
5.      for (int i=0;i<Graf[w1].size();i++)
6.      {
7.          int w2=Graf[w1][i];
8.          if (!Odwiedzone[w2]) DFS(w2,Graf,Odwiedzone);
9.      }
10. }
11.
12. int main()
13. {
14.     tgraf Graf;
15.     Czytaj(Graf);
16.     vector<bool> Odwiedzone;
17.     Odwiedzone.resize(Graf.size(),false);
18.     int w1;
19.     cout<<"Podaj numer wierzcholka poczatkowego: ";
20.     cin>>w1;
21.     DFS(w1,Graf,Odwiedzone);
22.     return 0;
23. }
```

Funkcja DFS nie zmienia wartości parametru Graf, ale jest on przekazywany przez referencję, aby nie była tworzona jego kopia przy wywołaniach rekurencyjnych. Parametr Odwiedzone musi być przekazany przez referencję, gdyż funkcja modyfikuje jego wartość. W linii 15 wywoływana jest funkcja `Czytaj`, wczytująca z pliku tekstowego opis grafu. W linii 16 deklarowana jest tablica `Odwiedzone`. Instrukcja w linii 17 tworzy tablicę Odwiedzone o rozmiarze zgodnym z liczbą wierzchołków grafu oraz nadaje jej elementom wartość początkową **false** (drugi parametr). W linii 20 wczytywany jest z klawiatury numer wierzchołka (zmienna w1), od którego chcemy rozpocząć przeglądanie grafu. W linii 21 wywołujemy funkcję DFS.

Kod źródłowy funkcji `Czytaj`, wczytującej z pliku tekstowego opis grafu nieważonego, s. 68

Rysunek 4.9 przedstawia przykład wykonania programu przeglądającego w głąb graf z rysunku 4.8 ze s. 68, zaczynając od wierzchołka numer 3.

```
Podaj numer wierzcholka poczatkowego: 3
Odwiedzony wierzcholek: 3
Odwiedzony wierzcholek: 1
Odwiedzony wierzcholek: 2
Odwiedzony wierzcholek: 0
Odwiedzony wierzcholek: 4
```

Rys. 4.9. Przykład wykonania programu przeglądającego w głąb graf z rysunku 4.8 od wierzchołka numer 3

Z wierzchołka oznaczonego liczbą 3 wychodzą dwie krawędzie, które prowadzą do wierzchołków 1 i 4. Najpierw został odwiedzony wierzchołek numer 1, potem jego sąsiedzi, a na końcu nastąpił powrót do wierzchołka początkowego i został odwiedzony wierzchołek numer 4.

Ćwiczenie 5

Napisz program, który wczyta opis spójnego grafu skierowanego z pliku tekstowego otrzymanego od nauczyciela (np. *graf.txt*), a następnie przejrzy wierzchołki grafu, wykorzystując algorytm przeglądania w głąb, i wypisze kolejno odwiedzone wierzchołki.

Zapamiętaj

Algorytm przeszukiwania w głąb przegląda wszystkie wierzchołki grafu spójnego. Jeśli istnieje nieodwiedzony wierzchołek sąsiadujący z aktualnie rozpatrywanym wierzchołkiem, to algorytm przechodzi do niego i ten wierzchołek staje się aktualnie rozpatrywanym. Jeśli taki sąsiedni wierzchołek nie istnieje, algorytm cofa się do poprzednio rozpatrywanego i powtarza czynności dla kolejnego sąsiedniego wierzchołka. Algorytm można zapisać rekurencyjnie.

4.4. Przeszukiwanie grafu wszerz

Algorytm przeszukiwania wszerz (BFS)

Omówimy teraz **algorytm przeszukiwania wszerz** (**BFS**, od ang. *breadth-first search*). Jest on podobny do **iteracyjnego algorytmu znajdującego wyjście z labiryntu** z wykorzystaniem kolejki.

Iteracyjny algorytm znajdowania wyjścia z labiryntu, s. 40

Wierzchołek, od którego zaczniemy analizę, zostanie wstawiony do kolejki i oznaczony w tablicy Odwiedzone jako odwiedzony. Następnie, dopóki kolejka nie będzie pusta, pobieramy wierzchołek z początku kolejki i analizujemy wierzchołki z nim sąsiadujące. Jeśli sąsiedni wierzchołek nie został wcześniej odwiedzony, to jest dopisywany na końcu kolejki i oznaczany jako odwiedzony. Zapis algorytmu w pseudokodzie może być następujący:

```
dla i ← 0, 1, ..., n − 1 wykonuj Odwiedzone[i] ← fałsz
wstaw w1 do kolejki
Odwiedzone[w1] ← prawda
dopóki nie pusta kolejka wykonuj
    w1 ← numer wierzchołka z początku kolejki
    usuń wierzchołek z kolejki
    wypisz w1
    dla i ← 0, 1, ..., (liczba krawędzi z w1) − 1 wykonuj
        w2 ← Graf[w1][i]
        jeśli nie Odwiedzone[w2] to
            wstaw w2 do kolejki
            Odwiedzone[w2] ← prawda
```

Kod źródłowy funkcji BFS, realizującej algorytm przeszukiwania grafu wszerz, może wyglądać następująco.

Kod źródłowy funkcji realizującej algorytm przeszukiwania grafu wszerz

```
1.  void BFS(int w1, tgraf Graf)
2.  {
3.      int w2;
4.      vector<bool> Odwiedzone;
5.      Odwiedzone.resize(Graf.size(),false);
6.      queue<int> wierzch;
7.      wierzch.push(w1); Odwiedzone[w1]=true;
8.      while (!wierzch.empty())
9.      {
10.         w1=wierzch.front(); wierzch.pop();
11.         cout<<"Odwiedzony wierzcholek: "<<w1<<endl;
12.         for (int i=0;i<Graf[w1].size();i++)
13.         {
14.             w2=Graf[w1][i];
15.             if (!Odwiedzone[w2])
16.             {
17.                 wierzch.push(w2); Odwiedzone[w2]=true;
18.             }
19.         }
20.     }
21. }
```

Dobra rada

Pamiętaj, że aby korzystać ze zmiennych typu **queue**, do programu musisz dołączyć bibliotekę **queue**.

W linii 4 jest deklarowana tablica `Odwiedzone`, a w linii 5 jest ona tworzona. Ponieważ funkcja `BFS` nie jest rekurencyjna, tablica `Odwiedzone` może być w tej funkcji zmienną lokalną. W linii 6 deklarowana jest kolejka `wierzch`, w której będą pamiętane numery wierzchołków do analizy. W linii 7 wierzchołek, od którego rozpoczniemy przeglądanie grafu, jest wstawiany do kolejki i oznaczany w tablicy `Odwiedzone` jako odwiedzony. Pętla w liniach 8–20 przegląda kolejkę, wykonując następujące operacje:

1. Pobiera pierwszy element kolejki i go usuwa. Numer wierzchołka pobranego z kolejki jest pamiętany w zmiennej `w1` (linia 10).
2. Wypisuje numer analizowanego wierzchołka (linia 11).
3. Rozpatruje sąsiednie wierzchołki (pętla w liniach 12–19). Najpierw pobiera numer sąsiedniego wierzchołka do zmiennej `w2` (linia 14). Jeśli wierzchołek nie został jeszcze odwiedzony (linia 15), to wstawia jego numer do kolejki i oznacza go jako odwiedzony (linia 17).

W programie realizującym przeglądanie grafu wszerz pozostałe funkcje (funkcja `Czytaj`, funkcja `main`) i deklaracje (poza tablicą `Odwiedzone`) wyglądają podobnie jak w programie przeglądającym graf w głąb.

Rysunek 4.10 przedstawia przykład wykonania programu przeglądającego wszerz graf z rysunku 4.8 ze s. 68, zaczynając od wierzchołka numer 3. Z tego wierzchołka wychodzą dwie krawędzie. Najpierw zostały odwiedzone wierzchołki numer 1 i 4, potem ich sąsiedzi.


```
Podaj numer wierzcholka poczatkowego: 3
Odwiedzony wierzcholek: 3
Odwiedzony wierzcholek: 1
Odwiedzony wierzcholek: 4
Odwiedzony wierzcholek: 2
Odwiedzony wierzcholek: 0
```

Rys. 4.10. Przykład wykonania programu przeglądającego wszerz graf z rysunku 4.8 od wierzchołka numer 3

Warto wiedzieć

Złożoność pamięciowa i czasowa algorytmów przeglądania grafu wszerz i w głąb wynosi $O(n + m)$, gdzie n określa liczbę wierzchołków, a m – liczbę krawędzi.

Ćwiczenie 6

Napisz program wczytujący opis grafu z pliku tekstowego otrzymanego od nauczyciela (np. *graf.txt*) i przeglądający wierzchołki grafu wszerz. Program powinien wypisać kolejno odwiedzane wierzchołki.

Zapamiętaj

Algorytm przeszukiwania wszerz przegląda wszystkie wierzchołki grafu spójnego. Najpierw odwiedza wszystkie wierzchołki sąsiadujące z danym wierzchołkiem, potem wszystkie nieodwiedzone jeszcze wierzchołki sąsiadujące z wierzchołkami sąsiednimi itd.

4.5. Algorytm Dijkstry

Rozważymy problem znajdowania drogi pomiędzy dwoma wierzchołkami w grafie ważonym, tak aby suma wag krawędzi, po których przejdziemy, była jak najmniejsza. Problem odpowiada znalezieniu najkrótszej drogi, jeśli wagi krawędzi będą odległościami między wierzchołkami, albo najszybszej drogi, jeśli wagi będą oznaczały czas przejścia. Łączną długość drogi lub łączny czas połączenia (sumę wag krawędzi) nazwiemy kosztem drogi, a drogę o najmniejszym koszcie – najkrótszą.

Algorytm Dijkstry

Rozwiązanie problemu dla grafu ważonego o nieujemnych wagach krawędzi umożliwia **algorytm Dijkstry**. Wyznacza on najmniejsze koszty dojścia z wybranego wierzchołka grafu do pozostałych wierzchołków oraz wskazuje, przez które wierzchołki prowadzą drogi o najmniejszych kosztach. Na razie skupimy się na sposobie wyznaczania najmniejszych kosztów dojścia.

Najpierw algorytm przypisuje wierzchołkom grafu początkowe koszty dojścia do nich: wierzchołkowi, z którego wychodzimy – koszt 0, pozostałym – wartość ∞ (nieskończoność). W każdym kroku z wierzchołków jeszcze nieodwiedzonych wybiera ten, który ma aktualnie najmniejszy koszt dojścia i oznacza go jako odwiedzony. Sprawdza koszty dojścia z tego wierzchołka do nieodwiedzonych wierzchołków sąsiadujących. Jeśli są mniejsze niż znalezione do tej pory, aktualizuje je.

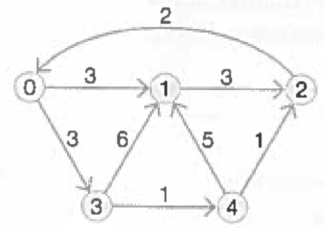
Warto wiedzieć

Algorytm Dijkstry opiera się na metodzie zachłannej, która polega na realizowaniu przez algorytm najlepszego wyboru w danym kroku rozwiązywania problemu. O metodzie tej można przeczytać w podręczniku *Informatyka na czasie 2. Zakres rozszerzony* (s. 255).

Algorytm Dijkstry – wyznaczanie najmniejszych kosztów dojścia do wierzchołków grafu

Prześledzimy działanie algorytmu dla skierowanego grafu ważonego o nieujemnych wagach krawędzi, w którym wierzchołki są oznaczone liczbami całkowitymi od 0 do 4. Wyznaczymy najmniejsze koszty dojścia z wierzchołka 0 do pozostałych wierzchołków. W każdym kroku na grafie kolorem niebieskim zaznaczony jest aktualnie rozpatrywany wierzchołek (oznaczany w danym kroku jako odwiedzony) wraz z krawędziami z niego wychodzącymi, a zielonym – wierzchołki odwiedzone w poprzednich krokach. Niebieski kolor w tabeli wskazuje zmianę w stosunku do poprzedniego kroku.

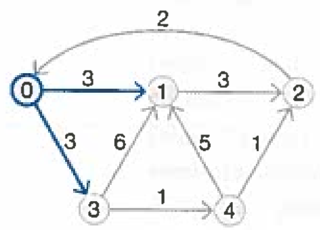
Graf oraz tabela z kosztami dojścia do poszczególnych wierzchołków na początku algorytmu wyglądają następująco:



Numer wierzchołka	0	1	2	3	4
Koszt dotarcia do wierzchołka	0	∞	∞	∞	∞

Krok 1

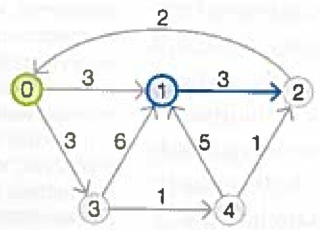
Żaden wierzchołek nie został jeszcze odwiedzony. Najniższy koszt ma wierzchołek 0, zatem od niego zaczynamy poszukiwanie najkrótszych dróg i oznaczamy go jako odwiedzony. Wychodzą z niego dwie krawędzie. Prowadzą one do wierzchołków o numerach 1 i 3, obie mają wagę 3. W tabeli zmieniamy koszty dotarcia do wierzchołków 1 i 3.



Numer wierzchołka	0	1	2	3	4
Koszt dotarcia do wierzchołka	0	3	∞	3	∞

Krok 2

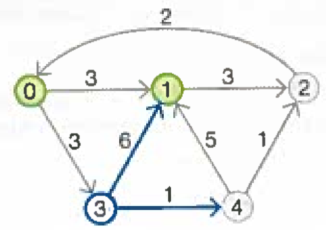
Najmniejszy koszt z nieodwiedzonych wierzchołków (1, 2, 3, 4) mają wierzchołki 1 i 3. Wybieramy dowolny z nich, np. wierzchołek 1, i oznaczamy go jako odwiedzony. Wychodzi z niego krawędź do wierzchołka 2. Koszt dojścia do wierzchołka 2 wynosi 6 i jest sumą kosztu dojścia do wierzchołka 1 oraz wagi krawędzi prowadzącej od wierzchołka 1 do 2 (3 + 3). Aktualizujemy w tabeli koszt dojścia do wierzchołka 2.



Numer wierzchołka	0	1	2	3	4
Koszt dotarcia do wierzchołka	0	3	6	3	∞

Krok 3

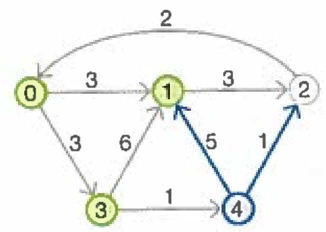
Spośród nieodwiedzonych wierzchołków (2, 3, 4) wybieramy ten o najmniejszym koszcie dotarcia – wierzchołek 3 – i oznaczamy go jako odwiedzony. Można z niego dojść do wierzchołków 1 oraz 4. Wierzchołek 1 został już odwiedzony, więc go nie rozpatrujemy. Koszt dotarcia do wierzchołka 4 przez wierzchołek 3 jest równy 4 (3 + 1). Zapisujemy tę wartość w tabeli.



Numer wierzchołka	0	1	2	3	4
Koszt dotarcia do wierzchołka	0	3	6	3	4

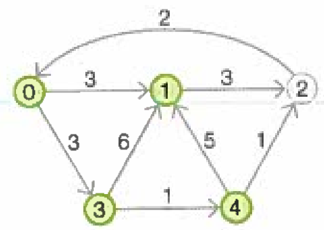
Krok 4

Z nieodwiedzonych wierzchołków (2 i 4) wybieramy ten o najmniejszym koszcie dotarcia – wierzchołek 4 – i oznaczamy go jako odwiedzony. Można z niego dojść do wierzchołków 1 i 2. Wierzchołek 1 został już odwiedzony, więc go nie rozpatrujemy. Koszt dotarcia do wierzchołka 2 jest równy 5 (4 + 1). Jest mniejszy od dotychczas znalezionego (6), dlatego zmieniamy wartość w tabeli.



Numer wierzchołka	0	1	2	3	4
Koszt dotarcia do wierzchołka	0	3	5	3	4

Algorytm zakończył działanie, ponieważ pozostał jeden nieodwiedzony wierzchołek (2), z którego nie może już wychodzić żadna krawędź do nieodwiedzonego wierzchołka.



Wynikiem działania algorytmu są koszty dotarcia z wierzchołka 0 do pozostałych wierzchołków przedstawione w poniższej tabeli.

Numer wierzchołka	0	1	2	3	4
Koszt dotarcia do wierzchołka	0	3	5	3	4

Specyfikacja problemu znajdowania najmniejszych kosztów dojścia od danego wierzchołka do pozostałych wierzchołków grafu wygląda następująco:

Specyfikacja

Dane: n – liczba wierzchołków grafu,
m – liczba krawędzi grafu,
Graf – reprezentacja skierowanego grafu ważonego o nieujemnych wagach krawędzi w postaci list sąsiedztwa,
pocz – numer wierzchołka początkowego.

Wynik: Koszt[n] – tablica najmniejszych kosztów dojścia z wierzchołka pocz do pozostałych wierzchołków.

Na początku dla każdego wierzchołka koszt dojścia ustawimy na nieskończoność (w programie będzie to duża wartość przekraczająca możliwy koszt drogi). Dla wierzchołka pocz koszt wynosi 0. Algorytm polega na przeglądaniu wierzchołków. Za każdym razem wybieramy wierzchołek jeszcze nieodwiedzony o najmniejszym dotychczas znalezionym koszcie dojścia i oznaczamy go jako odwiedzony. Następnie rozpatrujemy krawędzie wychodzące z tego wierzchołka i aktualizujemy koszty dotarcia do sąsiednich nieodwiedzonych jeszcze wierzchołków, o ile są niższe.

Oto zapis algorytmu w pseudokodzie:

Warto wiedzieć
Planszę z labiryntem z tematu 2 można reprezentować w postaci grafu. Pola, na których można stanąć, odpowiadają wierzchołkom. Jeśli na dwóch sąsiednich polach można stanąć, to otrzymujemy krawędź pomiędzy wierzchołkami. Odległość (waga krawędzi) pomiędzy dwoma sąsiednimi wierzchołkami jest dla każdej krawędzi taka sama (jednostkowa).

```
dla i ← 0, 1, ..., n - 1 wykonuj
    Odwiedzone[i] ← fałsz
    Koszt[i] ← ∞
Koszt[pocz] ← 0
dla i ← 0, 1, ..., n - 2 wykonuj
    w1 ← numer wierzchołka nieodwiedzonego
         o najmniejszym koszcie
    Odwiedzone[w1] ← prawda
    dla j ← 0, 1, ..., (liczba krawędzi z w1) - 1 wykonuj
        w2 ← Graf[w1][j]
        waga ← waga krawędzi od w1 do w2
        jeśli nie Odwiedzone[w2] oraz
              Koszt[w1] + waga < Koszt[w2] to
            Koszt[w2] ← Koszt[w1] + waga
```

Implementując powyższy algorytm, musimy uwzględnić wagę krawędzi. Krawędź będzie więc reprezentowana przez dwie liczby: numer wierzchołka i wagę krawędzi. Wygodnie będzie zdefiniować krawędź jako strukturę. Wówczas graf w postaci list sąsiedztwa możemy reprezentować jako zmienną typu **vector** o elementach typu **vector**, które przechowują informacje o krawędziach.

Definicja struktury przechowującej informację o krawędzi oraz definicja grafu wykorzystująca tę strukturę mogą wyglądać następująco.

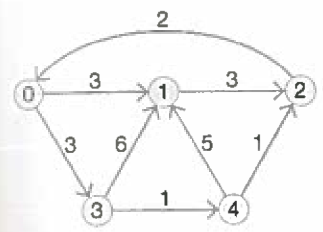

```
struct krawedz
{
    int w2;      // numer wierzchołka będącego
                 // końcem krawędzi
    int waga;    // waga krawędzi
};

typedef vector<vector<krawedz> > tgraf;
```

Definicja struktury przechowującej informacje o krawędziach skierowanego grafu ważonego oraz definicja grafu wykorzystująca tę strukturę

Opis skierowanego grafu ważonego w pliku tekstowym będzie wyglądał podobnie jak opis grafu, w którym krawędziom nie przypisano wag (rys. 4.8, s. 68). W każdym wierszu reprezentującym krawędź dodamy trzecią liczbę określającą wagę.

Przykład skierowanego grafu ważonego oraz jego opis w pliku tekstowym przedstawia rysunek 4.11.




```
5 8
0 1 3
0 3 3
1 2 3
2 0 2
3 1 6
3 4 1
4 1 5
4 2 1
```

Rys. 4.11. Przykład skierowanego grafu ważonego i jego opis w pliku tekstowym

Kod źródłowy zmodyfikowanej funkcji Czytaj, która wczyta dane o grafie ważonym z pliku tekstowego, oraz kod źródłowy funkcji realizującej algorytm Dijkstry są takie jak na s. 78.

A to ciekawe

Mosty królewieckie

Czy można narysować kopertę bez odrywania ołówka od kartki tak, aby nie rysować dwa razy tego samego odcinka? Ta i wiele podobnych zagadek sprowadza się do problemów związanych z grafami. Jednym z nich jest słynny problem mostów królewieckich. W osiemnastowiecznym Królewcu (dzisiejszy Kaliningrad) było aż siedem mostów łączących różne części miasta (zdjęcie obok przedstawia jeden z mostów w 1920 r.). Pytanie brzmiało: czy można przejść przez wszystkie mosty tak, aby każdy odwiedzić tylko raz i wrócić do miejsca, z którego się wyruszyło? W 1736 r. szwajcarski matematyk Leonhard Euler wykazał, że jest to niemożliwe. Jego praca dała początek teorii grafów – działowi matematyki zajmującemu się badaniem ich własności.

Kod źródłowy funkcji Czytaj, wczytującej z pliku tekstowego opis grafu ważonego

```
1.  void Czytaj(tgraf &Graf)
2.  {
3.      int n, m, w1;
4.      krawedz kraw;
5.      ifstream we("graf_1.txt");
6.      we>>n>>m;
7.      Graf.resize(n);
8.      for (int i=0;i<m;i++)
9.      {
10.         we>>w1>>kraw.w2>>kraw.waga;
11.         Graf[w1].push_back(kraw);
12.     }
13.     we.close();
14. }
```

Kod źródłowy funkcji realizującej algorytm Dijkstry

```
1.  void Dijkstra(tgraf Graf, int pocz, vector<int> &Koszt)
2.  {
3.      krawedz kraw;
4.      int i, j, k, w1;
5.      vector<bool> Odwiedzone;
6.      Odwiedzone.resize(Graf.size(),false);
7.      Koszt[pocz]=0;
8.      for (i=0;i<Graf.size()-1;i++)
9.      {
10.         k=0; while (Odwiedzone[k]) k++; w1=k;
11.         for (j=k+1;j<Graf.size();j++)
12.             if (!Odwiedzone[j] && Koszt[j]<Koszt[w1]) w1=j;
13.         Odwiedzone[w1]=true;
14.         for (j=0;j<Graf[w1].size();j++)
15.         {
16.             kraw=Graf[w1][j];
17.             if (!Odwiedzone[kraw.w2] &&
18.                 Koszt[w1]+kraw.waga<Koszt[kraw.w2])
19.                 Koszt[kraw.w2]=Koszt[w1]+kraw.waga;
20.         }
21.     }
22. }
```

Pętla w liniach 8–21 przegląda wierzchołki grafu. Jednym z kluczowych elementów algorytmu jest szukanie nieodwiedzonego wierzchołka o najmniejszym dotychczas znalezionym koszcie. Wykonują to instrukcje w liniach 10–12. Najpierw znajdowany jest nieodwiedzony wierzchołek o najmniejszym numerze (linia 10) – wartość początkowa zmiennej `w1`. Następnie w pętli przeglądamy tablicę kosztów w poszukiwaniu nieodwiedzonego wierzchołka o mniejszym koszcie (linie 11–12). Pętla w liniach 14–20 przegląda krawędzie wychodzące ze znalezionego wierzchołka i modyfikuje koszt jego nieodwiedzonych sąsiadów, jeśli przejście po danej krawędzi stanowi krótszą drogę.

Kod źródłowy funkcji `main` może być następujący:

Fragment kodu źródłowego programu obliczającego koszty najkrótszych dróg od danego wierzchołka do pozostałych wierzchołków grafu – funkcja main

```
1.  int main()
2.  {
3.      tgraf Graf;
4.      Czytaj(Graf);
5.      vector<int> Koszt;
6.      Koszt.resize(Graf.size(),1000);
7.      int pocz;
8.      cout<<"Numer wierzcholka poczatkowego: ";
9.      cin>>pocz;
10.     Dijkstra(Graf,pocz,Koszt);
11.     cout<<"Koszt dojscia z wierzcholka "<<pocz;
12.     cout<<" do wierzcholka:"<<endl;
13.     for (int i=0;i<Graf.size();i++)
14.         if (i!=pocz) cout<<i<<": "<<Koszt[i]<<endl;
15.     return 0;
16. }
```

Zwróć uwagę na linię 6. Tworzona jest w niej tablica kosztów. Każdemu elementowi tej tablicy przypisana jest wartość 1000, czyli wartość wielokrotnie większa od kosztu potencjalnej drogi. Pętla w liniach 13–14 wypisuje wartości tablicy `Koszt` z wyjątkiem kosztu wierzchołka początkowego, czyli najmniejsze koszty dojścia do poszczególnych wierzchołków. Jeśli droga dojścia do jakiegoś wierzchołka nie istnieje, to zostanie wypisana wartość początkowa 1000.

Rysunek 4.12 przedstawia efekt wykonania programu dla grafu z rysunku 4.11 ze s. 77, jeśli wierzchołkiem początkowym jest wierzchołek o numerze 0.

```
Numer wierzcholka poczatkowego: 0
Koszt dojscia z wierzcholka 0 do wierzcholka:
1: 3
2: 5
3: 3
4: 4
```

Rys. 4.12. Efekt wykonania programu dla grafu z rysunku 4.11 i wierzchołka początkowego o numerze 0

Ćwiczenie 7

Napisz program wczytujący opis grafu ważonego z pliku tekstowego, który otrzymasz od nauczyciela (np. *graf_1.txt*), i realizujący algorytm, który znajduje najniższe koszty dojścia z wierzchołka początkowego do pozostałych wierzchołków.

Złożoność obliczeniowa przedstawionej implementacji algorytmu znajdowania najmniejszych kosztów jest kwadratowa względem liczby wierzchołków. Przeglądamy $n - 1$ wierzchołków, a wyszukanie wierzchołka o najmniejszym koszcie odbywa się liniowo w pętli przeglądającej wszystkie wierzchołki. Złożoność obliczeniową można zredukować do liniowo-logarytmicznej, usprawniając metodę wyszukania kolejnego wierzchołka do rozpatrzenia. Wymaga to użycia **kolejki priorytetowej**. Jest to kolejka, w której elementy są uporządkowane według określonego kryterium. Koszt wstawienia elementu do takiej kolejki (a także koszt usunięcia elementu) jest logarytmiczny. Implementację tej wersji algorytmu pomijamy.

Kolejka priorytetowa

Pozostaje jeszcze zmodyfikować algorytm tak, aby oprócz znajdowania najmniejszych kosztów dojścia z wierzchołka początkowego do pozostałych wskazywał drogę do wybranego wierzchołka końcowego. Oto zmodyfikowana specyfikacja problemu:

Specyfikacja

Dane: n – liczba wierzchołków grafu,
m – liczba krawędzi grafu,
Graf – reprezentacja skierowanego grafu ważonego o nieujemnych wagach krawędzi w postaci list sąsiedztwa,
pocz – numer wierzchołka początkowego,
kon – numer wierzchołka końcowego.

Wynik: Koszt[n] – tablica najmniejszych kosztów dojścia z wierzchołka pocz do pozostałych wierzchołków,
numery wierzchołków, przez które należy przejść od pocz do kon z kosztem Koszt[kon].

Zapis zmodyfikowanego algorytmu w pseudokodzie jest następujący:

```
dla i ← 0, 1, …, n - 1 wykonuj
    Odwiedzone[i] ← fałsz
    Koszt[i] ← ∞
    Poprz[i] ← -1
Koszt[pocz] ← 0
dla i ← 0, 1, …, n - 2 wykonuj
    w1 ← numer wierzchołka nieodwiedzonego
         o najmniejszym koszcie
    Odwiedzone[w1] ← prawda
    dla j ← 0, 1,…, (liczba krawędzi z w1) - 1 wykonuj
        w2 ← Graf[w1][j]
        waga ← waga krawędzi od w1 do w2
        jeśli nie Odwiedzone[w2] oraz
              Koszt[w1] + waga < Koszt[w2] to
            Koszt[w2] ← Koszt[w1] + waga
            Poprz[w2] ← w1
```


W pseudokodzie użyliśmy dodatkowej tablicy o nazwie Poprz, w której dla każdego wierzchołka jest pamiętany numer wierzchołka bezpośrednio go poprzedzającego na drodze o najniższym koszcie. Elementom tablicy Poprz przypisujemy początkowe wartości –1. Oznaczają one, że na początku działania algorytmu wierzchołki nie mają wierzchołków poprzedzających. Podczas aktualizacji elementu tablicy Koszt aktualizujemy także wartość w tablicy Poprz.

Na podstawie informacji z tablicy Poprz, korzystając ze **stosu**, możemy wypisać drogę o najniższym koszcie od wierzchołka początkowego do końcowego.

Stos, s. 11

```
w ← kon
włóż w na stos
dopóki w ≠ pocz wykonuj
    w ← Poprz[w]
    włóż w na stos
dopóki nie pusty stos wykonuj
    wypisz wierzchołek stosu
    usuń element ze stosu
```

Kod źródłowy zmodyfikowanej funkcji main programu wyznaczającego drogę o najmniejszym koszcie może być następujący:

```
1.  int main()
2.  {
3.      tgraf Graf;
4.      Czytaj(Graf);
5.      vector<int> Koszt;
6.      Koszt.resize(Graf.size(),1000);
7.      vector<int> Poprz;
8.      Poprz.resize(Graf.size(),-1);
9.      int pocz, kon;
10.     cout<<"Numer wierzcholka poczatkowego: ";
11.     cin>>pocz;
12.     cout<<"Numer wierzcholka koncowego: ";
13.     cin>>kon;
14.     Dijkstra(Graf,pocz,Koszt,Poprz);
15.     cout<<"Koszt dojscia: "<<Koszt[kon]<<endl<<"Droga: ";
16.     WypiszDroge(pocz,kon,Poprz);
17.     return 0;
18. }
```

Fragment kodu źródłowego programu wyznaczającego drogę o najmniejszym koszcie między dwoma wierzchołkami – funkcja main

W linii 8 tworzona i inicjowana jest tablica Poprz. Zwróć uwagę na wywołanie funkcji Dijkstra (linia 14). Ma ona dodatkowy parametr – tablicę Poprz. Funkcja określa jej wartości, które są potem wykorzystywane w funkcji WypiszDroge (linia 16), realizującej algorytm przedstawiony w pseudokodzie. Dlatego parametr ten w funkcji Dijkstra przekazywany jest przez referencję, tak jak tablica Koszt.

Ćwiczenie 8

Napisz program, który wczyta opis skierowanego grafu ważonego z pliku tekstowego otrzymanego od nauczyciela (np. *graf_1.txt*), a następnie znajdzie i wypisze drogę o najmniejszym koszcie łączącą dwa podane wierzchołki.

Rysunek 4.13 przedstawia przykład wykonania programu dla grafu z rysunku 4.11 ze s. 77. Najmniejszy koszt dojścia z wierzchołka o numerze 0 do wierzchołka o numerze 2 wynosi 5. Droga o takim koszcie prowadzi przez wierzchołki 0, 3, 4, 2.

```
Numer wierzcholka poczatkowego: 0
Numer wierzcholka koncowego: 2
Koszt dojscia: 5
Droga: 0-->3-->4-->2
```

Rys. 4.13. Efekt wykonania programu dla grafu z rysunku 4.11 i wierzchołków o numerach 0 i 2

Zapamiętaj

Algorytm Dijkstry wyznacza najmniejsze koszty dojścia z wybranego wierzchołka grafu do pozostałych wierzchołków oraz wskazuje, przez które wierzchołki prowadzą drogi o najmniejszych kosztach. Można go stosować dla grafu ważonego o nieujemnych wagach krawędzi. Przeglądając wierzchołki grafu, wybieramy nieodwiedzony jeszcze wierzchołek grafu o najmniejszym dotychczas znalezionym koszcie dojścia i rozpatrujemy sąsiadujące z nim nieodwiedzone wierzchołki.

A to ciekawe

Jak powstał algorytm Dijkstry

Nazwa algorytmu pochodzi od nazwiska jego twórcy – holenderskiego informatyka Edsgera Dijkstry. Podczas pracy w Centrum Matematycznym w Amsterdamie zajmował się on m.in. programowaniem maszyny obliczeniowej ARMAC. Aby zademonstrować jej możliwości podczas prezentacji w 1956 r., postanowił posłużyć się rozwiązaniem problemu, który zrozumieją także osoby niezwiązane z matematyką. Dlatego napisał program znajdujący najkrótszą trasę między dwoma miastami w Holandii, korzystając z mapy drogowej, na której wybrał 64 miasta. W jednym z wywiadów Dijkstra powiedział, że wymyślenie algorytmu zajęło mu ok. 20 minut, a na jego pomysł wpadł, kiedy wraz z narzeczoną odpoczywał po zakupach w kawiarni przy kawie.

Podsumowanie

- Graf składa się z niepustego zbioru wierzchołków i zbioru krawędzi, czyli połączeń między wierzchołkami.
- Istnieją różne rodzaje grafów. W grafie nieskierowanym po krawędzi można przejść w obu kierunkach. W grafie skierowanym kierunek przejścia po krawędzi jest określony.
- Krawędziom mogą być przypisane wagi, określające np. odległość lub czas przejścia pomiędzy wierzchołkami. Taki graf nazywamy ważonym.
- Jeśli w grafie dowolne dwa wierzchołki łączy co najwyżej jedna krawędź oraz z żadnego wierzchołka nie prowadzi krawędź do niego samego, to taki graf jest grafem prostym.
- Graf, w którym istnieje droga z każdego wierzchołka do każdego, to graf spójny.
- Graf można reprezentować m.in. w postaci macierzy sąsiedztwa lub list sąsiedztwa.
- Do reprezentacji grafu można wykorzystać zmienną typu `vector` z biblioteki STL.
- Typ `vector` łączy możliwości tablicy dynamicznej i listy. Tak jak w tablicy dynamicznej w trakcie działania programu można określać i zmieniać rozmiar zmiennej typu `vector`. Podobnie jak w przypadku list nowy element można wstawić w dowolne miejsce tej zmiennej, można także usunąć dowolny element.
- Jednym z podstawowych problemów dotyczących grafów jest przejrzenie wierzchołków grafu w usystematyzowany sposób. Wyróżniamy dwie podstawowe metody przeglądania grafu: w głąb (DFS) oraz wszerz (BFS).
- Algorytm Dijkstry umożliwia znalezienie dróg o najmniejszych kosztach prowadzących od wybranego wierzchołka grafu ważonego o nieujemnych wagach krawędzi do wszystkich pozostałych wierzchołków grafu, do których można dojść.
- W każdym kroku algorytm Dijkstry wybiera nieodwiedzony wierzchołek o najmniejszym dotychczas znalezionym koszcie dojścia i rozpatruje sąsiadujące z nim nieodwiedzone wierzchołki.

Zadania

1. Napisz program, który wczyta opis skierowanego grafu nieważonego z pliku tekstowego otrzymanego od nauczyciela (np. *graf_2.txt*) oraz numer wierzchołka początkowego z klawiatury. Liczby w pierwszym wierszu pliku określają liczbę wierzchołków i liczbę krawędzi grafu, każdy kolejny wiersz zawiera numery wierzchołków będących początkiem i końcem krawędzi. Program ma sprawdzić, czy da się dojść z tego wierzchołka do pozostałych wierzchołków grafu.
2. Napisz program, który wczyta opis skierowanego grafu nieważonego z pliku tekstowego otrzymanego od nauczyciela (np. *graf_2.txt*) oraz numery wierzchołków początkowego i końcowego z klawiatury. Liczby w pierwszym wierszu pliku określają liczbę wierzchołków i liczbę krawędzi grafu, każdy kolejny wiersz zawiera numery wierzchołków będących początkiem i końcem krawędzi. Program powinien sprawdzić, czy istnieje droga z wierzchołka początkowego do końcowego.

* **3** Napisz program, który wczyta opis skierowanego grafu nieważonego z pliku tekstowego otrzymanego od nauczyciela (np. *graf_2.txt*) oraz numer wierzchołka z klawiatury. Liczby w pierwszym wierszu pliku odpowiadają liczbie wierzchołków i liczbie krawędzi grafu. Każdy kolejny wiersz zawiera numery wierzchołków będących początkiem i końcem krawędzi. Program ma sprawdzić, czy istnieje cykl zawierający ten wierzchołek.
Uwaga: Z cyklem w grafie mamy do czynienia, gdy istnieje ścieżka przejścia, w której wierzchołek początkowy i końcowy jest ten sam. Na przykład w grafie na rysunku 4.3 ze s. 63 jest cykl 0→1→2→0.

** **4** Napisz program, który wczyta opis skierowanego grafu nieważonego z pliku tekstowego otrzymanego od nauczyciela (np. *graf_2.txt*) i sprawdzi, czy w grafie istnieje cykl. Liczby w pierwszym wierszu pliku odpowiadają liczbie wierzchołków i liczbie krawędzi grafu. Każdy kolejny wiersz pliku zawiera numery wierzchołków będących początkiem i końcem krawędzi.
Uwaga: Z cyklem w grafie mamy do czynienia, gdy istnieje ścieżka przejścia, w której wierzchołek początkowy i końcowy jest ten sam. Na przykład w grafie na rysunku 4.3 ze s. 63 jest cykl 0→1→2→0.

** **5** Napisz program, który wczyta opis skierowanego grafu nieważonego z pliku tekstowego otrzymanego od nauczyciela (np. *graf_2.txt*) i sprawdzi, czy graf jest spójny. Liczby w pierwszym wierszu pliku określają liczbę wierzchołków i liczbę krawędzi grafu, każdy kolejny wiersz zawiera numery wierzchołków będących początkiem i końcem krawędzi.

** **6** Napisz program, który wczyta opis skierowanego grafu nieważonego z pliku tekstowego otrzymanego od nauczyciela (np. *graf_2.txt*) i przejrzy wierzchołki grafu w głąb bez użycia rekurencji, z wykorzystaniem stosu. Liczby w pierwszym wierszu pliku odpowiadają liczbie wierzchołków i liczbie krawędzi grafu. Każdy kolejny wiersz pliku zawiera numery wierzchołków będących początkiem i końcem krawędzi.

*** **7** Napisz program, który wczyta opis skierowanego grafu ważonego z pliku tekstowego otrzymanego od nauczyciela (np. *graf_3.txt*) oraz numery wierzchołków początkowego i końcowego z klawiatury. W pierwszym wierszu pliku znajdują się dwie liczby: pierwsza odpowiada liczbie wierzchołków grafu, a druga liczbie krawędzi. Każdy następny wiersz zawiera cztery liczby odpowiadające kolejno: numerowi wierzchołka będącego początkiem krawędzi, numerowi wierzchołka będącego końcem krawędzi, odległości między wierzchołkami, czasowi przejścia między wierzchołkami. Program ma znaleźć najkrótszą oraz najszybszą drogę pomiędzy wczytanymi wierzchołkami.

*** **8** Napisz program, który wczyta opis skierowanego grafu ważonego z pliku tekstowego otrzymanego od nauczyciela (np. *graf_4.txt*) oraz numery wierzchołków początkowego i końcowego z klawiatury. Liczby w pierwszym wierszu pliku odpowiadają liczbie wierzchołków i liczbie krawędzi grafu. Każdy kolejny wiersz pliku zawiera: numer wierzchołka będącego początkiem krawędzi, numer wierzchołka będącego końcem krawędzi, wagę krawędzi. Program ma znaleźć najkrótszą drogę pomiędzy wczytanymi wierzchołkami w czasie $O(n \cdot \log n)$.
Wskazówka: Wykorzystaj szablon typu `priority_queue` z biblioteki STL. Typ ten reprezentuje kolejkę priorytetową.

Sposób na zadania

Zadanie 1

W pliku, który otrzymasz od nauczyciela (np. *WUZ1_zad1_dane_liczby.txt*), w kolejnych 10 wierszach jest zapisanych po 20 liczb całkowitych z zakresu od 0 do 100 000 oddzielonych spacjami. Wśród tych 200 liczb są liczby zakończone każdą z cyfr od 0 do 9. Oto zawartość dwóch początkowych wierszy pliku:

```
820 21292 12637 322 21013 17178 16445 14757 9026 5161 25709 17961 14093 14598
12379 8458 8103 29640 426 2723
820 21292 12637 322 6736 6851 12166 8597 8781 9130 15482 25861 30047 15757 29632
30114 20605 29853 10139 14390
```

Napisz program lub programy rozwiązujące poniższe zadania. Odpowiedzi zapisz w sposób wskazany przez nauczyciela.

Zadanie 1.1 (0–3)

Utwórz plik wynikowy (np. *WUZ1_zad1_1_dane_liczby_wynik.txt*), w którym będzie 10 wierszy składających się z liczb oddzielonych spacjami. W poszczególnych wierszach znajdą się liczby z pliku z danymi, przy czym w pierwszym wierszu będą liczby zakończone cyfrą 0, w drugim – cyfrą 1 itd. Liczby w danym wierszu pliku wynikowego powinny wystąpić w tej samej kolejności co w pliku z danymi. Na przykład dla dwóch początkowych wierszy pliku z danymi wynik będzie następujący:

```
820 29640 820 9130 14390
5161 17961 6851 8781 25861
21292 322 21292 322 15482 29632
21013 14093 8103 2723 29853
30114
16445 20605
9026 426 6736 12166
12637 14757 12637 8597 30047 15757
17178 14598 8458
25709 12379 10139
```

Rozwiązanie

Tagi: tablica, dynamiczna struktura danych, kolejka

Sposób I. Wielokrotny odczyt danych z pliku bez użycia złożonych struktur danych

Krok 1

Liczby z pliku z danymi należy podzielić na 10 grup. O przynależności do grupy decyduje ostatnia cyfra liczby, czyli reszta z dzielenia liczby przez 10. Przejrzymy cały plik z danymi 10 razy. Za każdym razem zapiszemy do pliku wynikowego w oddzielnym wierszu liczby należące do jednej grupy.

Krok 2

Liczby z pliku z danymi odczytamy za pomocą zmiennej plikowej we typu `ifstream`. Wyniki zapiszemy z wykorzystaniem zmiennej plikowej wy typu `ofstream`.