# Sprawozdanie 06:  Praktyczna implementacja liniowych struktur danych: grafy

Dmytro Sakharskyi - L02 - 31259 14.05.2025

Metody Programowania

Informatyka 1 rok. 2 semestr

# Wstęp

Grafy są jedną z podstawowych struktur danych wykorzystywanych w informatyce oraz matematyce dyskretnej. Ich początki sięgają XVIII wieku, kiedy to Leonhard Euler rozwiązał **problem mostów** w Królewcu, tworząc tym samym fundamenty teorii grafów. Od tego czasu grafy znajdują szerokie zastosowanie w analizie sieci komputerowych, modelowaniu tras komunikacyjnych, zależności zadań w systemach operacyjnych, a nawet w algorytmach sztucznej inteligencji.

Podczas tych zajęć celem było zapoznanie się z podstawowymi reprezentacjami grafów oraz ich praktyczną implementacją w języku C++.

W szczególności skupiono się na dwóch dominujących sposobach przedstawienia grafu:

- za pomocą **macierzy sąsiedztwa**, czyli dwuwymiarowej tablicy wskazującej połączenia między wierzchołkami;
- za pomocą **listy sąsiedztwa**, która odwzorowuje relacje w bardziej dynamiczny sposób, szczególnie efektywny w grafach rzadkich.

# Zadanie 1 – Problem mostów w Królewcu

Problem mostów w Królewcu został postawiony w XVIII wieku i dotyczył pytania: **czy istnieje możliwość przejścia przez wszystkie siedem mostów w mieście tak, aby nie przechodzić przez żaden z nich więcej niż raz**?

Leonhard Euler rozwiązał ten problem, tworząc podstawy teorii grafów. W modelu grafowym:

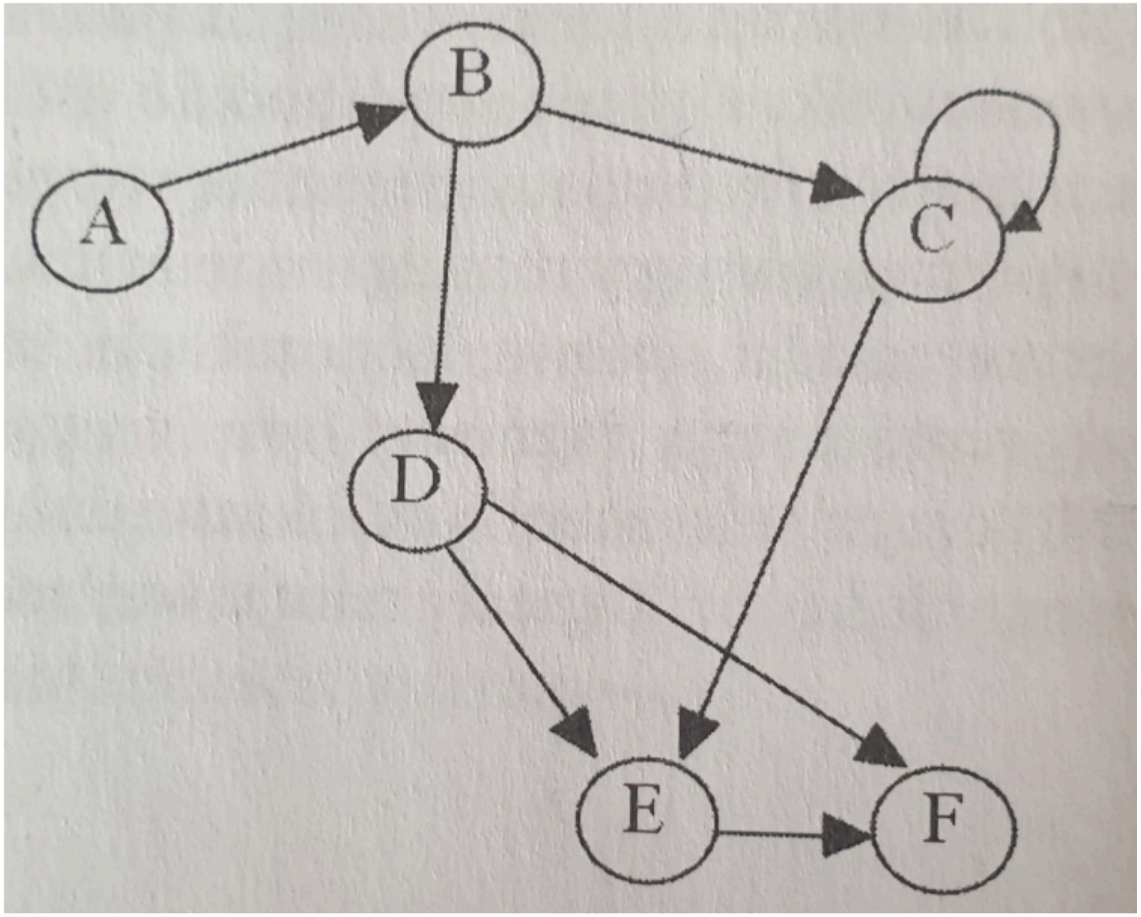
- **wierzchołki** reprezentują fragmenty lądu (wyspy i brzegi rzeki),
- **krawędzie** reprezentują mosty.

Z matematycznego punktu widzenia, rozwiązanie istniałoby tylko wtedy, gdy graf spełniałby warunki tzw. **ścieżki Eulera** — czyli drogi przechodzącej przez każdą krawędź dokładnie raz. Taka ścieżka istnieje **tylko wtedy**, gdy dokładnie **0 lub 2 wierzchołki mają nieparzysty stopień** (liczbę krawędzi wychodzących).

W przypadku mostów w Królewcu **wszystkie cztery wierzchołki mają nieparzysty stopień**, co oznacza, że **nie istnieje ścieżka Eulera**.

**Odpowiedź: Nie da się przejść przez wszystkie mosty dokładnie raz.**

# Zadanie 2 – Czy graf na Obraz 2 jest planarny?




*Obraz 2. Przykład grafu.*

Graf przedstawiony na *Obraz 2* to **graf skierowany**, posiadający m.in. pętlę (krawędź z C do C) oraz krawędzie krzyżujące się wizualnie na rysunku.

**Graf jest planarny**, jeśli da się go narysować na płaszczyźnie **bez przecinających się krawędzi** (oprócz punktów wspólnych w wierzchołkach).

Mimo że rysunek zawiera przecinające się krawędzie (np. linie między B–C i D–F), **istnieje możliwość narysowania tego grafu bez przecinania się krawędzi**, np. poprzez przemieszczenie wierzchołków lub zakrzywienie linii. Pętla na C również nie łamie planaryzmu.

**Odpowiedź:**
**Tak, graf z Obrazu 2 jest planarny.** Można go przedstawić w postaci planarnej poprzez odpowiednie przemieszczenie wierzchołków i uniknięcie przecięć między krawędziami.

# Zadanie 3

W ramach zadania zaimplementowano dwie klasy: GraphMatrix oraz GraphList, które reprezentują graf w postaci odpowiednio: **macierzy sąsiedztwa** oraz **listy sąsiedztwa**. Obie klasy zawierają metody addEdge() i print().

```
void addEdge(int i, int j) {
    graphMatrix[i][j] = 1;
    graphMatrix[j][i] = 1; // usun, jeśli graf ma być skierowany
}
```

Metoda ustawia wartość 1 w komórkach [i][j] oraz [j][i] w macierzy sąsiedztwa, co oznacza, że dodane zostaje połączenie pomiędzy wierzchołkami i i j. Wersja ta zakłada, że graf jest **nieskierowany** – połączenie działa w obie strony.

```
void addEdge(int u, int v) {
    graphList[u].push_back(v);
    graphList[v].push_back(u); // usun, jeśli graf ma być skierowany
}
```

Tutaj połączenie dodawane jest do wewnętrznych list sąsiedztwa w obu kierunkach. Każdy z wierzchołków u i v zapisuje drugiego jako swojego sąsiada.

```
void print() {
    cout << "  ";
    for (const auto& name : vertexNames) cout << name << " ";
    cout << "\n";
    for (int i = 0; i < verticesNumber; ++i) {
        cout << vertexNames[i] << " ";
        for (int j = 0; j < verticesNumber; ++j) {
            cout << graphMatrix[i][j] << "  ";
        }
        cout << "\n";
    }
}
```

Metoda wyświetla macierz sąsiedztwa – czyli tabelę, gdzie na przecięciu wierszy i kolumn znajdują się **1** lub **0**:

- **1** oznacza, że istnieje połączenie między wierzchołkami,

- **0** – brak połączenia.

Na górze i z lewej strony wypisane są nazwy wierzchołków, co ułatwia interpretację.

```
void print() {
    for (int i = 0; i < graphList.size(); ++i) {
        cout << vertexNames[i] << ": ";
        for (int neighbor : graphList[i]) {
            cout << vertexNames[neighbor] << " ";
        }
        cout << "\n";
    }
}
```

Ta metoda drukuje listę sąsiedztwa – każda linijka pokazuje, z którymi wierzchołkami połączony jest dany wierzchołek. Reprezentacja jest bardziej zwięzła i czytelna, szczególnie w przypadku rzadkich grafów (czyli takich, które mają niewiele krawędzi).

## Przykłady pracy:

### Macierz sąsiedztwa

```
Podaj liczbe wierzcholkow: 3
Podaj nazwy wierzcholkow:
0: d
1: v
2: w
Wybierz typ grafu:
1. Macierz sasiedztwa
2. Lista sasiedztwa
Wybˇr: 1
Podaj krawedzie w formacie <index1> <index2> (-1 -1 aby zakonczyc):
1 2
-1 -1

Reprezentacja grafu - macierz sasiedztwa:
  d v w
d 0  0  0
v 0  0  1
w 0  1  0
```

### Lista sąsiedztwa

```
Podaj liczbe wierzcholkow: 3
Podaj nazwy wierzcholkow:
0: d
1: v
2: w
Wybierz typ grafu:
1. Macierz sasiedztwa
2. Lista sasiedztwa
Wybˇr: 2
Podaj krawedzie w formacie <index1> <index2> (-1 -1 aby zakobczyc):
1 2
-1 -1

Reprezentacja grafu - lista sasiedztwa:
d:
v: w
w: v
```

# Zadanie 4

***a. Wybór sposobu reprezentacji grafu;***

```
cout << "Wybierz typ grafu:\n1. Macierz sasiedztwa\n2. Lista sasiedztwa\nWybór: ";
int choice;
cin >> choice;
```

Wpisanie 1 oznacza utworzenie grafu za pomocą **macierzy sąsiedztwa**.

Wpisanie 2 – za pomocą **listy sąsiedztwa**.

Ten wybór warunkuje, która klasa zostanie użyta do dalszego działania.

***b. Wybór liczby wierzchołków;***

```
int numVertices;
cout << "Podaj liczbe wierzcholkow: ";
cin >> numVertices;
```

Po wyborze reprezentacji użytkownik proszony jest o podanie liczby wierzchołków w grafie.

Na podstawie tej liczby tworzona jest odpowiednia struktura danych – tablica lub lista o zadanej wielkości.

***c. Wpisanie nazw wierzchołków;***

```
vector<string> names(numVertices);
cout << "Podaj nazwy wierzcholkow:\n";
for (int i = 0; i < numVertices; ++i) {
    cout << i << ": ";
    cin >> names[i];
}
```

Dzięki temu każdemu wierzchołkowi przypisywana jest unikalna nazwa (np. A, B, Sania...), która później wykorzystywana jest przy wyświetlaniu grafu w metodzie **print().**

***d. Stworzenie grafu, czyli możliwość wpisania przez użytkownika, który wierzchołek ma połączenie z którym.***

```
GraphMatrix gm(numVertices, names);
cout << "Podaj krawedzie w formacie <index1> <index2> (-1 -1 aby zakonczyc):\n";
while (true) {
    int u, v;
    cin >> u >> v;
    if (u == -1 || v == -1) break;
    gm.addEdge(u, v);
}
cout << "\nReprezentacja grafu - macierz sasiedztwa:\n";
gm.print();
```

Użytkownik podaje krawędzie w formie par indeksów (np. 0 2), które oznaczają połączenie między dwoma wierzchołkami. Pętla **kończy się**, gdy użytkownik wpisze -1 -1. Metoda **addEdge()** aktualizuje wewnętrzną strukturę danych i zapisuje połączenia między wierzchołkami.

# Zadanie 5

```
void print() {
    cout << "  ";
    for (const auto& name : vertexNames) cout << name << " ";
    cout << "\n";
    for (int i = 0; i < verticesNumber; ++i) {
        cout << vertexNames[i] << " ";
        for (int j = 0; j < verticesNumber; ++j) {
            cout << graphMatrix[i][j] << "  ";
        }
        cout << "\n";
    }
}
```

Tabela jest teraz podpisana nazwami wierzchołków zarówno w nagłówku, jak i po lewej stronie. Użytkownik widzi wyraźnie, które wierzchołki są ze sobą połączone.

**Dzięki wprowadzeniu nazw wierzchołków:**

- program stał się bardziej intuicyjny,
- grafy są czytelne nawet dla osób nieznających numeracji wierzchołków,
- reprezentacja przypomina realne scenariusze (np. mapa miast, połączenia sieciowe itp.).

# Wnioski

W ramach ćwiczenia zapoznano się z podstawową strukturą danych, jaką jest **graf**. Przeanalizowano dwa sposoby jego reprezentacji:

- **macierz sąsiedztwa**, która umożliwia szybki dostęp do informacji o istnieniu połączenia między dowolnymi dwoma wierzchołkami,
- **lista sąsiedztwa**, która jest bardziej efektywna pamięciowo dla grafów rzadkich.

Zaimplementowano klasy dla obu reprezentacji, w których przetestowano metody:

- **addEdge()** – służącą do dodawania krawędzi pomiędzy wierzchołkami,
- **print()** – wyświetlającą strukturę grafu w sposób czytelny.

Stworzono również prosty interfejs użytkownika, który umożliwia:

- wybór typu grafu,
- wpisanie liczby i nazw wierzchołków,
- podanie krawędzi w formie indeksów.

Na końcu ulepszono metody print(), tak aby wyświetlały **nazwy wierzchołków** zamiast indeksów. Dzięki temu struktura grafu jest łatwiejsza do odczytania przez człowieka i bardziej praktyczna w użyciu.