Técnicas e Análise de Algoritmos

Grafos - Parte 01

Professor: **Jeremias Moreira Gomes**

E-mail: jeremias.gomes@idp.edu.br

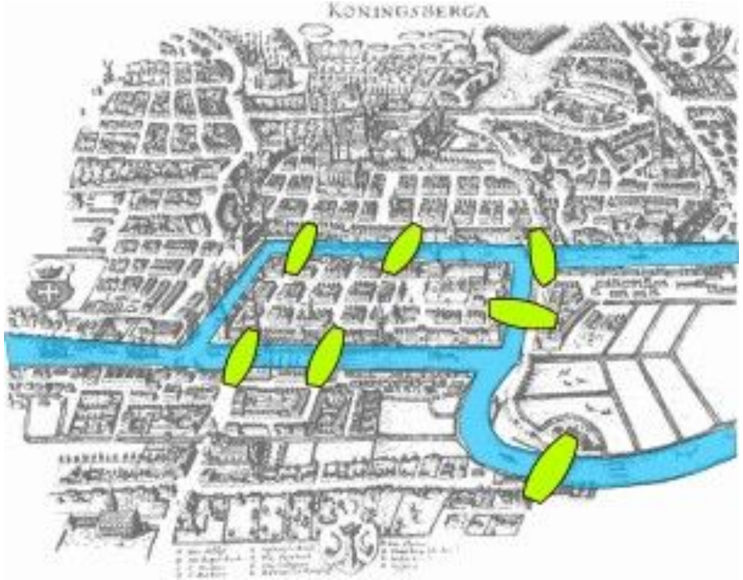
Introdução

Introdução

- **O que é um grafo?**
    - É um objeto abstrato para representar relações entre “coisas”
    - Possui dois tipos de entidades
        - Nós (ou vértices)
        - Ramos (ou arestas)

Introdução

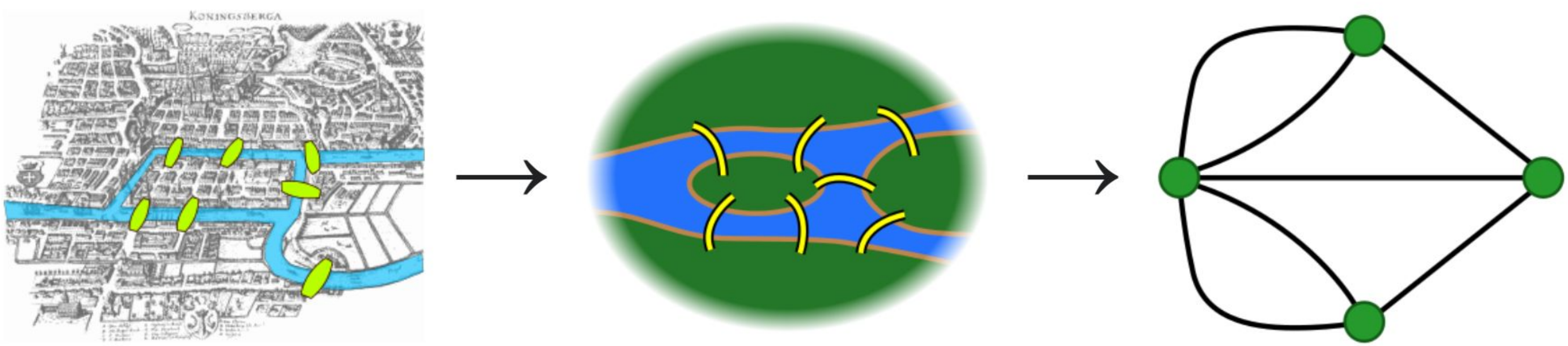
- **De onde surgiu?**
    - Um problema (Euler) chamado As Pontes de Königsberg



Seria possível visitar todas as regiões da cidade, passando por cada ponte exatamente uma vez?

Introdução

- **De onde surgiu?**
  - Um problema (Euler) chamado As Pontes de Königsberg



Seria possível visitar todas as regiões da cidade, passando por cada ponte exatamente uma vez?

Introdução

- **Por que estudar grafos?**
  - Os grafos aparecem intimamente ligados a quase todas as outras estruturas de dados
    - Árvores (já estudadas na disciplina) são um tipo de grafo
  - Grafos modelam muitos problemas do mundo real
    - Algoritmos clássicos resolvem problemas recorrentes
  - Travessias em grafos são eficientes e úteis

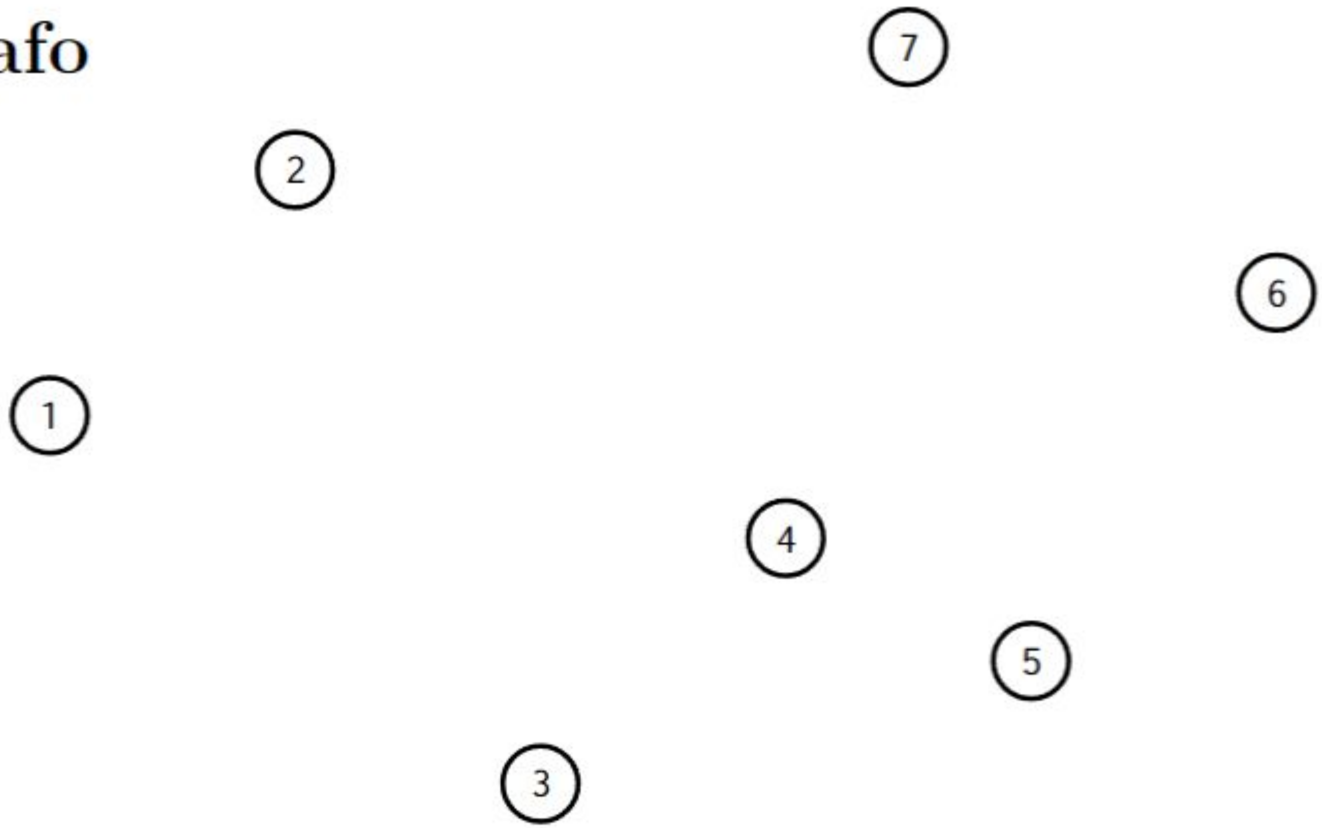
Introdução

- **Definição formal de um grafo**
    - Um grafo G(V, A) é definido pelo par de conjunto V e A onde:
        - V - conjunto não vazio e são os vértices do grafo
        - A - conjunto de pares ordenados a = (v, w), onde v e w $\in$ V e são as aresta desse grafo

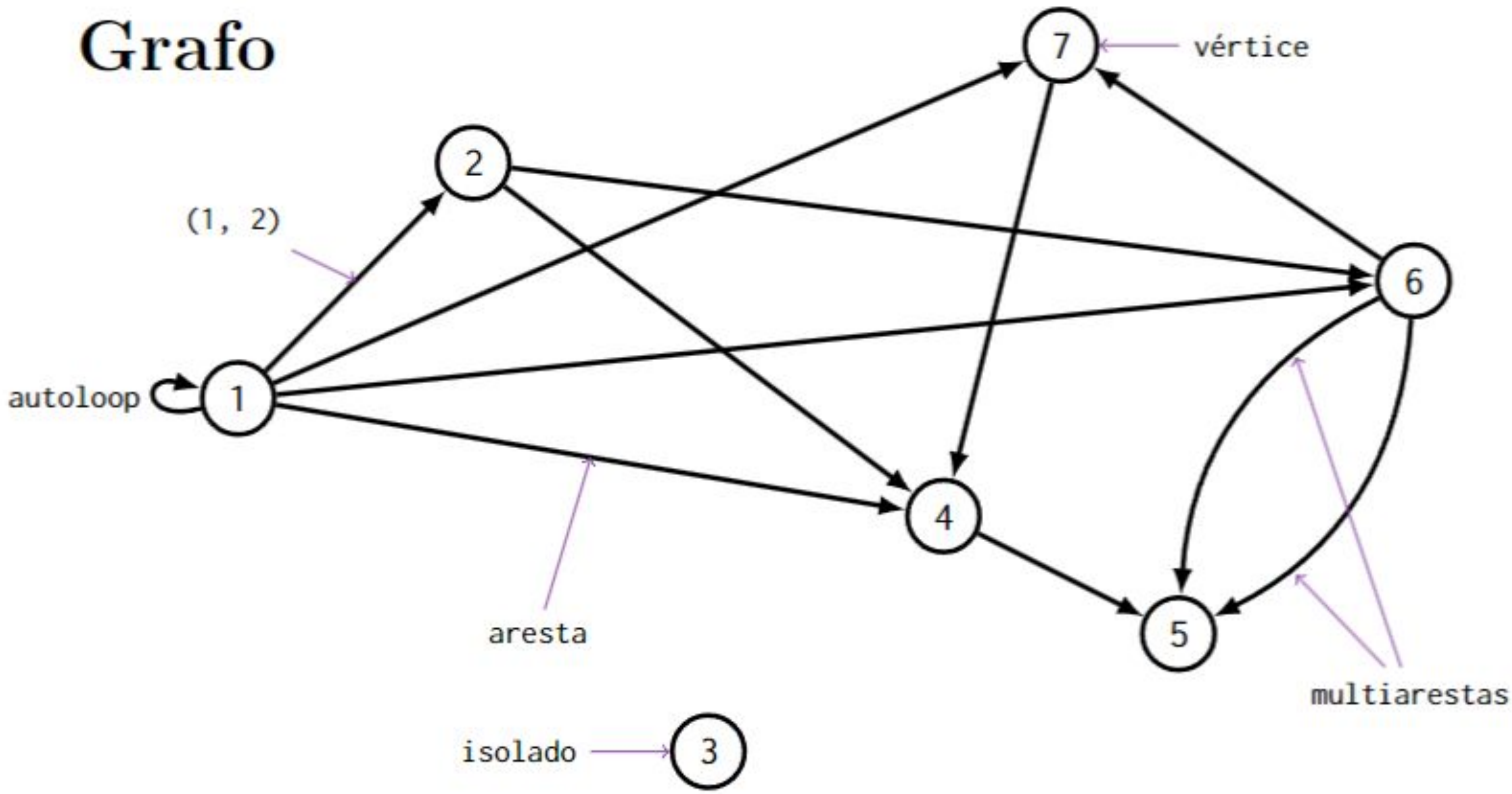
Introdução

- **Definição formal de um grafo**
  - Exemplo:
    - V = { p | p é uma pessoa }
      - V = { Klayton, LucasA, Lucas, Leandro }
    - A = { (v,w) | < v é amigo de w > }
      - A = {(Klayton, LucasA), (Klayton, Lucas), (Klayton,Leandro), (LucasA, Klayton)}

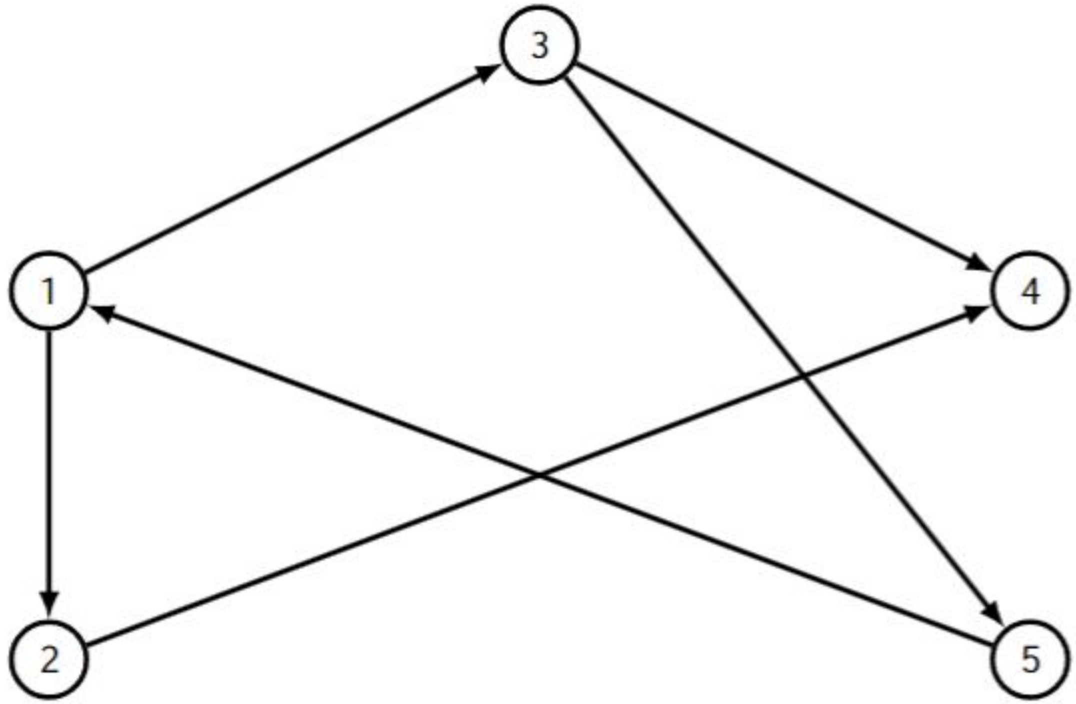
Grafo



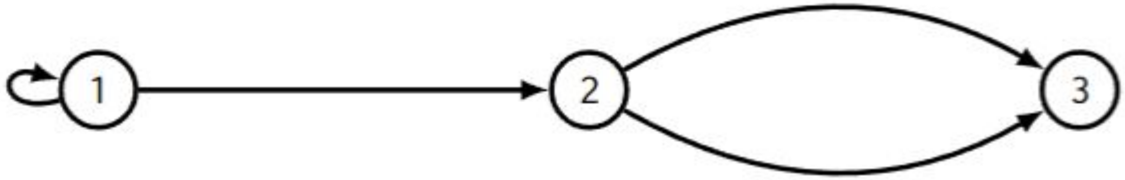
Grafo



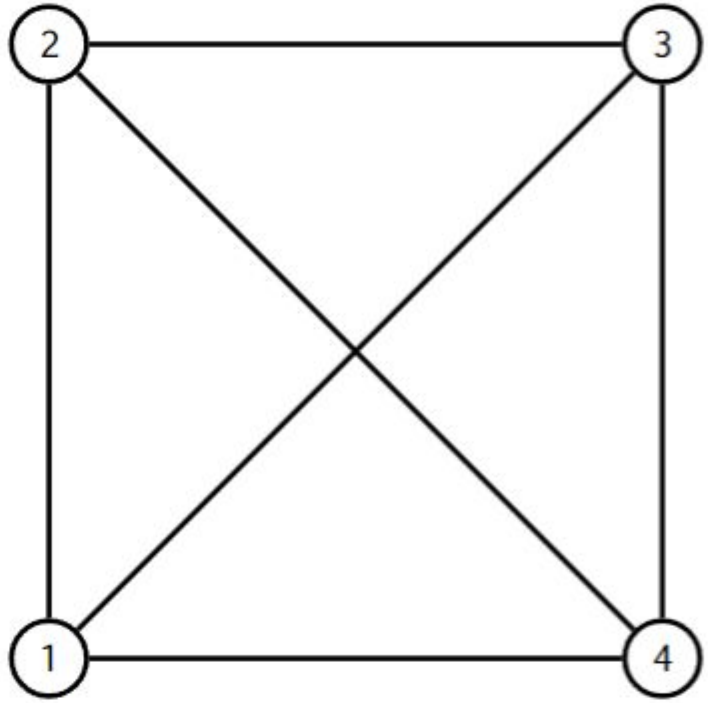
Grafo simples



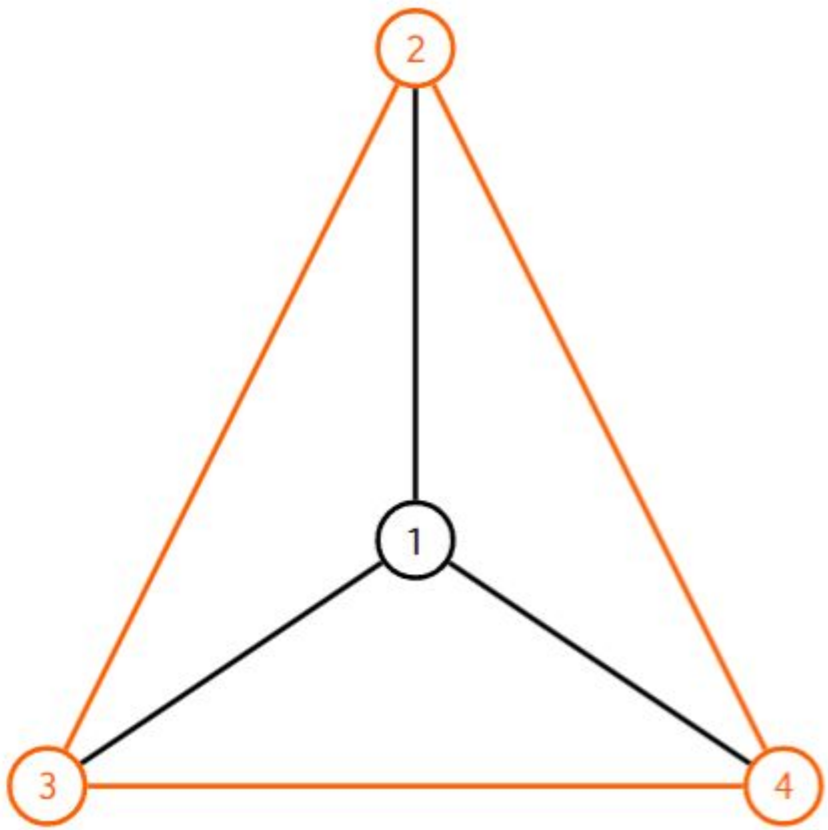
Multigrafo



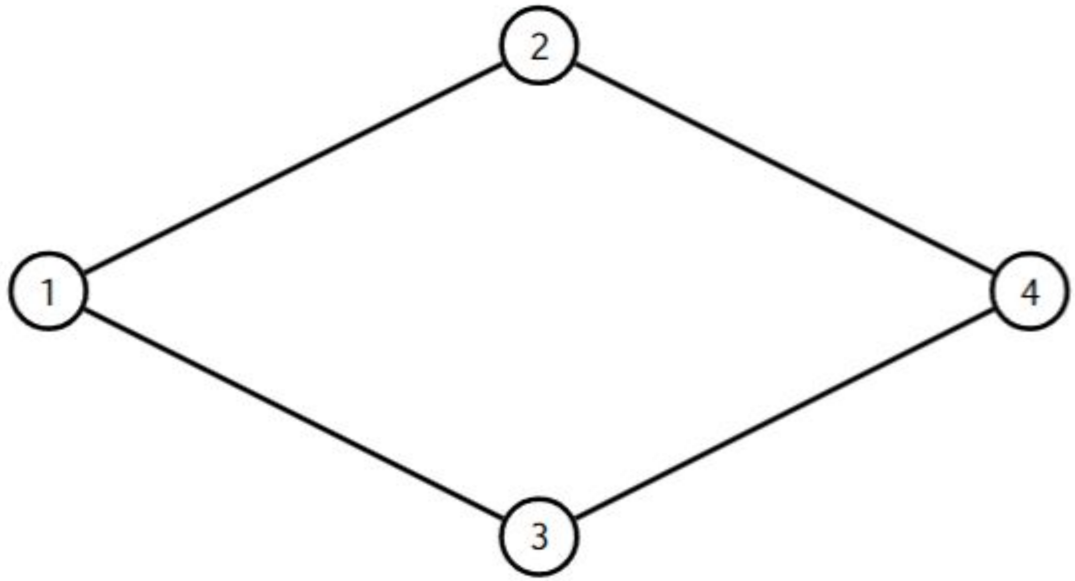
Grafo completo



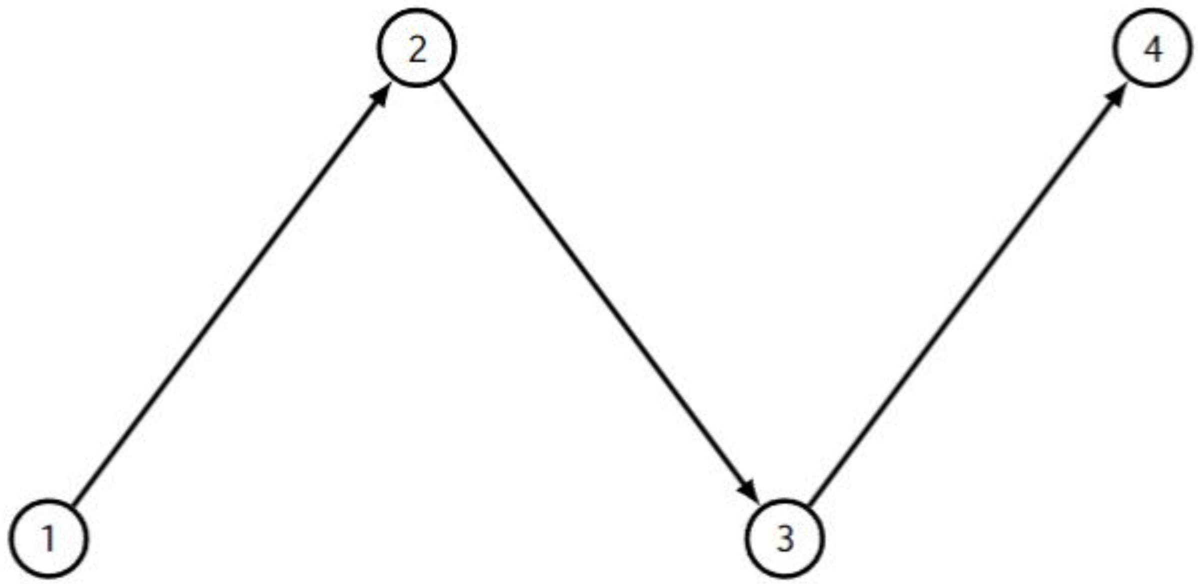
Subgrafo



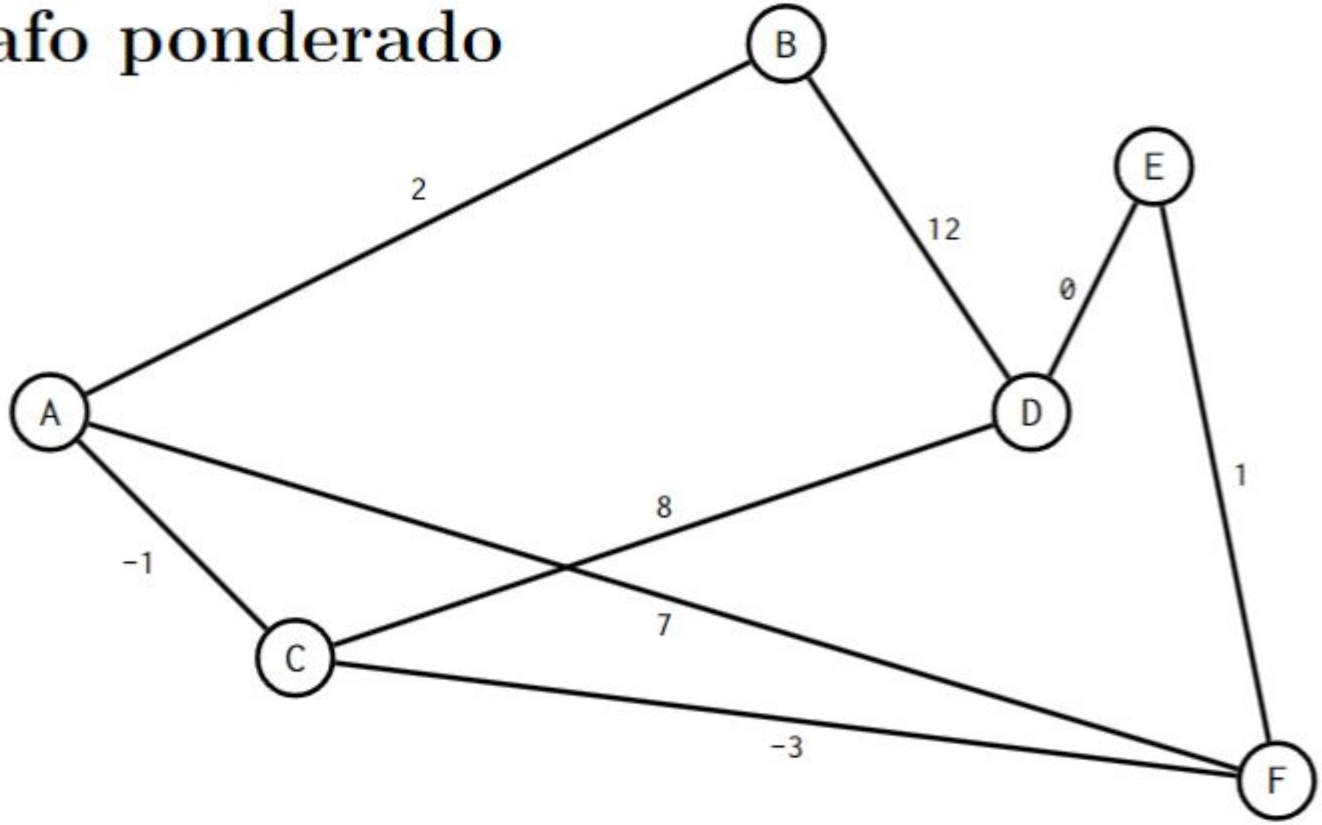
Grafo não-direcionado



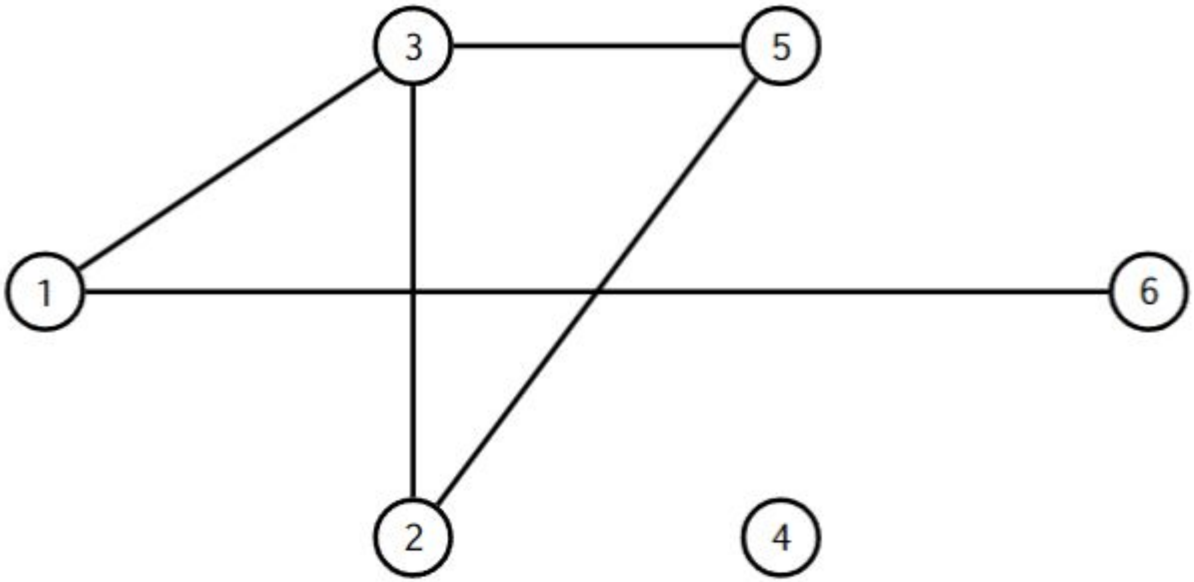
Grafo direcionado



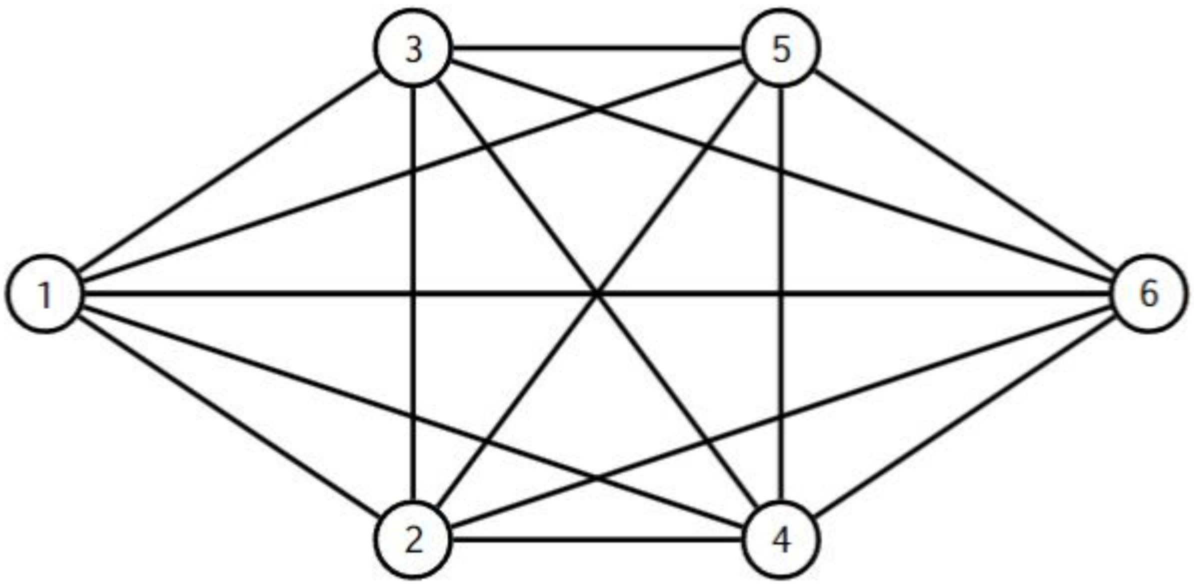
Grafo ponderado



Grafo esparso



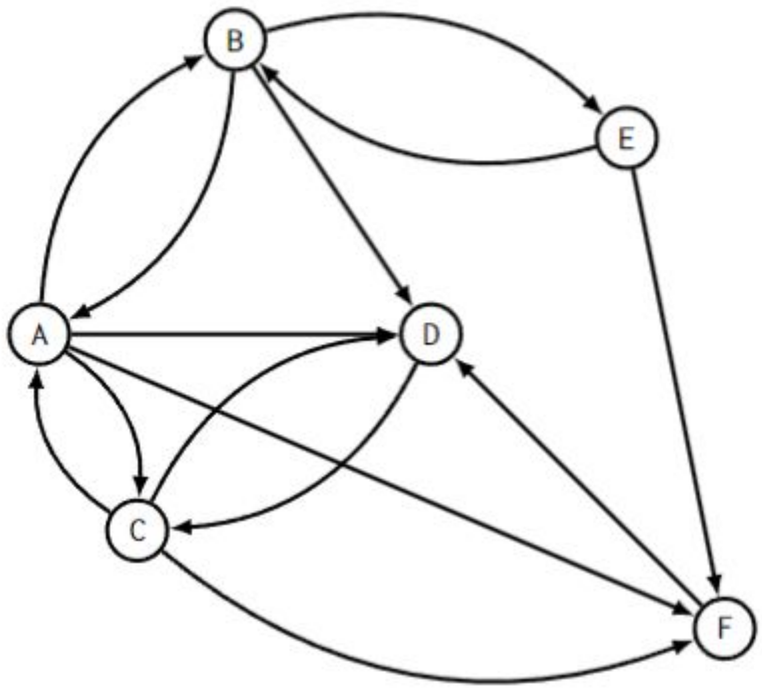
Grafo denso



Grau de um Grafo

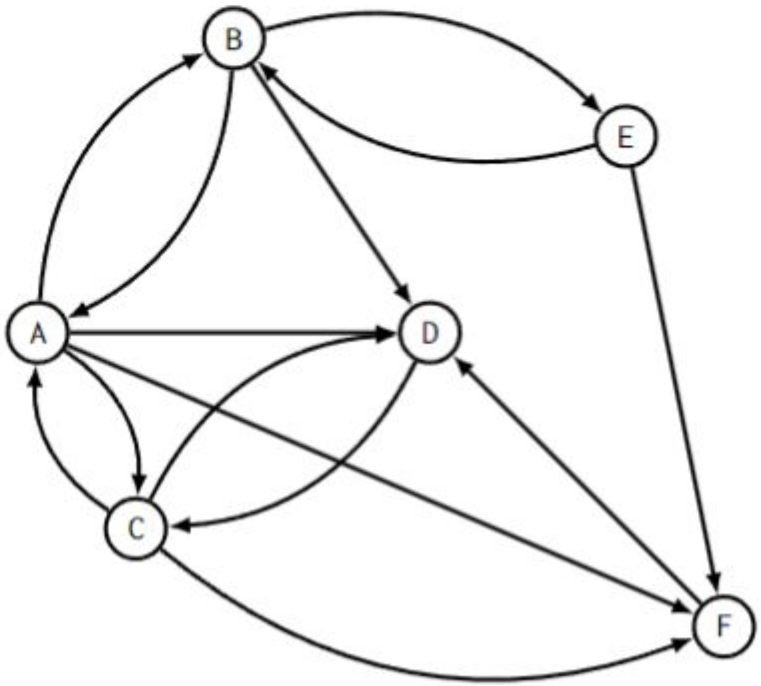
- **O grau de vértice u, é a quantidade de arestas que se ligam a esse vértice**
- Em grafos direcionados (digrafos), graus são divididos em:
    - Grau de entrada gi(u): arestas que chegam em u
    - Grau de saída go(u): arestas que partem de u

Grau de um Grafo



u	$g_i(u)$	$g_o(u)$

Grau de um Grafo



u	$g_i(u)$	$g_o(u)$
A	2	4
B	2	3
C	2	3
D	4	1
E	1	2
F	3	1

Caminho em um Grafo

- Um caminho é uma sequência não-nula de vértices da forma

$$(u, w_1), (w_1, w_2), (w_2, w_3), \ldots, (w_{n-1}, w_n), (w_n, v)$$

onde u é o ponto de partida e v o ponto de chegada

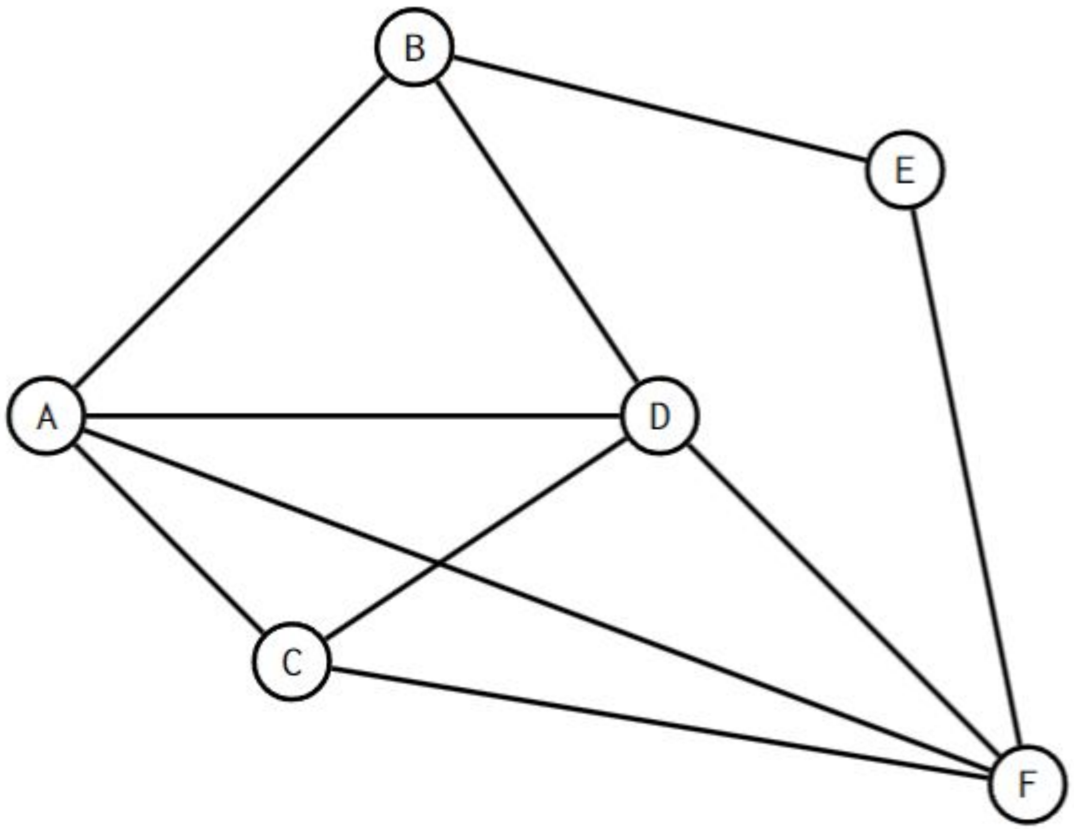
Caminho em um Grafo

- Um caminho é uma sequência não-nula de vértices da forma

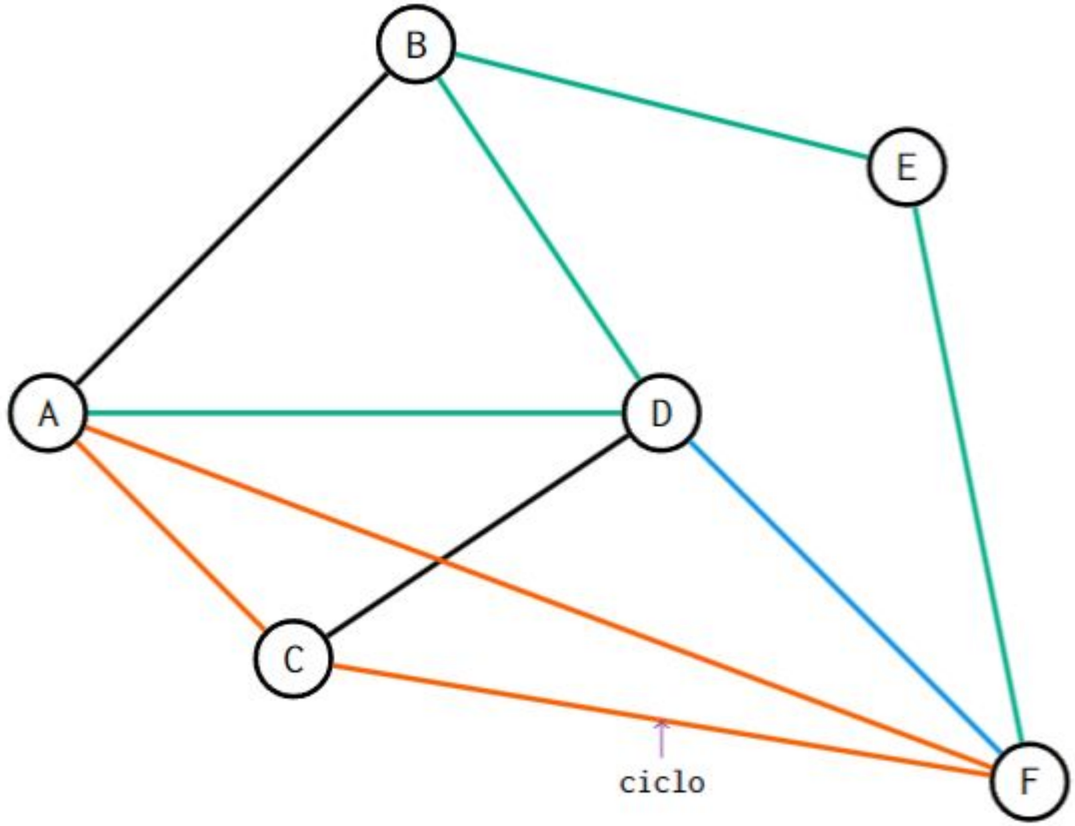
$$(u, w_1), (w_1, w_2), (w_2, w_3), \ldots, (w_{n-1}, w_n), (w_n, v)$$

onde u é o ponto de partida e v o ponto de chegada

Caminho em um Grafo



Caminho em um Grafo



Representações em Grafos

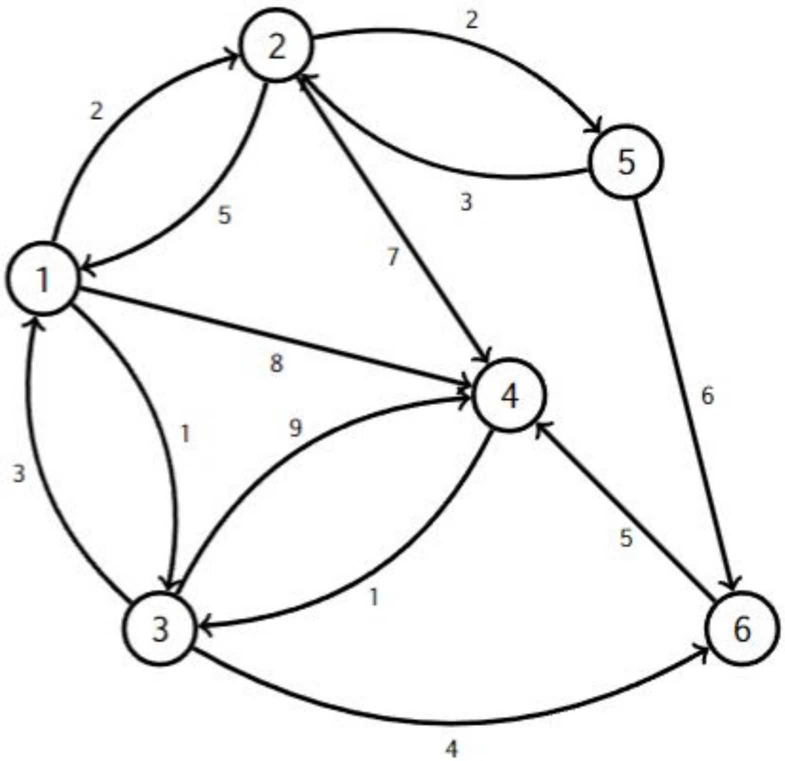
Representações em Grafos

- Grafos podem ser representados de quatro formas diferentes:
  - Matriz de adjacências
  - Lista de adjacências
  - Lista de arestas
  - Representações implícitas

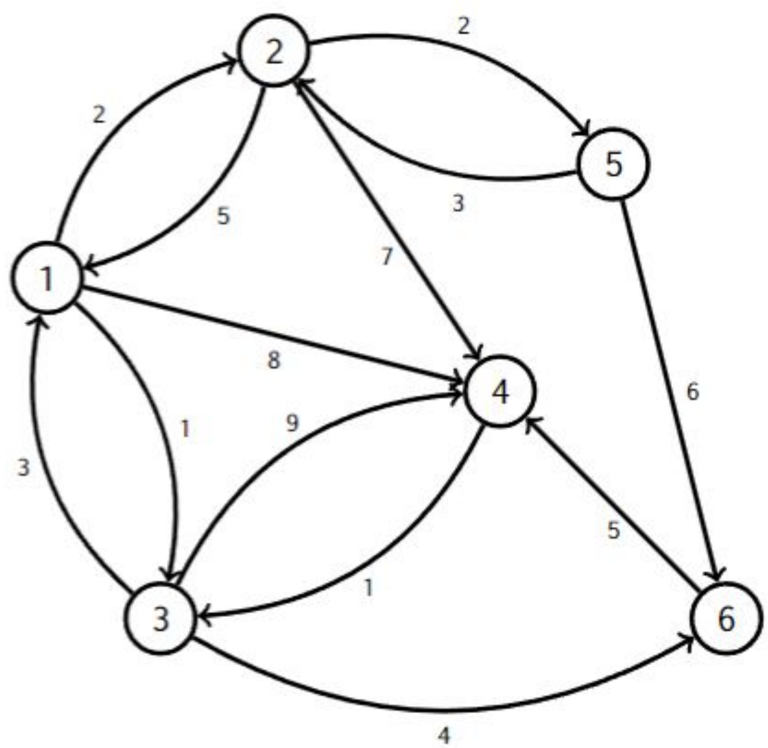
Representações em Grafos

- **Matriz de adjacências**
  - Seja G um grafo com N vértices
    - Assuma que cada vértice seja associado a um inteiro positivo em [1, N]
    - Na matriz de adjacências $A_{N \times N}$ o elemento a_{ij} é o peso da aresta (i, j)
  - Se $(i, j) \notin E$, então $a_{ij} = 0$

Representações em Grafos



Representações em Grafos



$$A = \begin{bmatrix} 0 & 2 & 1 & 8 & 0 & 0 \\ 5 & 0 & 0 & 7 & 2 & 0 \\ 3 & 0 & 0 & 9 & 0 & 4 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 5 & 0 & 0 \end{bmatrix}$$

Representações em Grafos

- **Matriz de adjacências - Características**
  - Se G é não ponderado, $a_{ij} \in [0, 1]$
  - Se G é um multigrafo, a_{ij} pode registrar o número de ocorrências de (i, j)
  - Se G é simples, $a_{ii} = 0, \forall_i \in V$
  - Vantagem: Consulta "(i, j) ∈ E" é respondida em O(1)
  - Desvantagem: Complexidade de memória $O(N^2)$

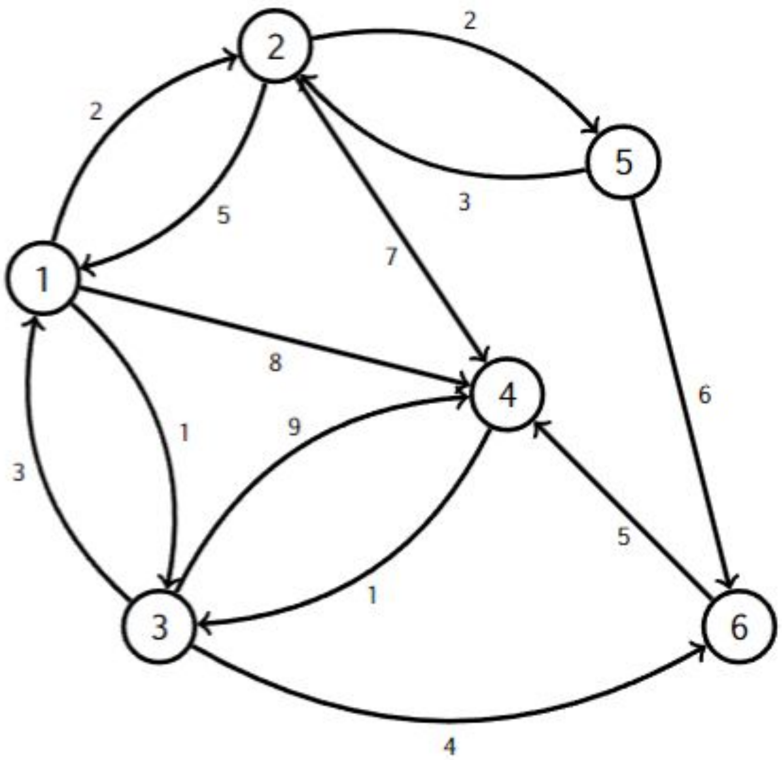
```cpp
#define N 6
int G[N + 1][N + 1];
int main() {
    G[1][2] = 2, G[1][3] = 1, G[1][4] = 8;
    G[2][1] = 5, G[2][4] = 7, G[2][5] = 2;
    G[3][1] = 3, G[3][4] = 9, G[3][6] = 4;
    G[4][3] = 1;
    G[5][2] = 3, G[5][6] = 6;
    G[6][4] = 5;

    for (int i = 1; i <= N; ++i) {
        for (int j = 1; j <= N; ++j) {
            cout << G[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

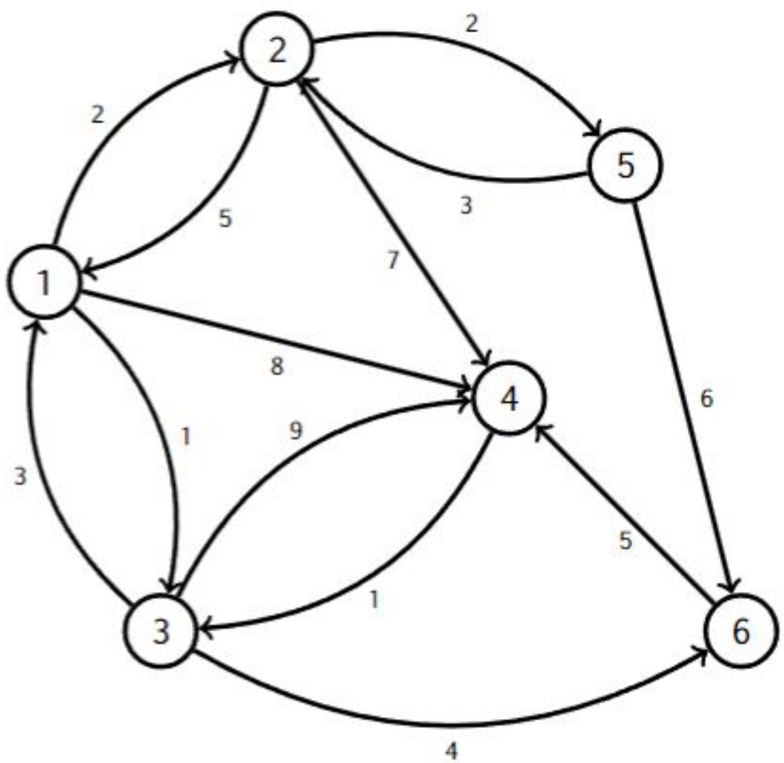
Representações em Grafos

- **Lista de adjacências**
  - A cada vértice u é associada uma lista L(u)
  - Essa lista contém os vértices v tais que (u, v) ∈ E
  - Se G é ponderado, então os elementos de l(u) são pares (vi, wi)

Representações em Grafos



Representações em Grafos



1	(2, 2)(3, 1)(4, 8)
2	(1, 5)(4, 7)(5, 2)
3	(1, 3)(4, 9)(6, 4)
4	(3, 1)
5	(2, 3)(6, 6)
6	(4, 5)

Representações em Grafos

- **Lista de adjacências**
  - Possíveis listas em C++: list, vector ou forward_list
  - Complexidade de memória: O(N + M)
    - Onde M é o número de arestas
  - São melhores para grafos esparsos
  - Algoritmos clássicos utilizam esta representação (opção mais utilizada)

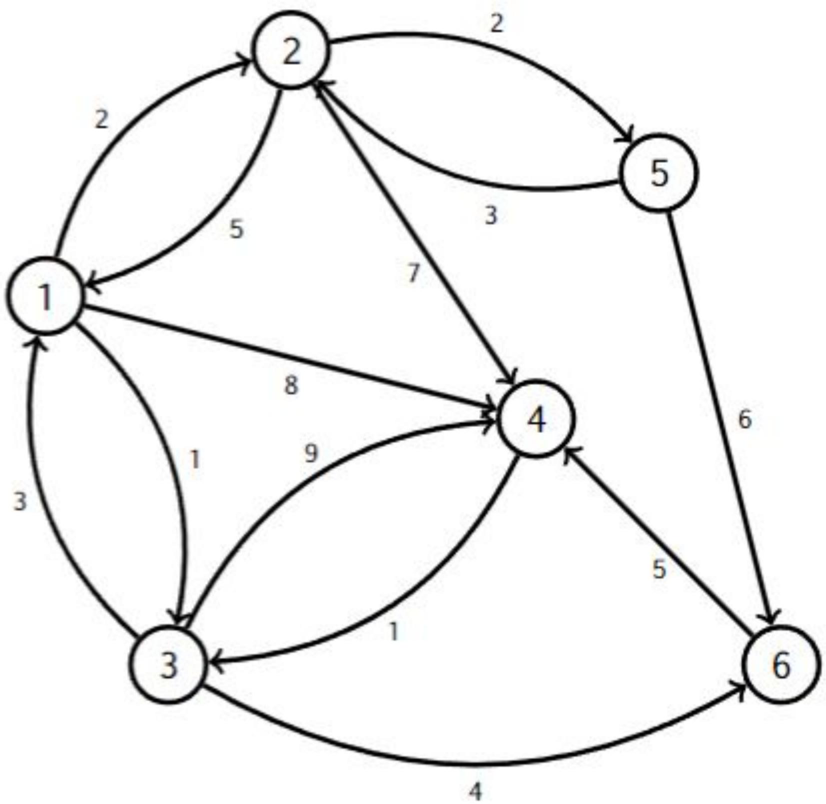
```cpp
vector<pair<int, int>> G[] {
    {},
    { { 2, 2 }, { 3, 1 }, { 4, 8 } },
    { { 1, 5 }, { 4, 7 }, { 5, 2 } },
    { { 1, 3 }, { 4, 9 }, { 6, 4 } },
    { { 3, 1 } },
    { { 2, 3 }, { 6, 6 } },
    { { 4, 5 } }
};

for (int u = 1; u <= N; u++) {
    cout << u << ": ";
    for (auto [v, w]: G[u]) {
        cout << "(" << v << ", " << w << ") ";
    }
    cout << endl;
}
```

Representações em Grafos

- **Lista de arestas**
  - O grafo G é representado pelo conjunto de arestas E
  - Cada aresta é representada pela tripla (u, v, w)
  - É possível deduzir V a partir de E se G não tem vértices isolados
  - Complexidade de memória: O(M)

Representações em Grafos



(1, 2, 2)
(1, 3, 1)
(1, 4, 8)
(2, 1, 5)
(2, 4, 7)
(2, 5, 2)
(3, 1, 3)
(3, 4, 9)
(3, 6, 4)
(4, 3, 1)
(5, 2, 3)
(5, 6, 6)
(6, 4, 5)

Representações em Grafos

```
vector<tuple<int, int, int>> A {
{ 1, 2, 2 }, { 1, 3, 1 }, { 1, 4, 8 },
{ 2, 1, 5 }, { 2, 4, 7 }, { 2, 5, 2 },
{ 3, 1, 3 }, { 3, 4, 9 }, { 3, 6, 4 },
{ 4, 3, 1 }, { 5, 2, 3 }, { 5, 6, 6 },
{ 6, 4, 5 } };

for (auto [u, v, w]: A) {
    cout << "(" << u << ", " << v << ", " << w << ") " << endl;
}
```

Representações em Grafos

- **Representação implícita**
  - Os vértices e as arestas são definidas por relações
  - Adequada para grafos complexos ou com infinitos vértices e arestas
  - Os elementos do grafo são identificados sob demanda

Representações em Grafos

- **Representação implícita**
  - Exemplo
    - Imagine um mapa bidimensional onde:
      - Paredes são representadas por #
      - Caminhos livres são representados por _

Representações em Grafos

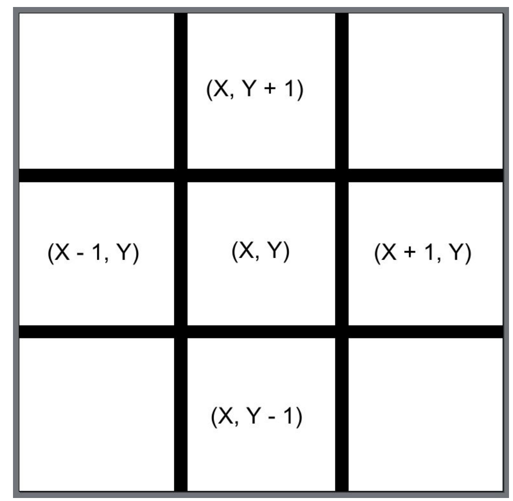
- **Representação implícita**

```
__##################
________##__##____##
######__##__##__####
######__##__##__####
##__________##____##
##########__########
##########__########
##____####________##
####________####__##
################____
```

Representações em Grafos

- **Representação implícita**

```
__##################
________##__##____##
######__##__##__####
######__##__##__####
##__________##____##
##########__########
##########__########
##____####________##
####________####__##
################____
```



Travessia

Travessias

- Uma travessia consiste em visitar todos vértices de um grafo em alguma ordem
- Cada vértice deve ser visitado exatamente uma vez

Travessias

- Características:
  - Duas travessias são distintas se as ordens de visitação são diferentes
  - Um grafo conectado com N vértices tem N! travessias distintas
  - Travessias mais conhecidas (e aplicadas):
    - Por profundidade
    - Por extensão

Travessia por Profundidade (Depth-first search)

- Seja s o vértice de partida e u o vértice observado no momento. As regras abaixo definem a DFS:

    a. Faça u = s
    b. Visite u
    c. Se u tiver ao menos um vizinho v ainda não visitado, faça u = v e retorne para a regra 2
    d. Caso contrário, volte para o vértice que descobriu u

Travessia por Profundidade (Depth-first search)

- Características da DFS:
  - Cada nó é visitado uma única vez
  - A complexidade é O(V + E) em listas de adjacências
  - Já em matrizes de adjacência a complexidade é $O(N^2)$

Travessia por Profundidade (Depth-first search)

- Implementação:
  - A DFS pode ser implementada por recursão
  - Caso-base: vértice já visitado
  - Chamada recursiva: vizinhos de u ainda não visitados

Travessia por Profundidade (Depth-first search)

```
bool visitado[N + 1];

void dfs(int u)
{
    visitado[u] = true;
    for (auto [v, w]: G[u]) {
        if (!visitado[v]) {
            dfs(v);
        }
    }
}
```

Travessia por Largura (Breadth-First Search)

- Seja s o vértice de partida e u o vértice observado no momento. As regras abaixo definem a BFS:
    a. Faça u = s
    b. Visite u
    c. Enfileire todos vizinhos de u não visitados
    d. Extraia o próximo elemento p da fila, faça u = p e retorne a regra 2

Travessia por Largura (Breadth-First Search)

- Características da BFS:
  - A DFS e a BFS visitam os mesmos vértices, em ordem distintas
  - A complexidade em listas de adjacências é $O(V + E)$
    - A mesma da DFS
  - Em matrizes de adjacências, a complexidade é $O(N^2)$

Travessia por Largura (Breadth-First Search)

- Implementação da BFS:
  - Mais elaborada do que a da DFS, pois não usa recursão
  - Demanda uma fila para a manutenção da ordem de travessia
  - O vetor visitados pode ser substituído pela distância de u até s em arestas

```cpp
bool visitado[N + 1];

void bfs(int u) {
    queue<int> fila;
    fila.push(u);
    visitado[u] = true;

    while (!fila.empty()) {
        u = fila.front();
        fila.pop();

        for (auto [v, w]: G[u]) {
            if (!visitado[v]) {
                fila.push(v);
                visitado[v] = true;
            }
        }
    }
}
```

```
vector<int>dist(N + 1, oo);

void bfs(int u) {
    queue<int> fila;
    fila.push(u);
    dist[u] = 0;
    while (!fila.empty()) {
        u = fila.front();
        fila.pop();

        for (auto [v, w]: G[u]) {
            if (dist[v] == oo) {
                dist[v] = dist[u] + w;
                fila.push(v);
            }
        }
    }
}
```

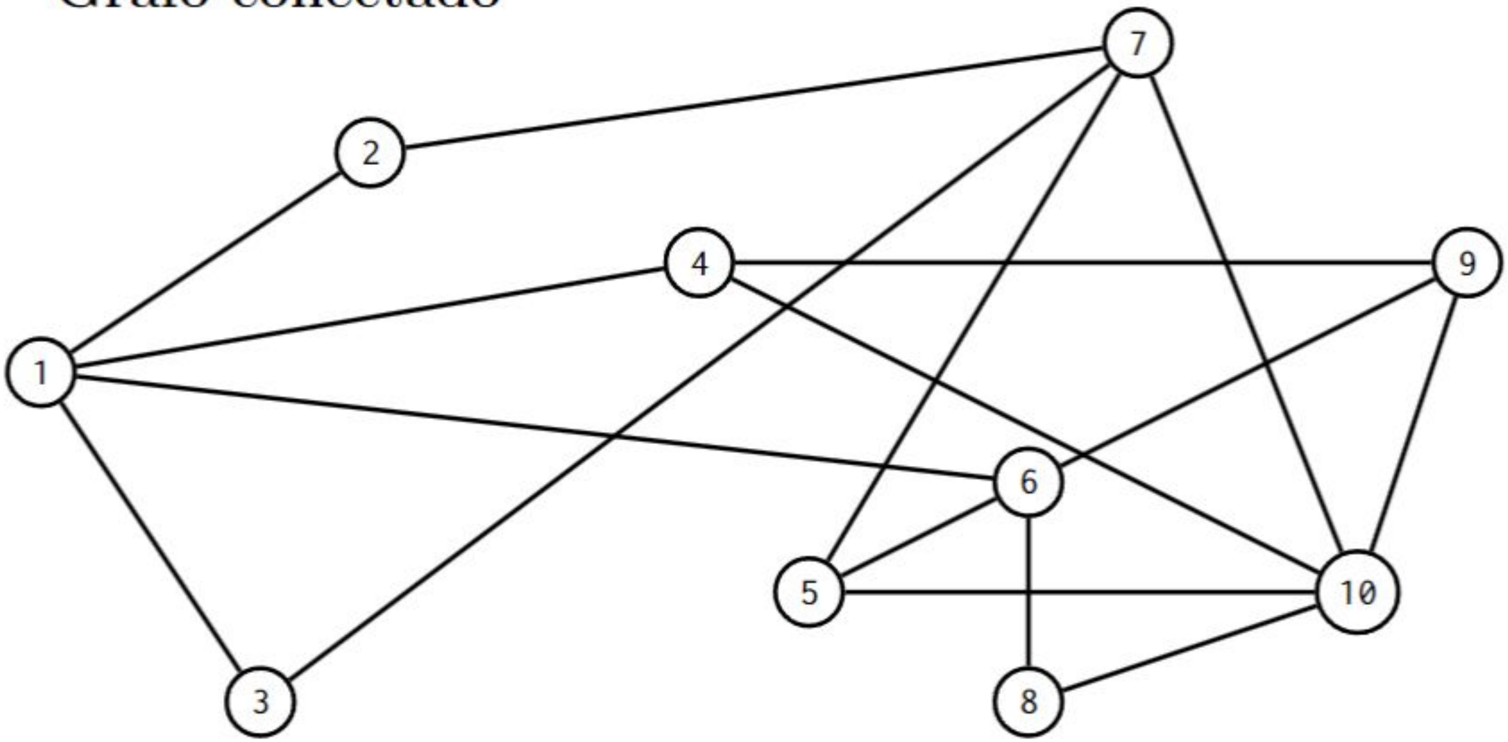
Componentes Conectados

Componentes Conectados

- Um grafo não-direcionado G(V, E) é dito conectado se, para qualquer par de vértices u, v $\in$ V , existe ao menos um caminho de u a v
  - Ou seja, é possível acessar qualquer vértice a partir de qualquer outro vértice

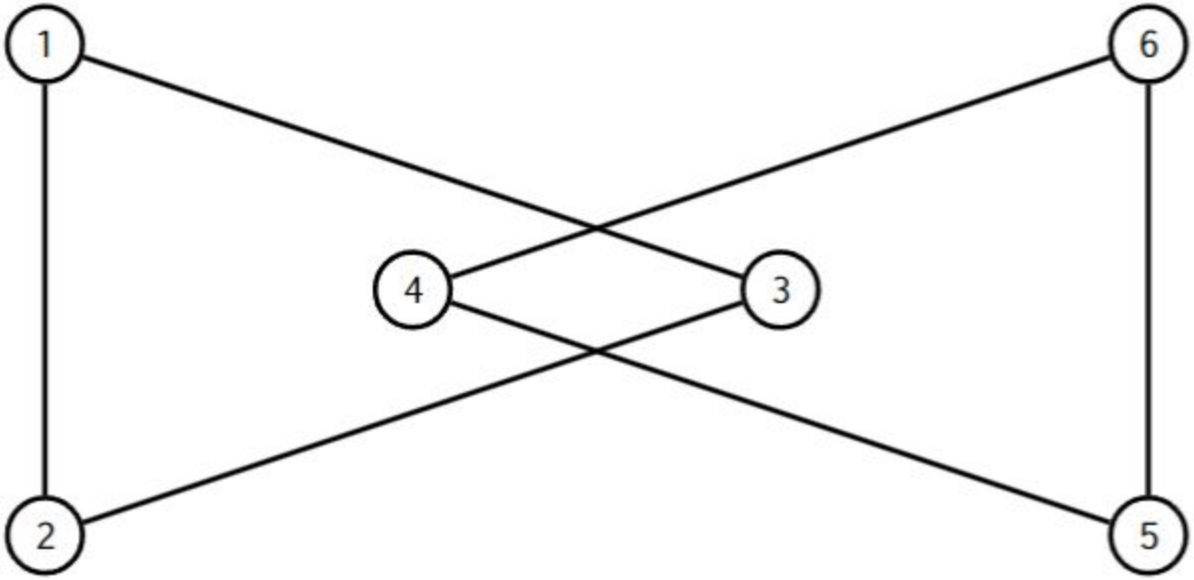
Componentes Conectados

Grafo conectado



Componentes Conectados

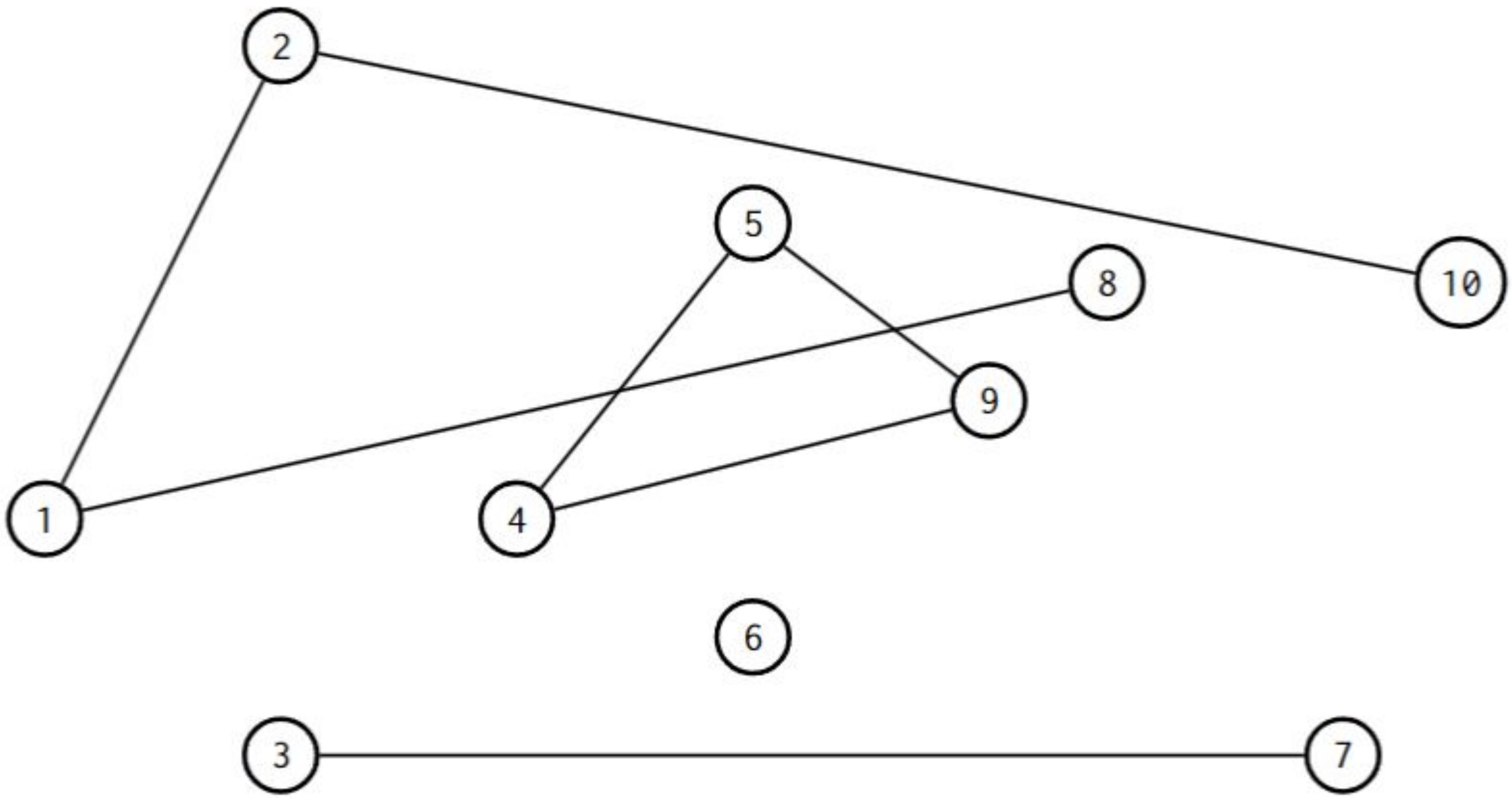
Grafo não-conectado



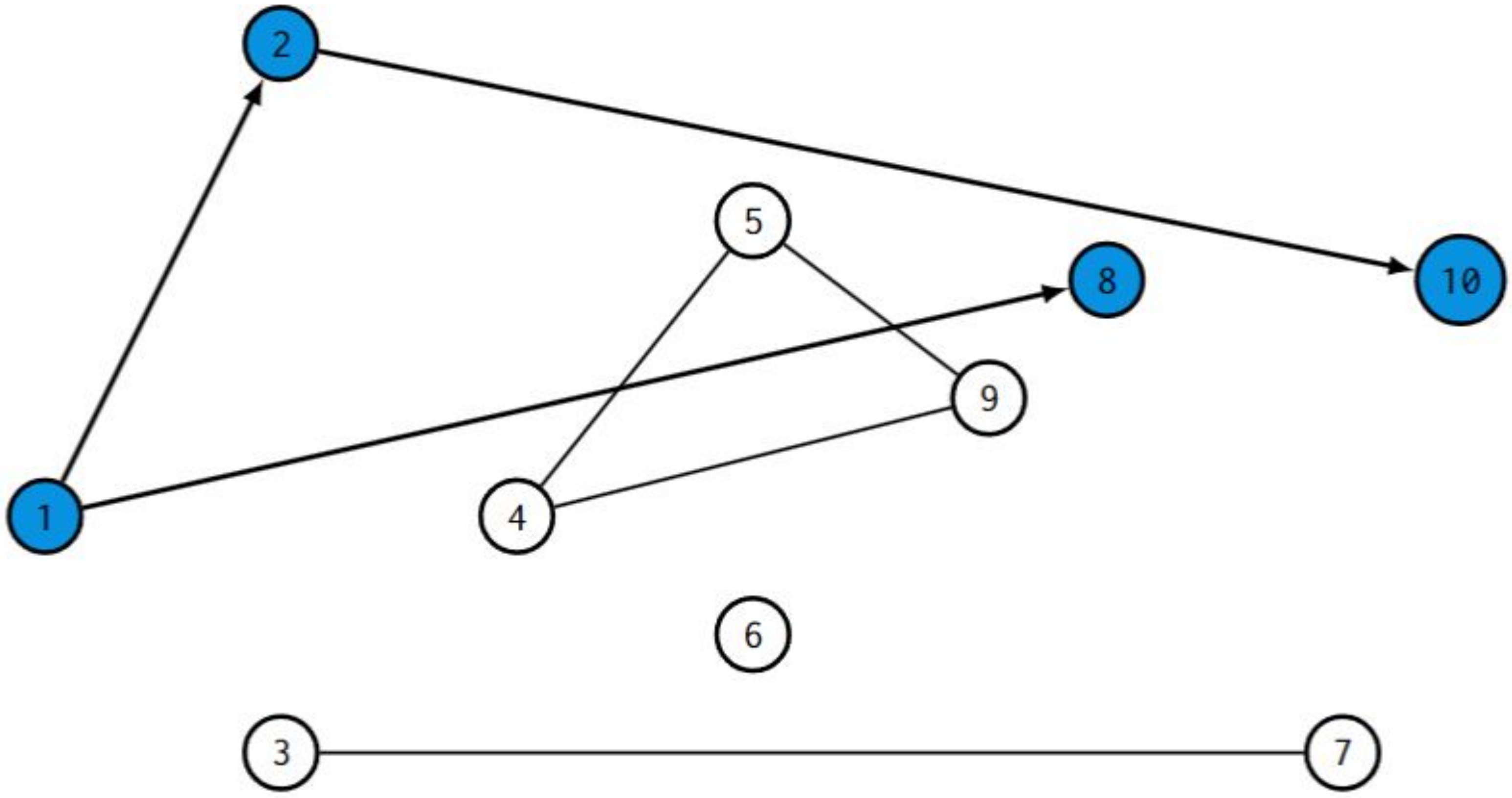
Componentes Conectados

- Já um componente conectado do grafo G(V, E) que contém o vértice u é o maior subgrafo conectado S(V’ , E’) de G tal que u ∈ V’
  - Os elementos de V’ podem ser determinados por meio de uma travessia com início em u

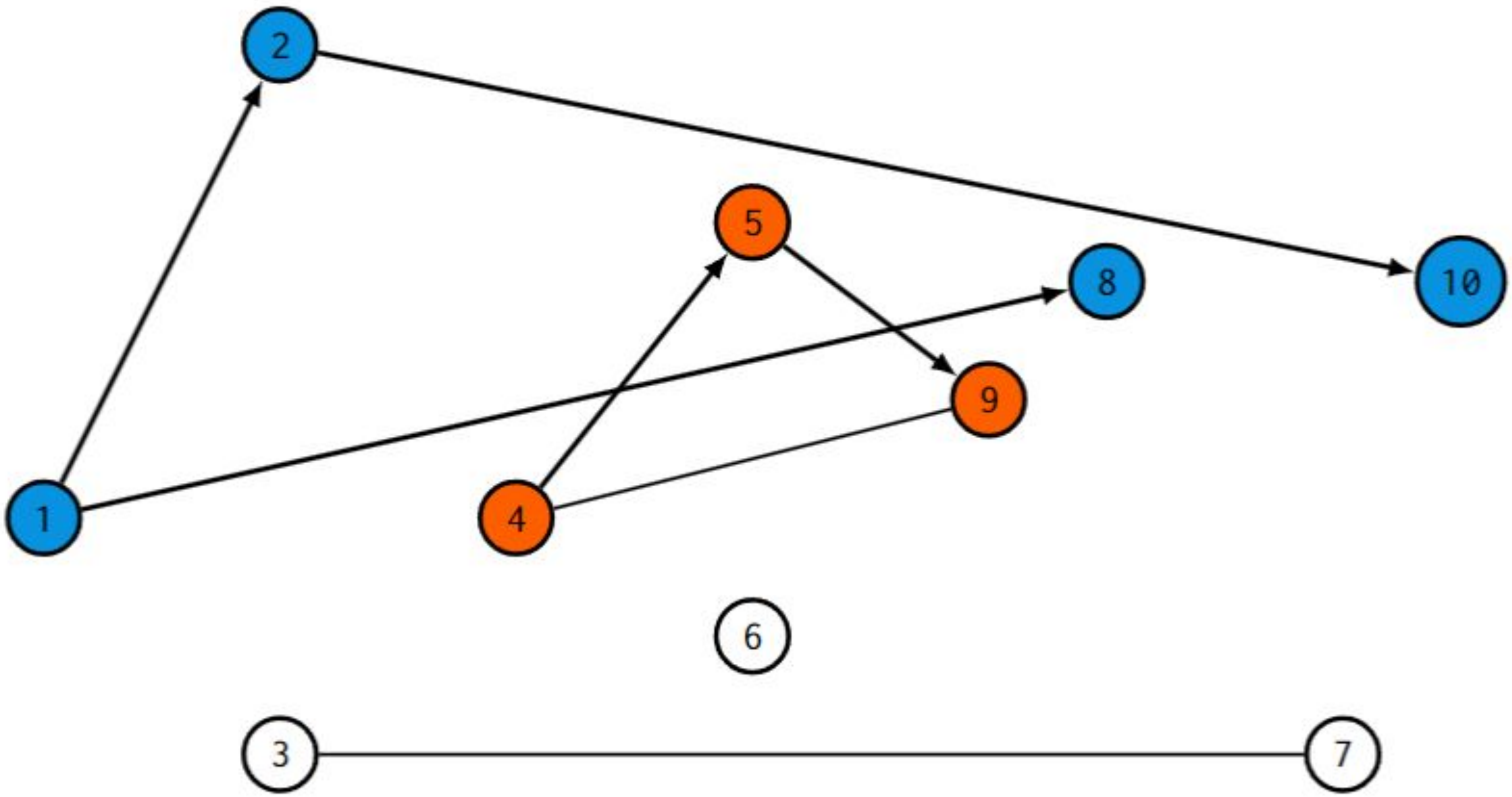
Componentes Conectados



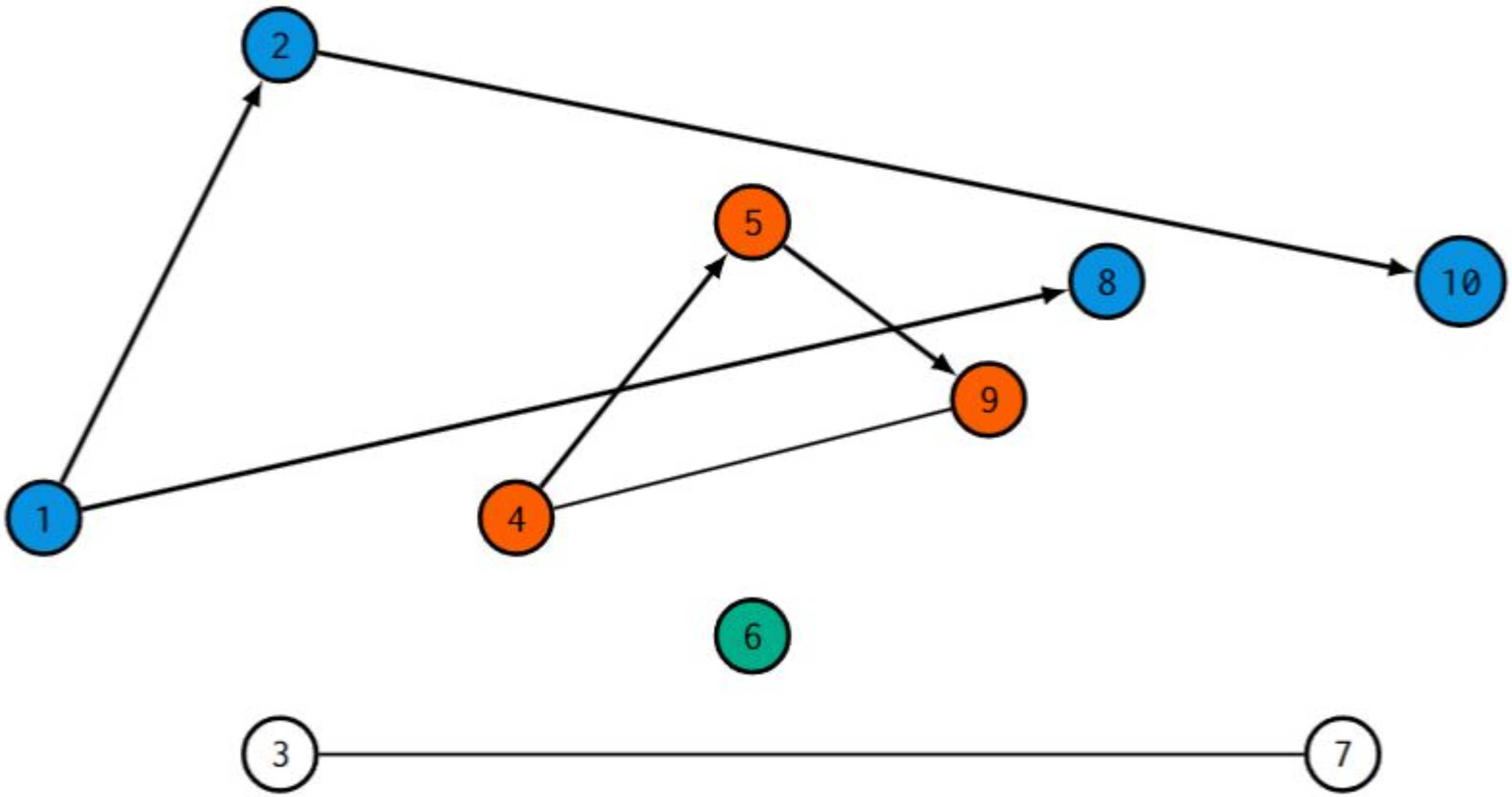
Componentes Conectados



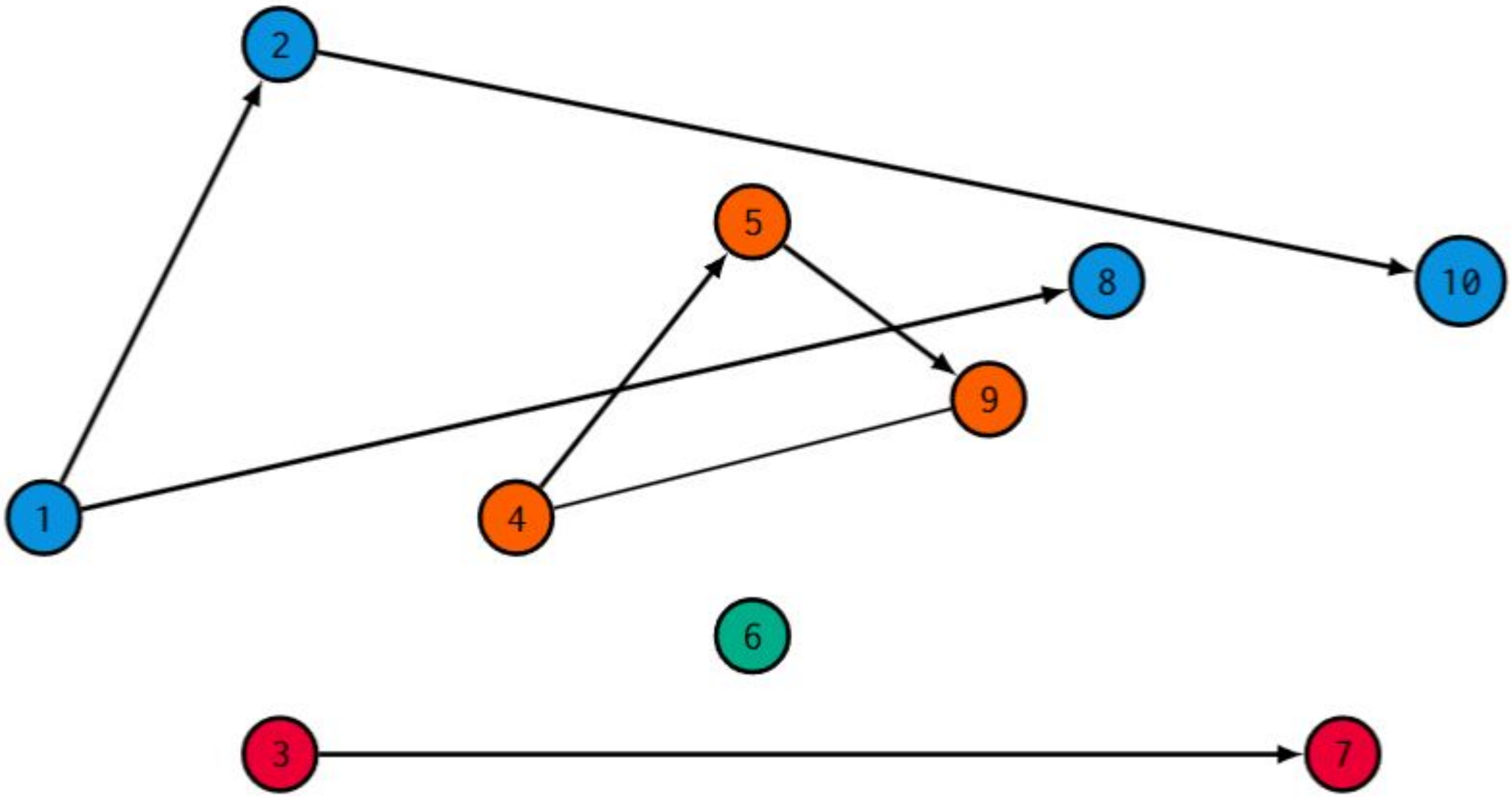
Componentes Conectados



Componentes Conectados



Componentes Conectados



Componentes Conectados

- **Implementação**
  - Para contabilizar o número de componentes conectados, basta executar a DFS para cada vértice ainda não visitado do grafo

Componentes Conectados

```
int compoentes_conectados()
{
    int quantidade = 0;

    for (int u = 1; u <= N; u++) {
        if (!visitado[u]) {
            quantidade++;
            cout << "Componente " << quantidade << " a partir de " << u << endl;
            dfs(u);
        }
    }
    return quantidade;
}
```

Componentes Conectados

- Grafo conectado
    - Por último, um grafo não-direcionado G é conectado se, e somente se, G tem um único componente conectado

Conclusão