Departamento de Computação

Programação Competitiva unesp

Introdução à Teoria dos Grafos

Laboratório de Programação Competitiva I

Pedro Henrique Paiola

Rene Pegoraro

Wilson M Yonezawa

Arissa Yoshida

Nicolas Barbosa Gomes

Luis Henrique Morelli

Definição

- Um grafo é uma abstração matemática que representa situações reais através de um diagrama, buscando representar a relação entre pares de elementos.

- Formalmente, um grafo $\boldsymbol{G}$ é um par $(\boldsymbol{V}, \boldsymbol{A})$ em que:
  - $\boldsymbol{V}$ é um conjunto de **vértices (nós)**;
  - $\boldsymbol{A}$ é um conjunto de **arestas** do tipo (u, v) com $\boldsymbol{u}$ e $\boldsymbol{v} \in \boldsymbol{V}$.

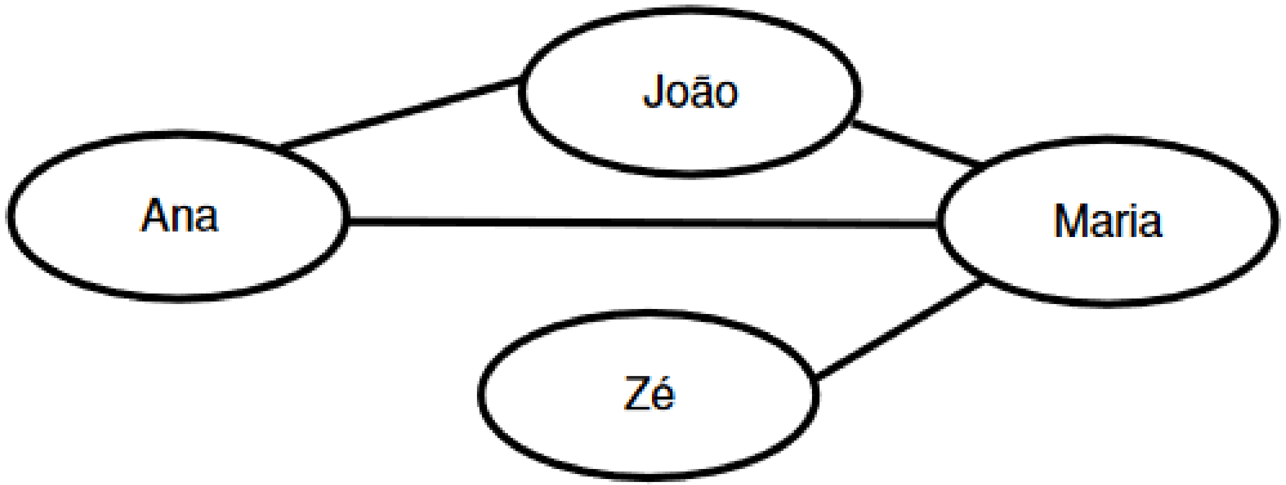
- Vértice: representa um elemento em si.
- Aresta: representa o relacionamento entre um par de elementos.

Definição

Exemplo de grafo: `G = (V, E)`

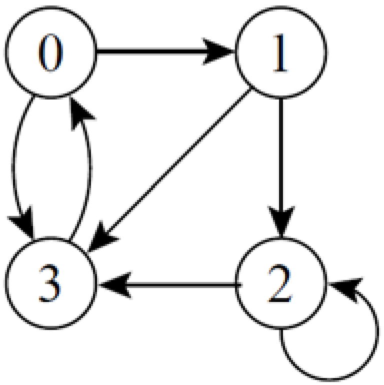
`V = {Ana, João, Maria, Zé}`

`E = {(Ana,João), (Ana,Maria),(João,Maria),(Maria,Zé)}`



Grafo orientado

- Um **grafo orientado** $\boldsymbol{G}$, também chamado de **grafo direcionado** ou **dígrafo**, é aquele em que o conjunto de arestas $\boldsymbol{A}$ é uma relação binária em $\boldsymbol{V}$, isto é, um conjunto finito de pares ordenados de vértices.
- Uma aresta (u, v) "sai" do vértice u e "entra" no vértice v. Nesse caso, dizemos que v é adjacente à u.
- Podem existir arestas de um vértice para ele mesmo (*self-loop* ou laço)

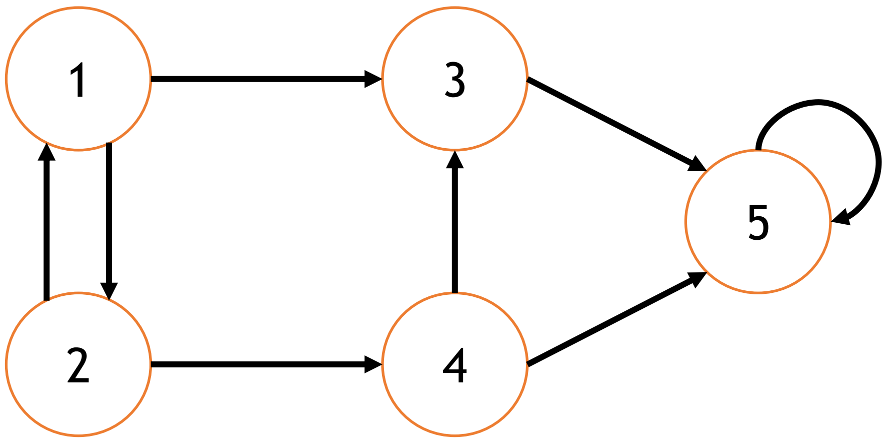


Grafo orientado

$G = (V, A)$

V = {1, 2, 3, 4, 5}

A = {(1,2), (1,3), (2, 1), (2, 4), (3,5), (4, 3), (4, 5), (5, 5)}



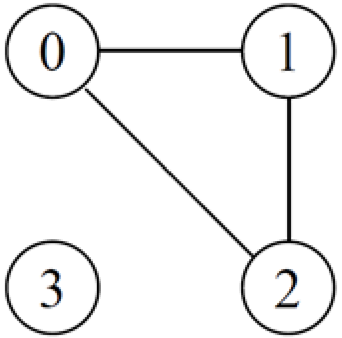
1 é adjacente à 2 e 2 é adjacente à 1

3 é adjacente à 1, mas 1 não é adjacente à 3

5 é adjacente a ele mesmo (laço)

Grafo não orientado

- Um **grafo não orientado G**, ou não direcionado, é aquele em que o conjunto de arestas A é um conjunto finito de pares não ordenados de vértices.
- (u, v) e (v, u) representam uma única aresta.
- Laços não são permitidos.

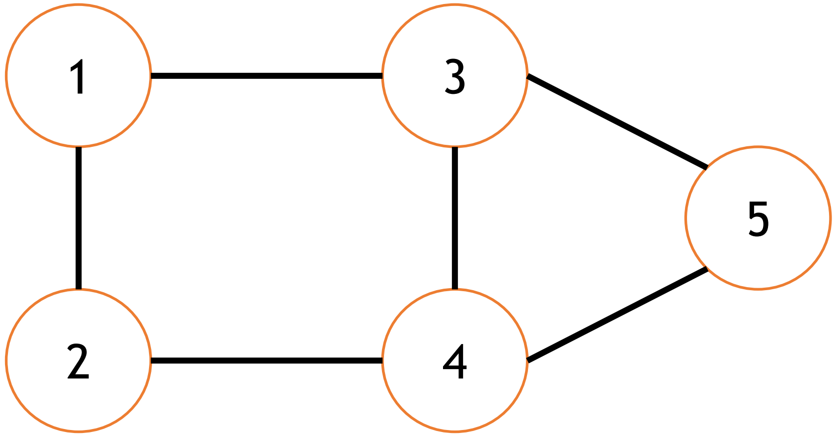


Grafo não orientado

$G = (V, A)$

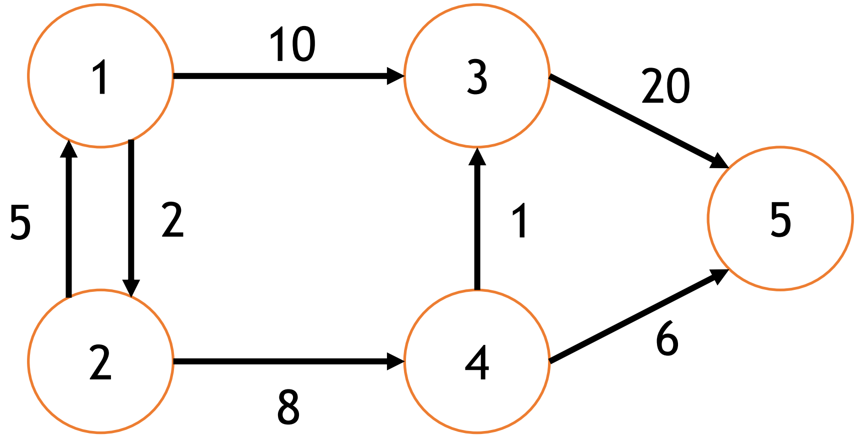
V = {1, 2, 3, 4, 5}

A = {(1,2), (1,3), (2,4), (3,4), (4,5)}



Grafo ponderado

- Um **grafo ponderado** é um grafo que possui **pesos** associados às arestas;
- Pode ser direcionado ou não;
- Os pesos podem representar, por exemplo, custos ou distâncias.



Grau de um vértice

- Em um grafo não direcionado:

$$grau(v) = \text{número de arestas que incidem em } v$$

- Em um grafo direcionado:

$$grau(v) = grau\_entrada(v) + grau\_saída(v)$$

em que

$grau\_entrada(v)$ = número de arestas que entram em v

$grau\_saída(v)$ = número de arestas que saem em v

Grau de um vértice

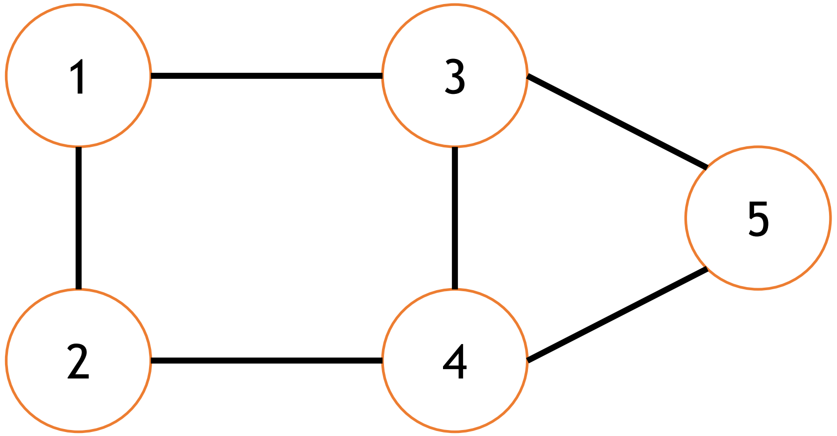
$grau(1) = 2$

$grau(2) = 2$

$grau(3) = 3$

$grau(4) = 3$

$grau(5) = 2$



Grau de um vértice

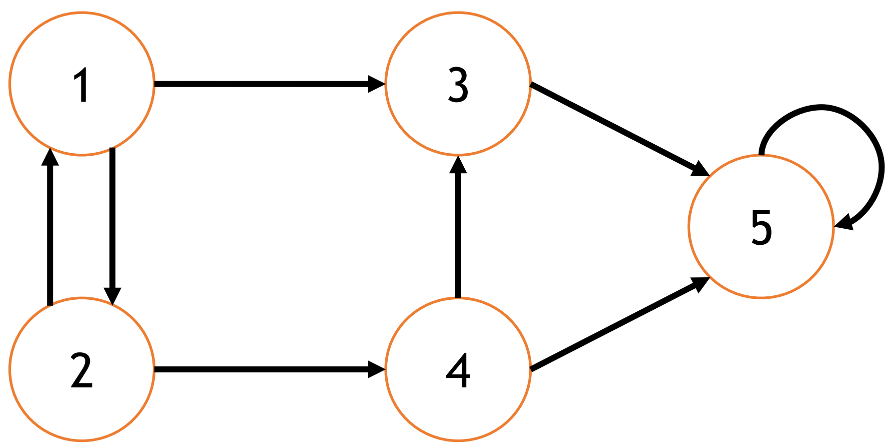
$$grau(1) = 1 + 2 = 3$$

$$grau(2) = 1 + 2 = 3$$

$$grau(3) = 2 + 1 = 3$$

$$grau(4) = 1 + 2 = 3$$

$$grau(5) = 3 + 1 = 4$$



Caminho entre vértices

- Um caminho é uma sequência de vértices conectados por arestas.
- De um vértice x a um vértice y, por exemplo, podemos ter um caminho $(v_0, v_1, \dots, v_k)$ em que $x = v_0$ e $y = v_k$;
- O comprimento do caminho é a quantidade de arestas que o formam.

Caminho entre vértices

- Exemplos de caminhos:

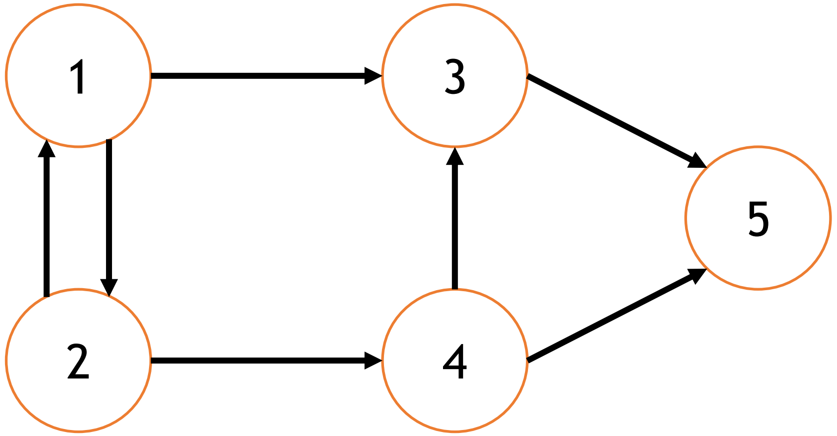
$(1, 3, 5)$

$(2, 4, 3)$

$(1, 2, 4, 3, 5)$

$(1, 2, 4, 5)$

- Perceba que de um vértice a outro pode existir mais de um caminho possível.



Caminho entre vértices

- Exemplos de caminhos:

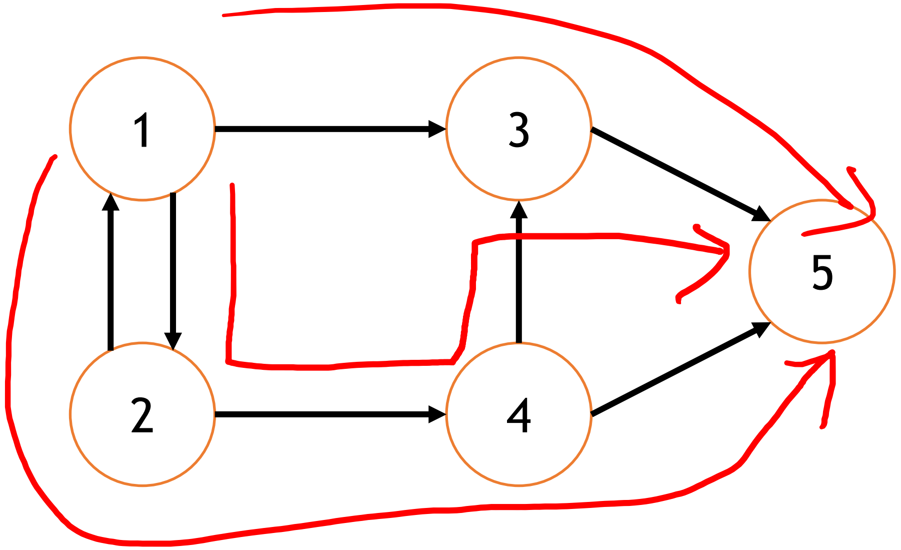
$(1, 3, 5)$

$(2, 4, 3)$

$(1, 2, 4, 3, 5)$

$(1, 2, 4, 5)$

- Perceba que de um vértice a outro pode existir mais de um caminho possível.

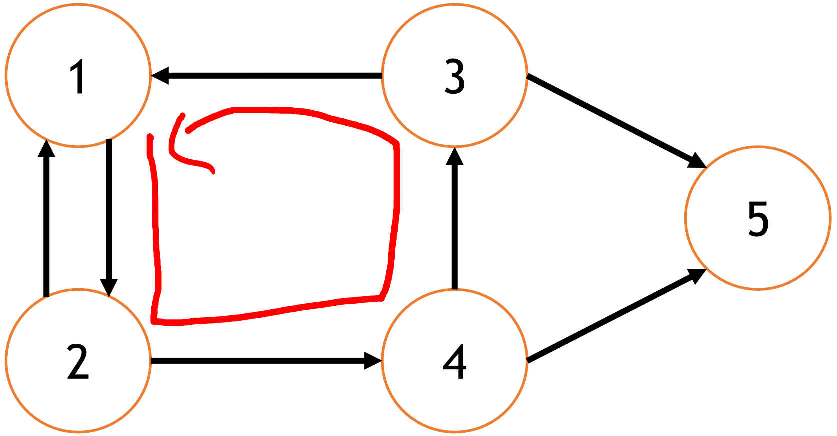


Caminho entre vértices

- Um caminho é **simples** se todos os vértices do caminho são distintos.
- Um caminho $(v_0, v_1, \dots, v_k)$ forma um ciclo se $v_0 = v_k$.
- Exemplo:

$$(1, 2, 4, 3, 1)$$

- Um grafo sem ciclos é chamado **acíclico**.



Implementação

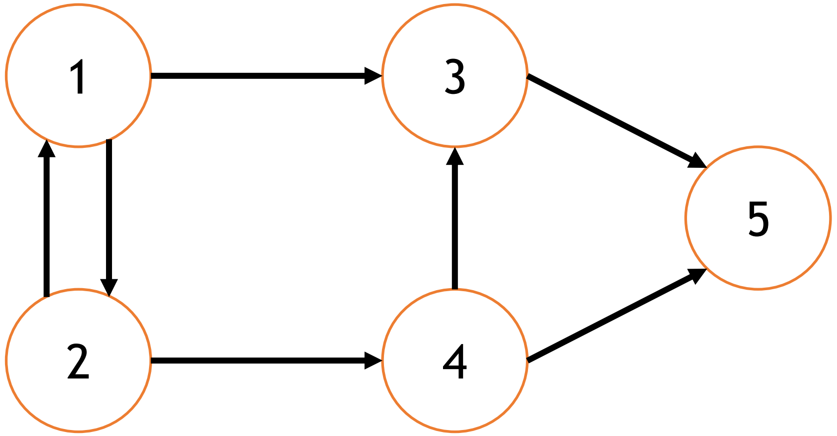
- Principal preocupação: como representar o conjunto de arestas A?
- Duas formas usuais:
  - Matriz de adjacência
  - Lista de adjacência

Matriz de Adjacência

- Para um grafo de n vértices, podemos utilizar uma matriz $M_{n \times n}$.
- $M_{i,j} = 1 \leftrightarrow j$ é adjacente a i.
  - $M[i][j] = 1$ se há uma aresta do nó i ao nó j.
  - $M[i][j] = 0$ se não há uma aresta do nó i ao nó j.
- Quando o grafo é não direcionado, a matriz é simétrica.
- Para grafos ponderados, a matriz de adjacência pode ser utilizada para armazenar os pesos das arestas (desde que não haja peso nulo).

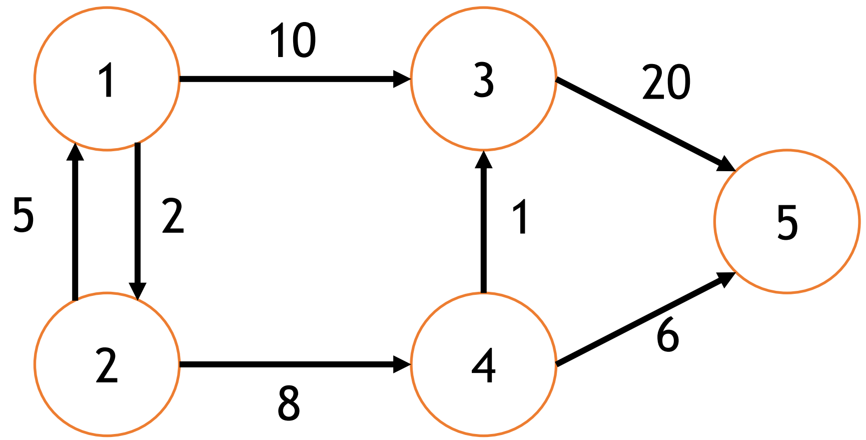
Matriz de Adjacência

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



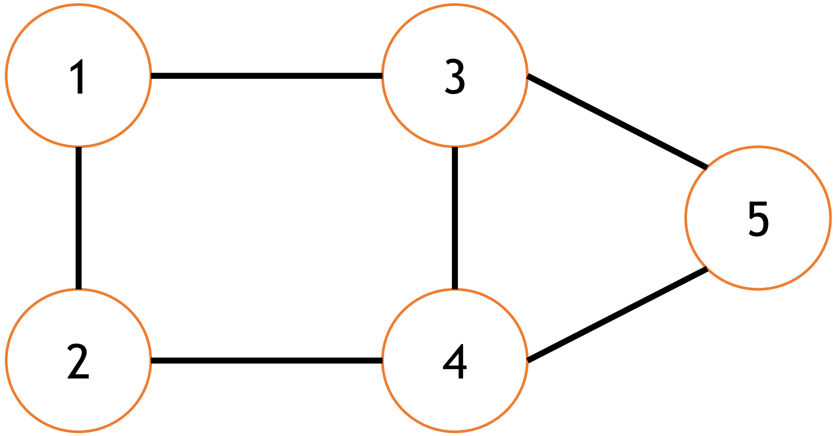
Matriz de Adjacência

	1	2	3	4	5
1	0	2	10	0	0
2	5	0	0	8	0
3	0	0	0	0	20
4	0	0	1	0	6
5	0	0	0	0	0



Matriz de Adjacência

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

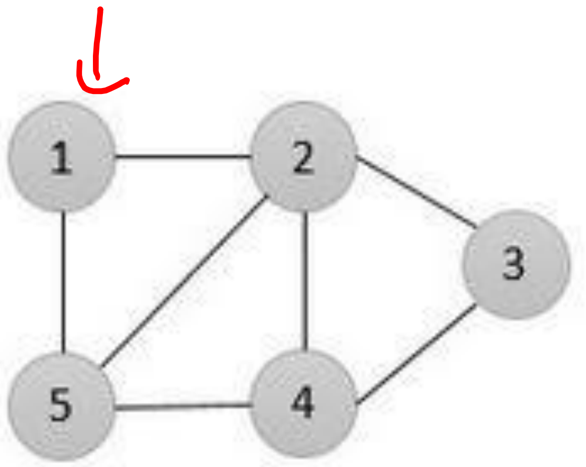


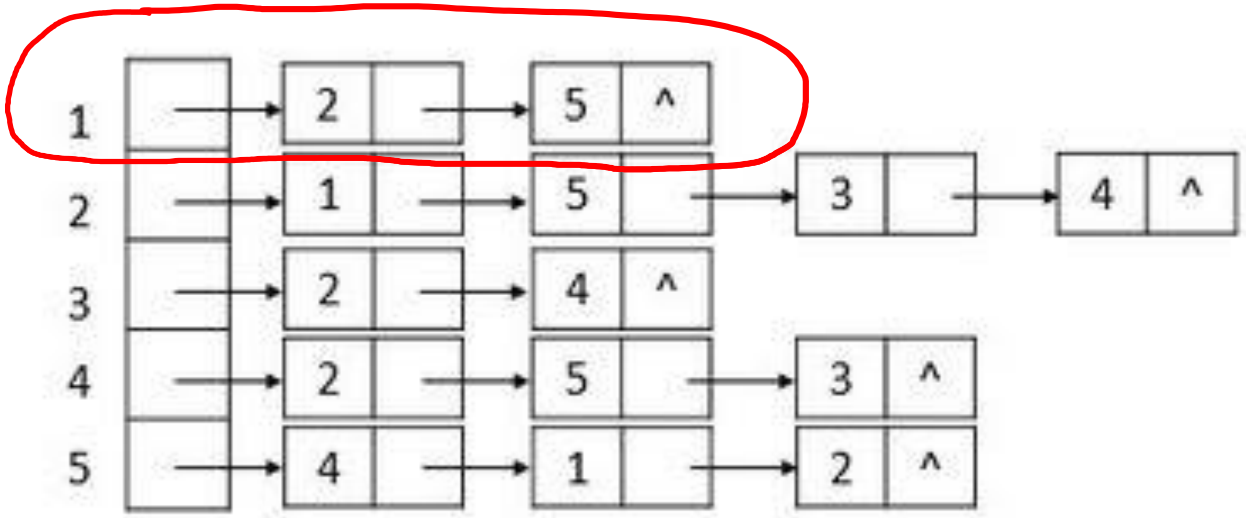
Matriz de Adjacência

- Vantagens:
  - Implementação simples;
  - Verificar se existe uma aresta (i, j) pode ser feito em tempo constante.
  - Inserção ou remoção de arestas também podem ser realizadas com custo constante.
- Desvantagens:
  - Espaço necessário: $O(|V|^2)$
  - Tempo para acessar todos os nós adjacentes à um vértice v qualquer: $O(|V|)$

Lista de Adjacência

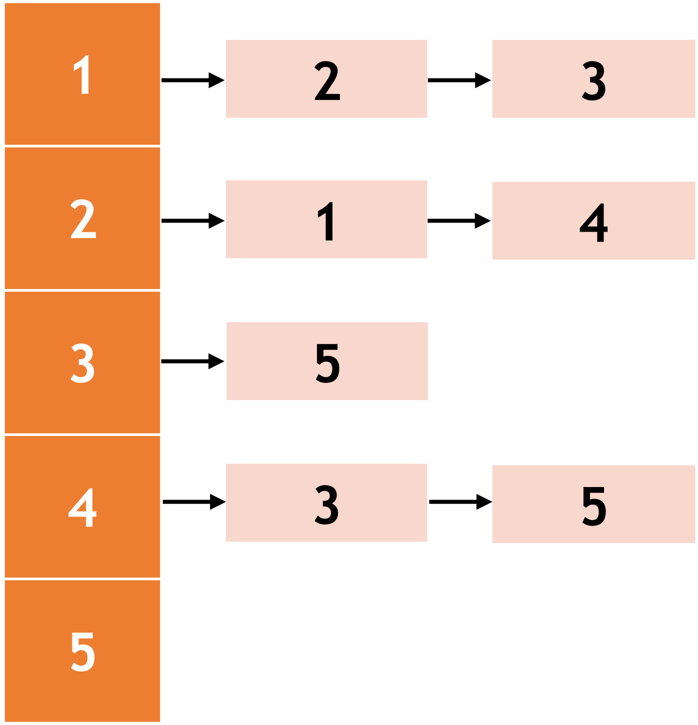
- Maneira mais comum de se representar um grafo;
- Para cada vértice é armazenada uma lista de vértices adjacentes.

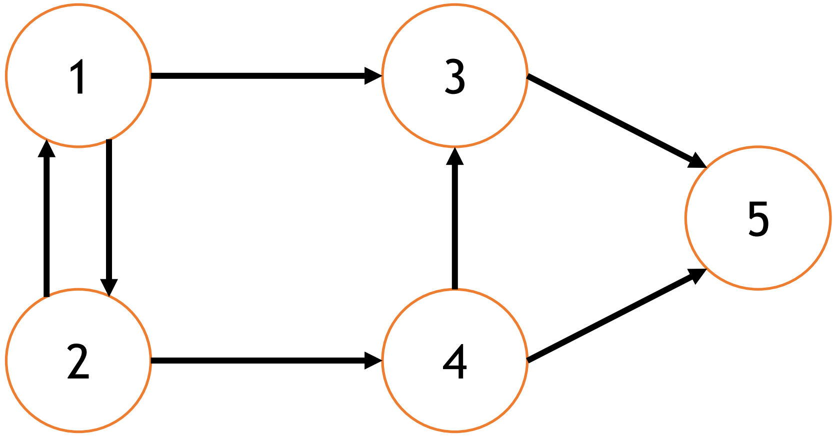




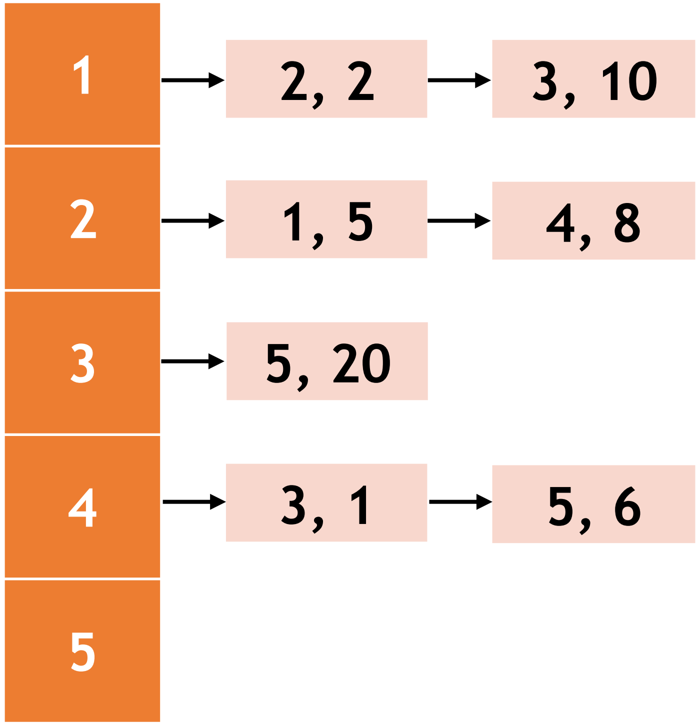
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

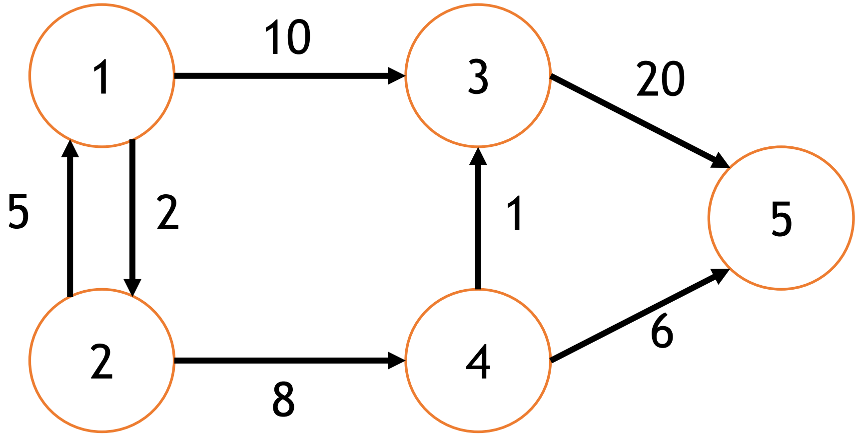
Lista de Adjacência



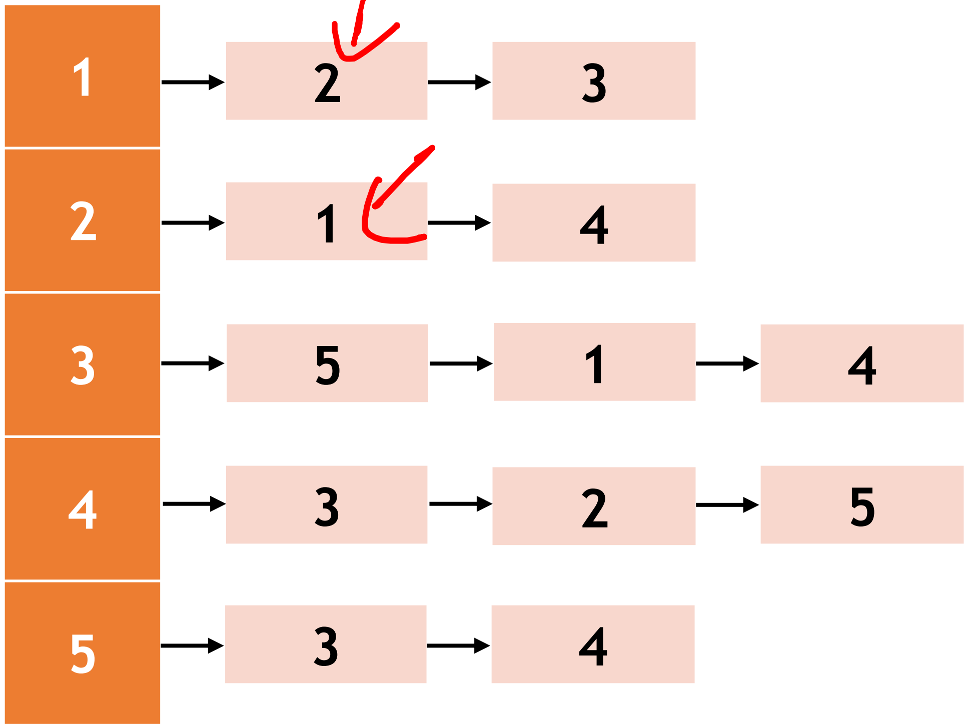


Lista de Adjacência

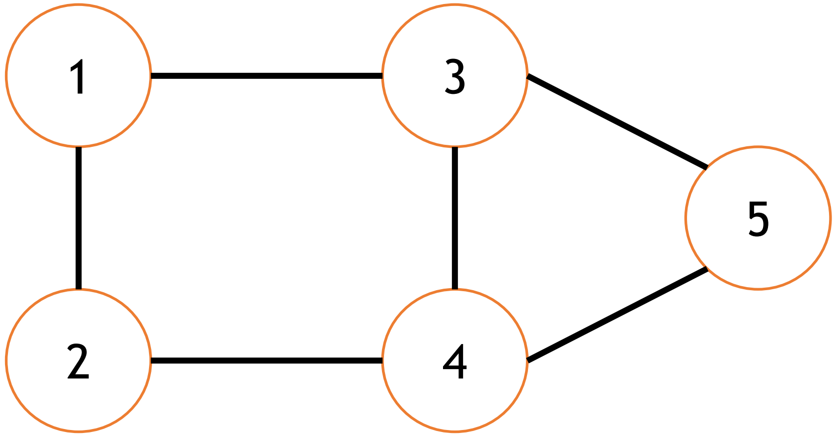




Lista de Adjacência



(1, 2)



Lista de Adjacência

```
typedef struct{
    int v;    //vértice adjacente
    int w;    //peso
} TAdj;
```

} ARESTAS

```
vector<TAdj> adj[MAX_V]; //Lista de adjacência
int grau[MAX_V];          //número de arestas do vértice

void initGrafo(int qtdeVertices){
    memset(grau, 0, sizeof(grau));
    for(int i = 0; i < qtdeVertices; i++)
        adj[i].clear();
}
```

Lista de Adjacência

```
//Cria aresta de a para b, com peso w
void aresta(int a, int b, int w){
    TAdj aux;
    aux.v = b;
    aux.w = w;
    grau[a]++;
    adj[a].push_back(aux);
    //Se o grafo for não orientado, também adicionamos a aresta (b, a) co
m peso w
}
```

Lista de Adjacência

- Vantagens:
  - É possível iterar pelos nós adjacentes facilmente;
  - Os algoritmos de grafos, no geral, se tornam mais eficientes;
  - Economia de espaço, em relação a matriz de adjacência.
- Desvantagens:
  - Implementação mais complexa;
  - Verificar de um vértice v é adjacente a outro vértice u não pode mais ser realizado em tempo constante.

Busca em Profundidade (DFS)

- Generalização da busca em profundidade em árvores.
- Dado um grafo G e um nó inicial s, a estratégia é explorar o grafo em profundidade, visitando as arestas do vértice mais recentemente descoberto que levam a vértices ainda inexplorados.
- Implementação: recursiva ou iterativa com auxílio de pilha.
- Complexidade: $O(V + A)$ para lista de adjacência e $O(V^2)$ para matriz de adjacência.

- Possíveis usos: encontrar caminhos, contagem de componentes conexas e detecção de ciclos.

Busca em Profundidade (DFS)

- Pseudo-código:

DFS(v)
- Marcar v como visitado
- Para cada vértice u adjacente à v
    - Se u não foi visitado
        - DFS(u)

Busca em Profundidade (DFS)

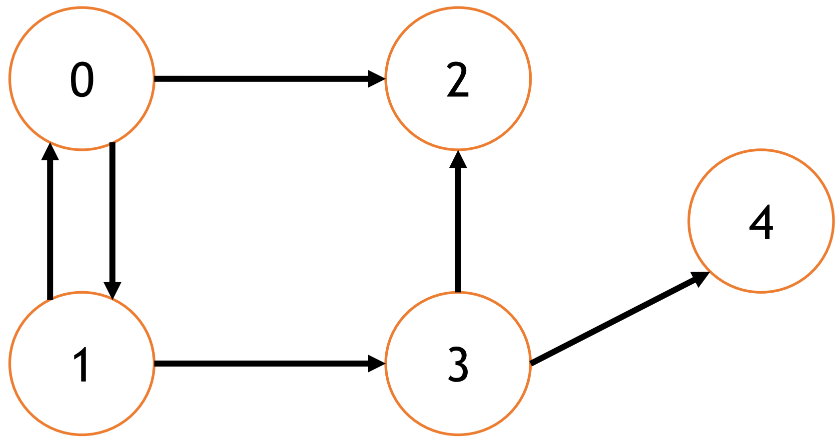
```
int visitado[MAX_V];
int p[MAX_V];
int ordemVis;

void initDfs(){
    memset(visitado, 0, sizeof(visitado));
    memset(p, -1, sizeof(p));
    ordemVis = 0;
}
```

Busca em Profundidade (DFS)

```
void dfs(int s){
    visitado[s] = ++ordemVis;
    for(auto t : adj[s]){
        if (visitado[t.v] == 0){
            p[t.v] = s;
            dfs(t.v);
        }
    }
}
```

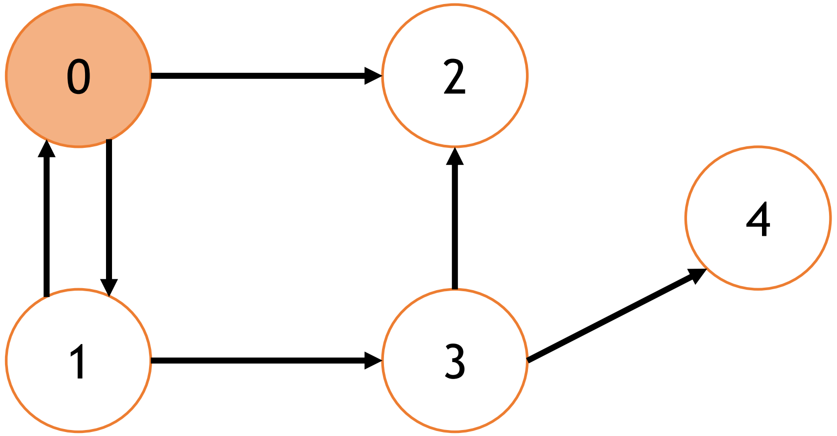
Busca em Profundidade (DFS)

v	vis	p
0		
1		
2		
3		
4		



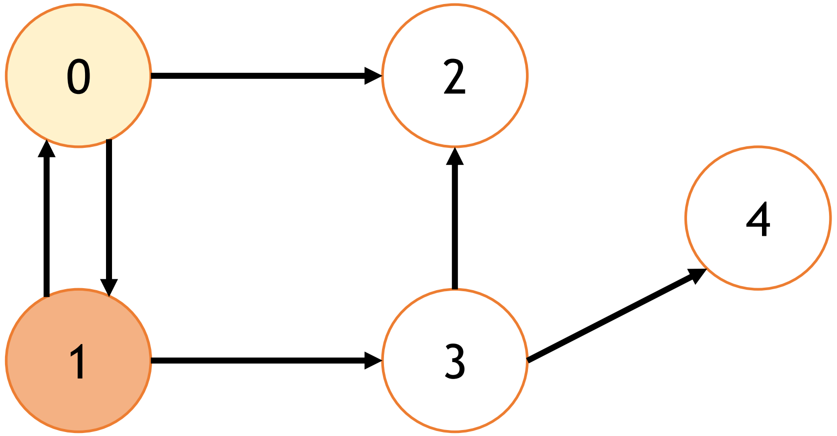
Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1		
2		
3		
4		



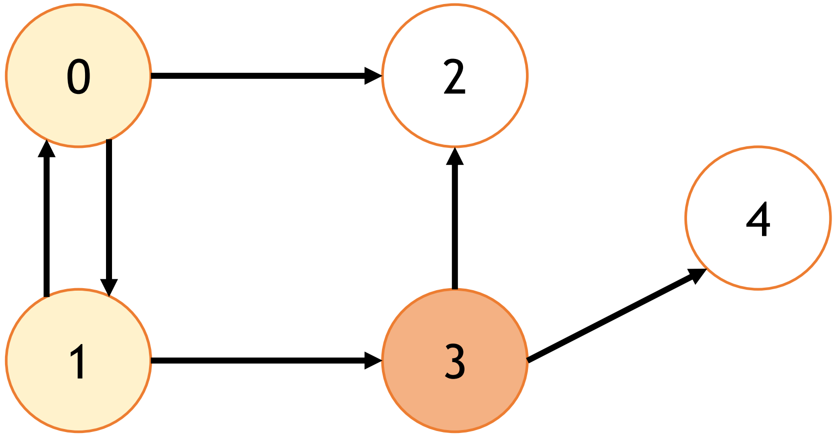
Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1	2	0
2		
3		
4		



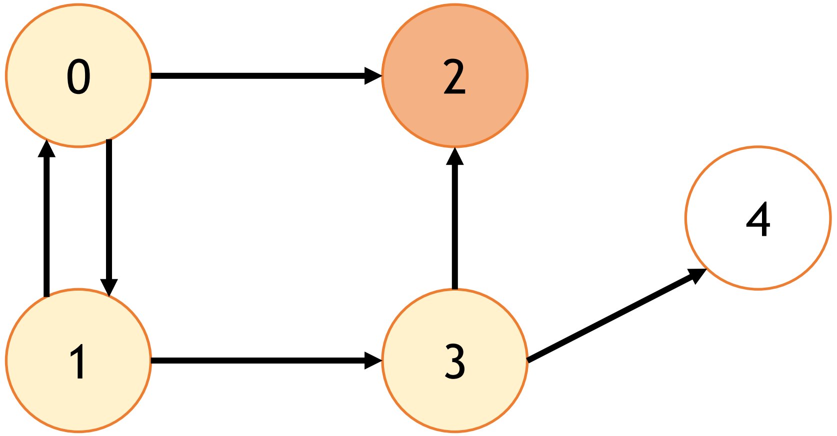
Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1	2	0
2		
3	3	1
4		



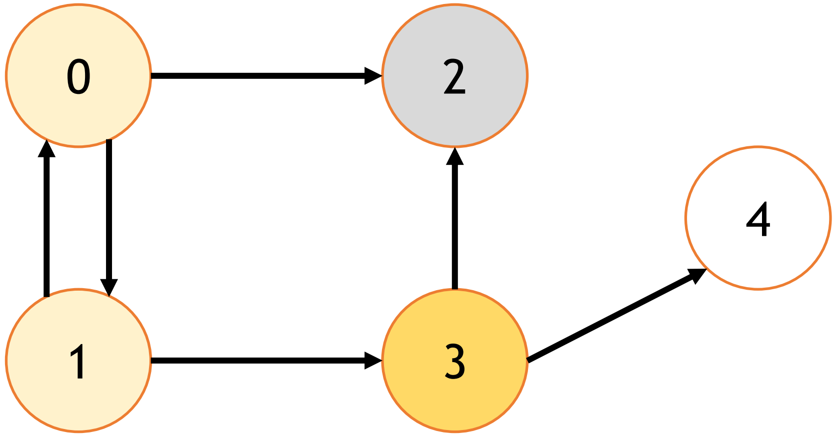
Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1	2	0
2	4	3
3	3	1
4		



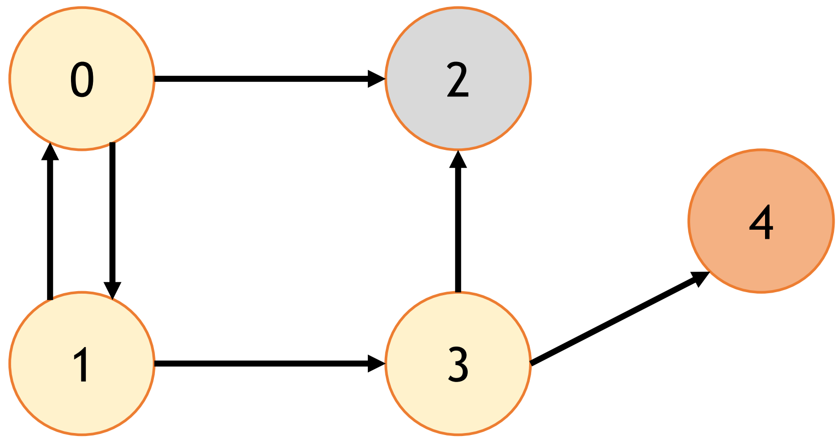
Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1	2	0
2	4	3
3	3	1
4		



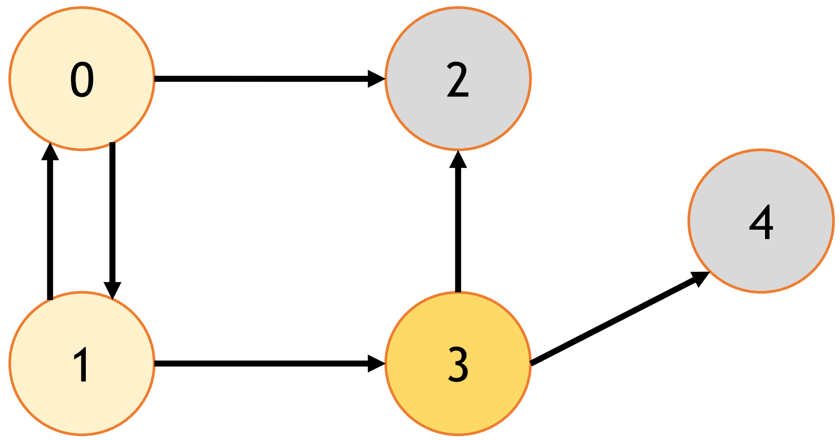
Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1	2	0
2	4	3
3	3	1
4	5	3



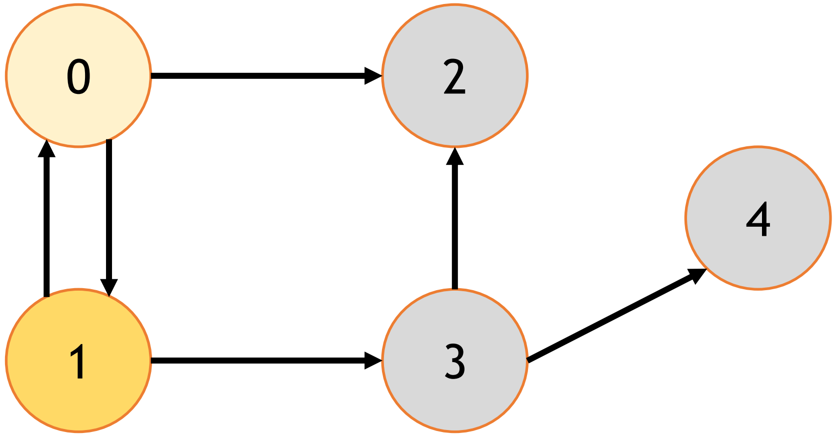
Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1	2	0
2	4	3
3	3	1
4	5	3



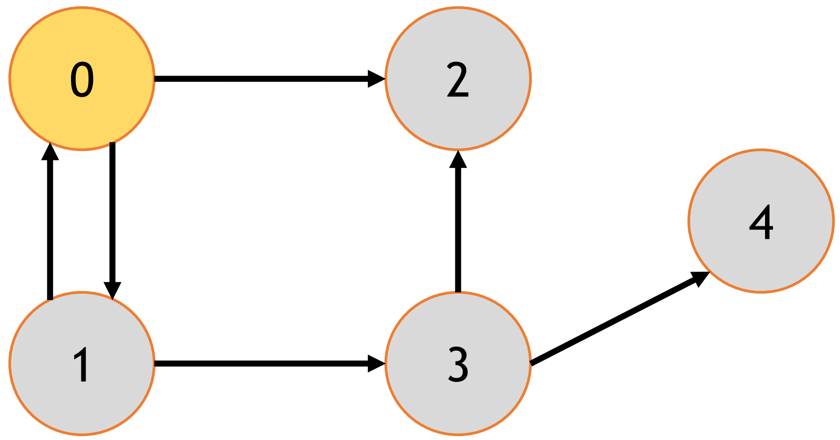
Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1	2	0
2	4	3
3	3	1
4	5	3



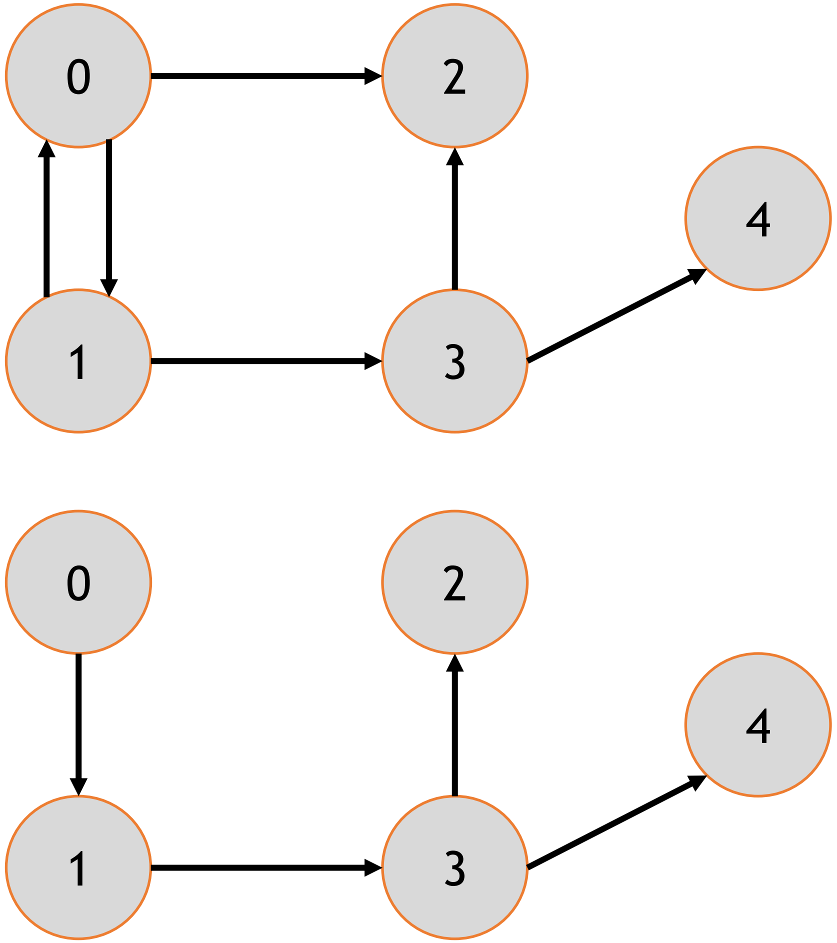
Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1	2	0
2	4	3
3	3	1
4	5	3



Busca em Profundidade (DFS)

v	vis	p
0	1	-1
1	2	0
2	4	3
3	3	1
4	5	3



Busca em Largura (BFS)

- Generalização da busca em largura em árvores.
- Dado um grafo G e um nó inicial s, a estratégia é explorar o grafo por "nível". Vamos definir nível de v como sendo o comprimento do menor caminho do vértice inicial até v.
- Implementação: iterativa com auxílio de fila.
- Complexidade: $O(V + A)$ para lista de adjacência e $O(V^2)$ para matriz de adjacência.

- Possíveis usos: encontrar o menor caminho (em número de arestas) entre vértices.

Busca em Largura (BFS)

- Pseudo-código:

BFS(v)

- Enfileirar v na fila Q
- Enquanto Q não estiver vazia
    - Desenfileirar o vértice u de Q
        - Marcar u como visitado
        - Para cada vértice w adjacente à u
            - Se w ainda não foi visitado
                - Enfileirar w na fila Q

Busca em Largura (BFS)

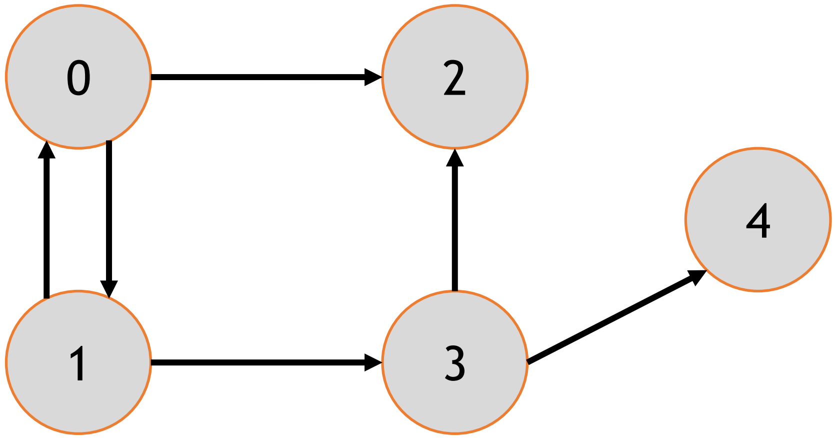
```
int d[MAX_V];         //armazena a distância do nó inicial até cada nó i

void bfs(int inicio)
{
    int s, t;
    queue<int> Q;

    memset(visitado, 0, sizeof(visitado));
    memset(p, -1, sizeof(p));
    d[inicio] = 0;
    visitado[inicio] = ++ordemVis;

    Q.push(inicio);
```

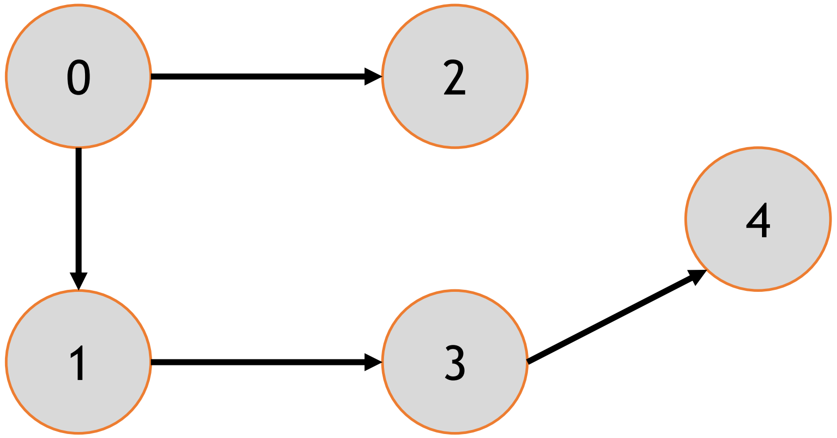
Busca em Largura (BFS)

```
    while(!Q.empty()){
        s = Q.front();
        Q.pop();
        for(auto t : adj[s]){
            if (visitado[t] == 0){
                visitado[t] = ++ordemVis;
                d[t] = d[s] + 1;
                p[t] = s;
                Q.push(t);
            }
        }
    }
}
```

Busca em Profundidade (DFS)

v	vis	p	d
0	1	-1	0
1	2	0	1
2	3	0	1
3	4	1	2
4	5	3	3





Conexidade

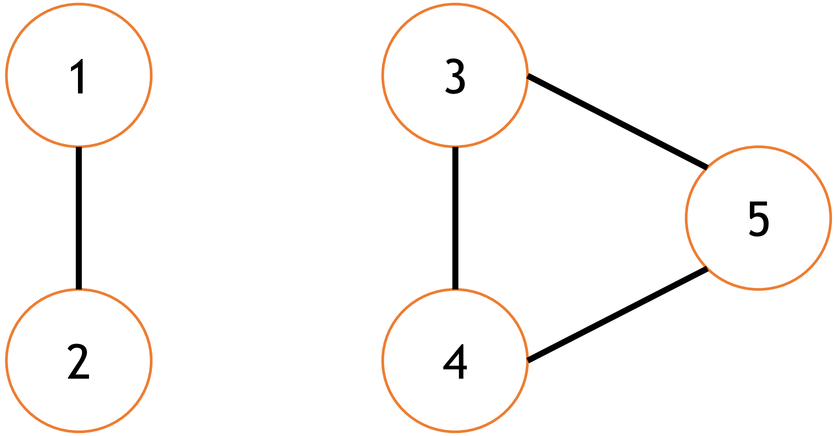
- Um grafo **não direcionado** $G = (V, A)$ é conexo sse existe um caminho em G entre todos os pares de vértices.

conexo

desconexo

Conexidade

- Um grafo $G' = (V', A')$ é um **subgrafo** de $G = (V, A)$ sse $V' \subseteq V$ e $A' \subseteq A$.
- Um subgrafo conexo de G é chamado de **componente conexa** de G.
- O grafo a seguir, por exemplo, possui duas componentes conexas.

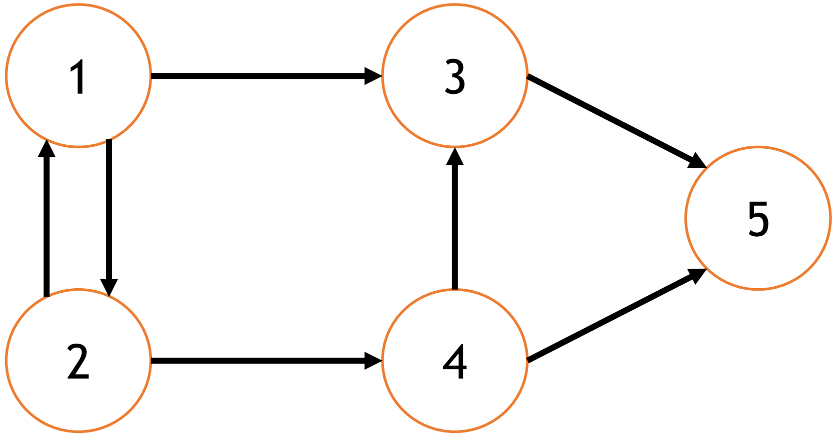


Conexidade

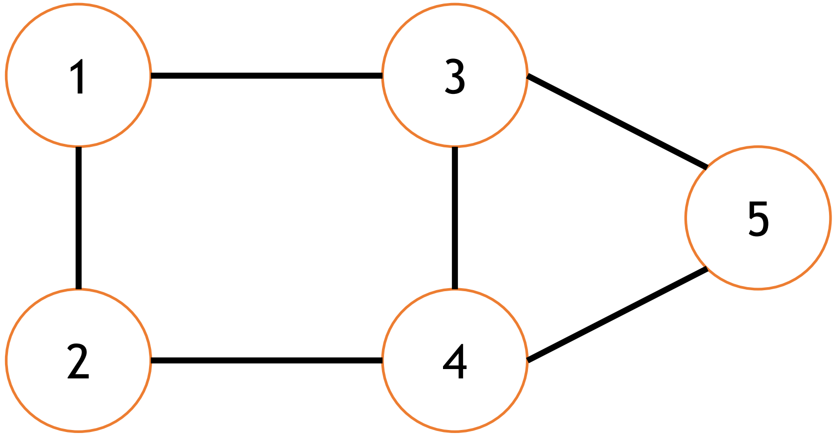
- Para grafos direcionados, definimos dois tipos de conexidade: forte e fraca.
- Um grafo direcionado é **fortemente conexo** se existir um caminho entre todos os pares de vértices do grafo.
- Um grafo direcionado é **fracamente conexo** se o seu grafo não direcionado subjacente (retirando a orientação das arestas) é conexo.

Caminho entre vértices

- Grafo



- Grafo não direcionado subjancente:



Esse grafo não é fortemente conexo, mas é fracamente conexo.

Conexidade

- Como determinar se um **grafo não direcionado** é conexo?
  - Basta fazer um percurso no grafo (em profundidade ou em largura), a partir de qualquer nó.
  - Se neste percurso todos os vértices foram visitados, então ele é conexo.
  - Caso contrário, não é, e os vértices visitados formam uma componente conexa.

Conexidade

- Como determinar se um **grafo direcionado** é fortemente conexo?
  - Deve-se fazer um percurso no grafo para cada vértice, e cada um desses percursos deve conseguir visitar todos os vértices do grafo.

A Bug's Life (Spoj BUGLIFE)

- **Problema:** estudando uma espécie de inseto, o professor Hopper criou a hipótese que insetos de um determinado gênero interagem apenas com o gênero oposto.
- **Objetivo:** dada diversas interações entre os insetos (numerados), determinar se a hipótese do professor é falsa ou não há nenhuma evidência que o contrarie.

A Bug’s Life (Spoj BUGLIFE)

- **Solução:** primeiramente, vamos modelar este problema na forma de um grafo, em que os vértices representam os insetos e as arestas as interações lidas na entrada.
- Exemplo:

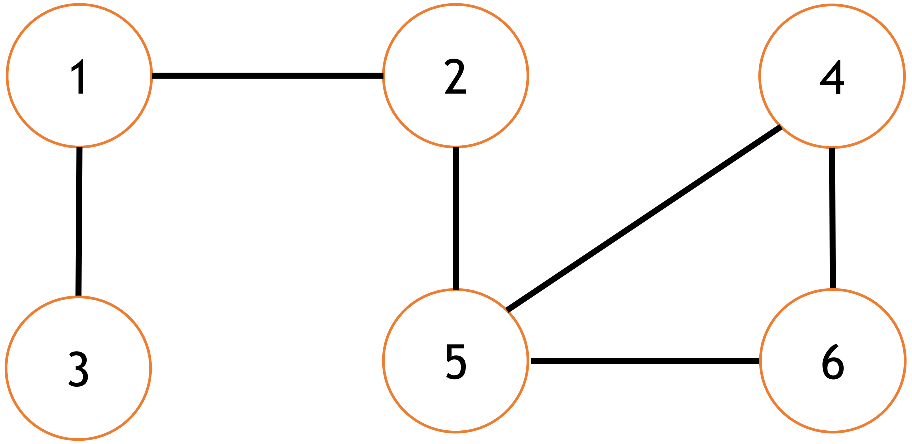
1 2

1 3

4 5

5 6

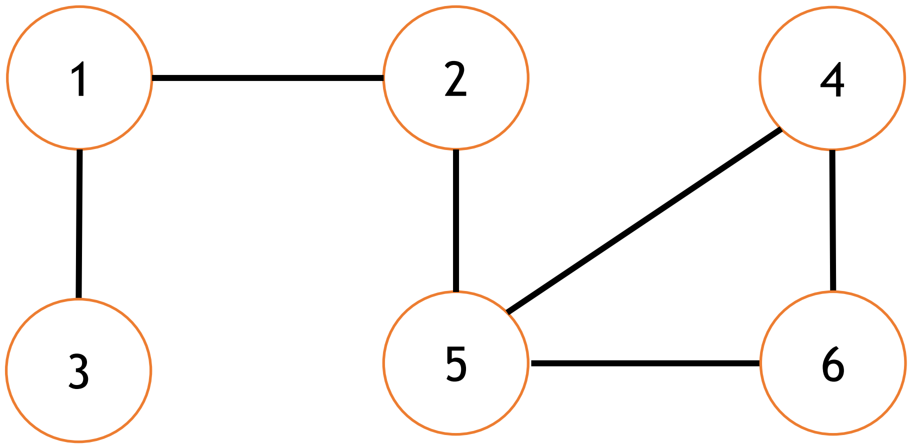
4 6



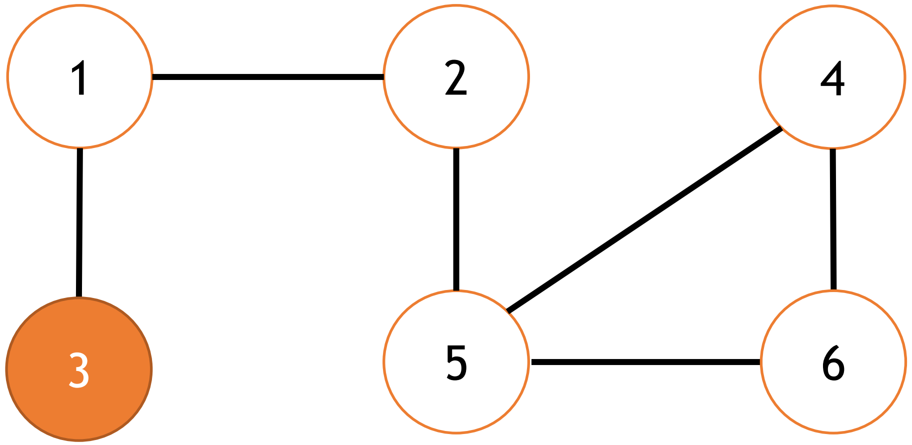
A Bug's Life (Spoj BUGLIFE)

- **Solução:** agora uma forma de solucionar este problema é tentar colorir o grafo com duas cores, de forma que dois nós adjacentes não possuam a mesma cor. Neste caso, cada cor representa um determinado gênero.

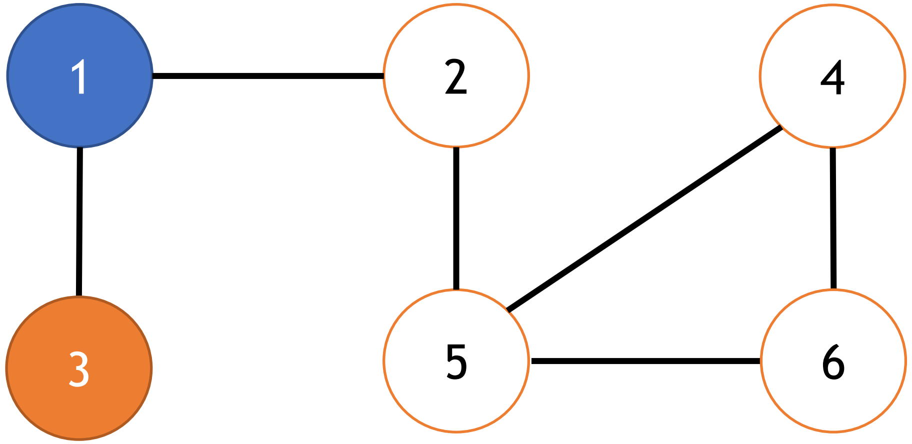
- Se durante a busca encontrarmos um nó adjacente já visitado com a mesma cor que o atual, então a hipótese do professor é falsa.
- Caso contrário, se conseguirmos pintar todo o grafo sem nenhum problema, então não encontramos nada que o contradiga.



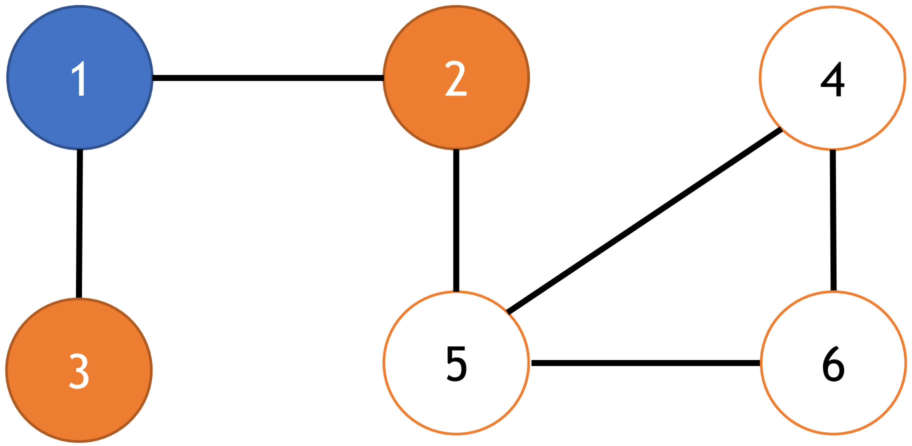
A Bug's Life (Spoj BUGLIFE)



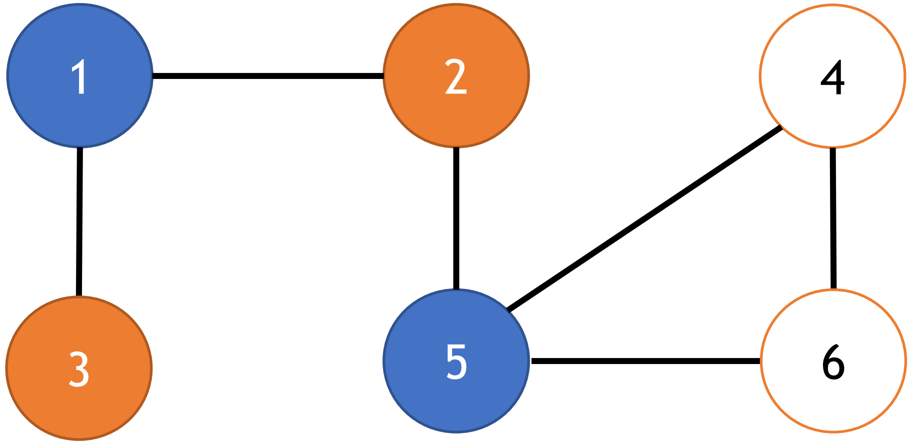
A Bug's Life (Spoj BUGLIFE)



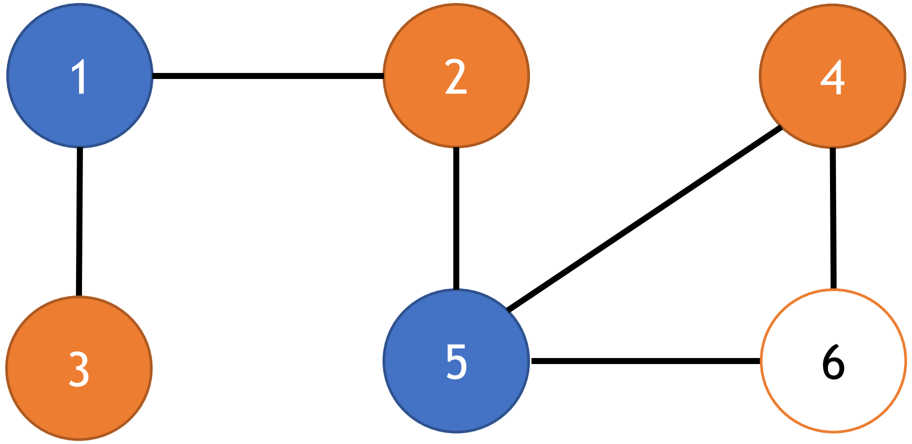
A Bug's Life (Spoj BUGLIFE)



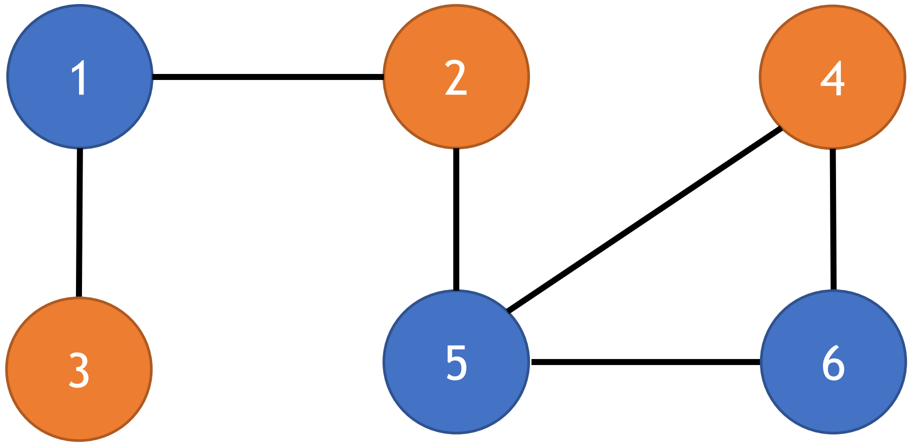
A Bug's Life (Spoj BUGLIFE)



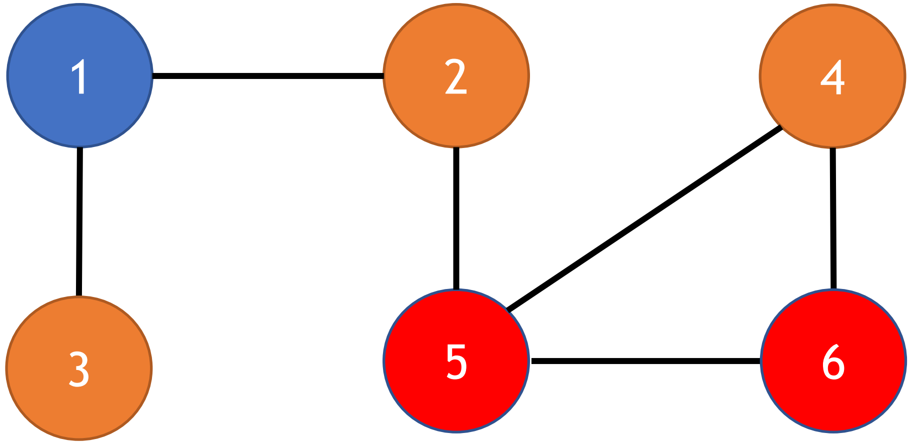
A Bug's Life (Spoj BUGLIFE)



A Bug's Life (Spoj BUGLIFE)



A Bug’s Life (Spoj BUGLIFE)



Sugestões

- Gravações de LPC I e II – 2020:
  - Introdução à Teoria dos Grafos
  - Seminário: Teoria dos Grafos (Amigos do Davizaum)
  - Pontes e Bellman-Ford
  - Problema do Fluxo Máximo
  - Ordenação Topológica
  - Emparelhamento máximo em grafos bipartidos
  - Menor Ancestral Comum (LCA)
- Material do GEMA (ICMC) - Vídeo
- Material do UnBallon (UnB)
- Vídeo: Busca em Grafos (MaratonUSP) – Giovana Delfino

Referências

Aulas de Estrutura de Dados II da Profª Drª Marcia Aparecida Zanoli Meira e Silva.

Matemática Discreta e Suas Aplicações. Kenneth H. Rosen.

Seminário sobre Introdução a Teoria dos Grafos. Davi Neves, Giovani Candido, Luis Morelli e Luiz Sementille.

Biblioteca de códigos de Thiago Alexandre Domingues de Souza.

http://www.lcad.icmc.usp.br/~jbatista/scc210/Aula_Grafos1.pdf

http://www.lcad.icmc.usp.br/~jbatista/scc210/Aula_Grafos2.pdf

http://www4.pucsp.br/~jarakaki/grafos/Aula2.pdf

https://miltonborba.org/Algf/Grafos.htm

https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphs.html

https://www.obm.org.br/content/uploads/2017/01/Nivel1_grafos_bruno.pdf

http://www.inf.ufsc.br/grafos/definicoes/definicao.html