Aula 7:
Introdução a Grafos

Disciplina: Maratona de Programação 1

Profs. Edmilson Marmo e Luiz Olmes

edmarmo@unifei.edu.br, olmes@unifei.edu.br

UNIFEI
UNIVERSIDADE FEDERAL DE ITAJUBÁ

Nas aulas anteriores...

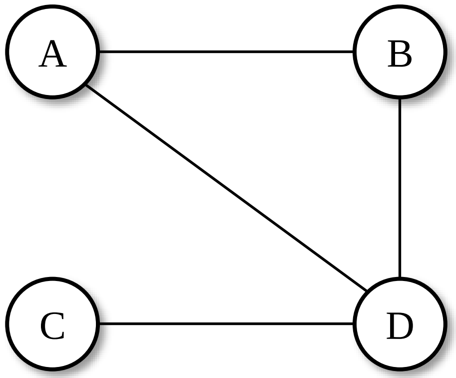
- **O QUE JÁ ESTUDAMOS?**
- Introdução à Maratona
- Problemas ad hoc
- Standard Template Library (STL)

- **OBJETIVOS:**
- Grafos
- Definição
- Representação
- Buscas:
  - Profundidade
  - Largura

Definição

- Um grafo é uma estrutura de dados formada por um conjunto de nós (também chamados de vértices), conectados por um conjunto de arestas (também chamadas de arcos).

- G = (V, A, f)
  - V: conjunto de vértices
  - A: conjunto de arestas
  - f: função que associa cada aresta a um par de nós.



Outra definição

- Abstração que permite codificar relacionamentos entre pares de objetos.
- Quais objetos?
- Quais relacionamentos?

Outra definição

- Abstração que permite codificar relacionamentos entre pares de objetos.

- Quais objetos?
- Quaisquer objetos!
  - Pessoas
  - Cidades
  - Empresas
  - Países
  - Páginas web
  - Filmes
  - etc.

- Quais relacionamentos?
- Quaisquer relacionamentos!
  - Amizade
  - Conectividade
  - Produção
  - Língua falada
  - Acessibilidade
  - Gênero
  - etc.

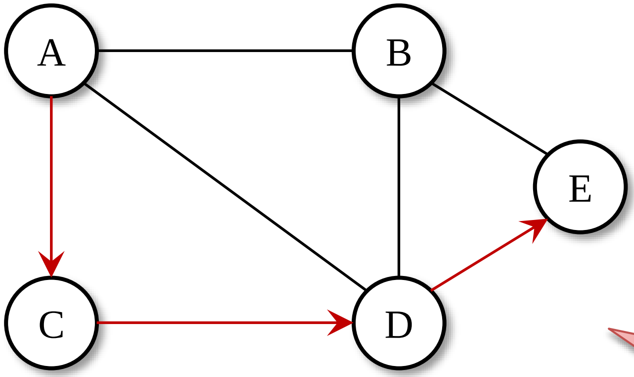
Grafos

- Uma grande variedade de problemas pode ser expressa com clareza através da representação de grafos:
  - Definição do roteiro mais curto para visitar as principais cidades de uma região.
  - Coloração de um mapa político, sem que áreas fronteiriças possuam a mesma cor.
  - Fluxo máximo de líquidos em sistemas de tubos.
  - Controle de tráfego de uma cidade.
  - Auxílio na busca de informações por search engines.
  - Rotas: menor número de vôos entre duas localidades.

- Basta que o problema seja modelado na forma de um grafo!

Terminologias

- Em um grafo, um caminho é uma sequência de arestas e vértices que partem de um nó A e levam até outro nó B.

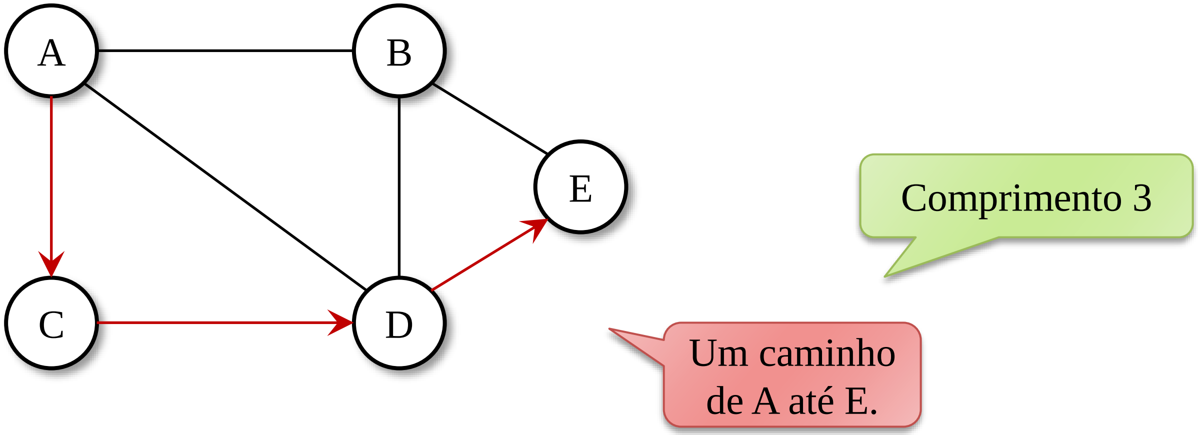


Um caminho de A até E.

Terminologias

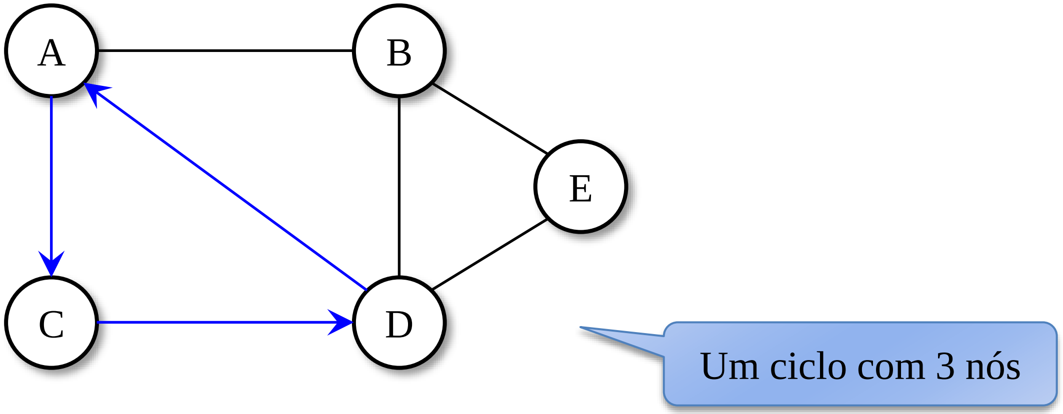
- Em um grafo, um caminho é uma sequência de arestas e vértices que partem de um nó A e levam até outro nó B.

- O comprimento de um caminho é o número de arestas que ele contém.



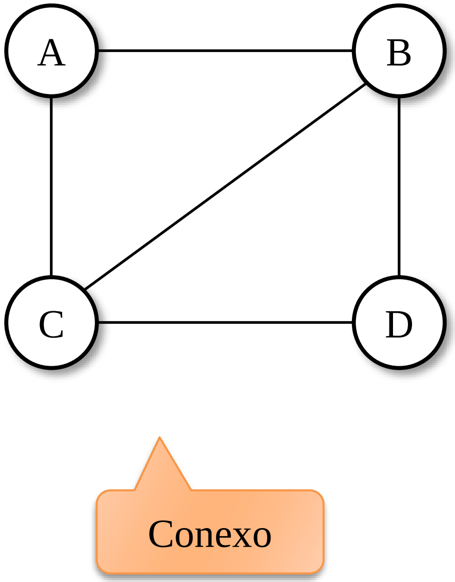
Terminologias

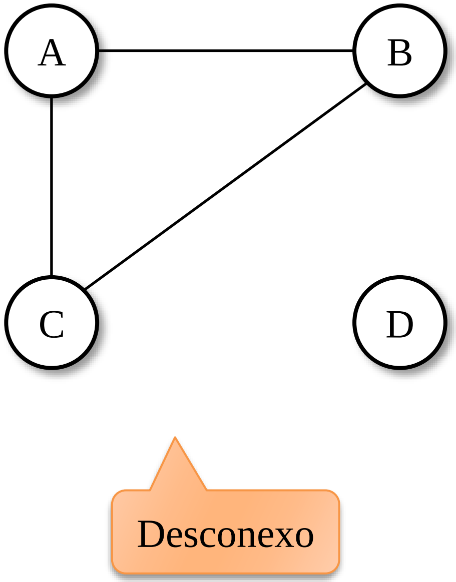
- Um ciclo é um caminho de comprimento maior ou igual a 3, em que o primeiro e o último vértice são o mesmo.



Terminologias

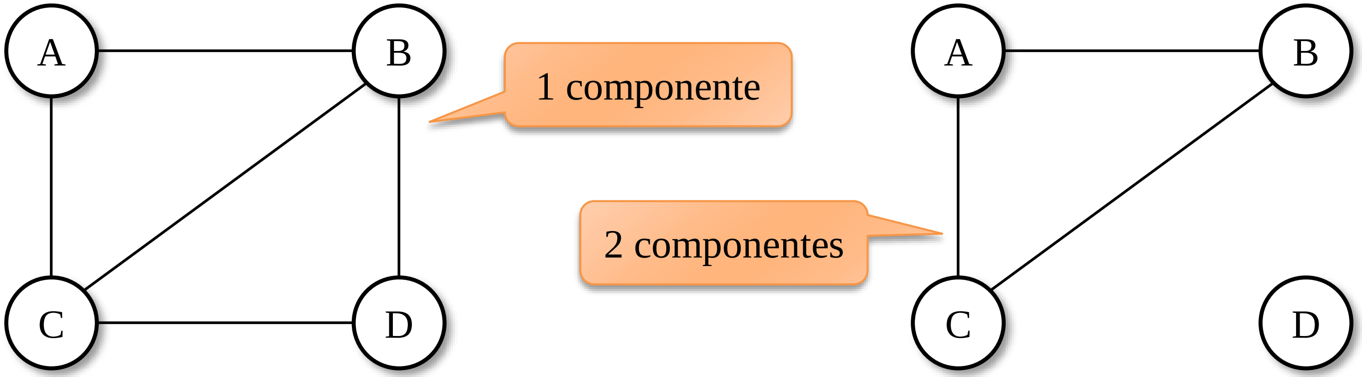
- Um grafo é conexo se existe um caminho entre quaisquer dois nós.





Terminologias

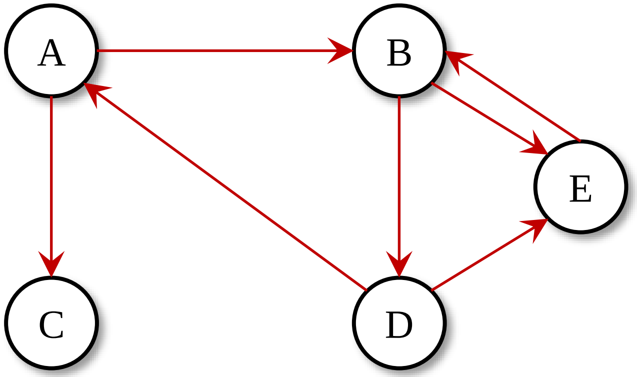
- Um grafo é conexo se existe um caminho entre quaisquer dois nós.



- As partes conexas de um grafo são chamadas de componentes.

Terminologias

- Em um grafo direcionado (também chamado de dígrafo), as arestas são representadas por setas e podem ser percorridas apenas em uma direção (a menos que exista uma aresta de volta).

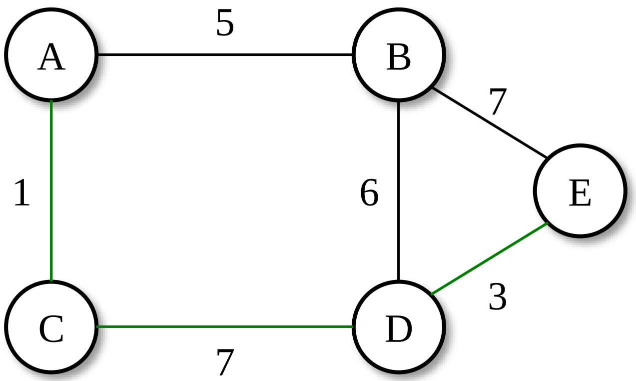


Terminologias

- Em um grafo ponderado, a cada aresta é atribuído um peso.

- Pesos são interpretados como os comprimentos das arestas, e o comprimento de um caminho é a soma de seus pesos.

- Na figura, o comprimento total do caminho A → C → D → E é 1 + 7 + 3 = 11. Este é o caminho mínimo entre os nós A e E.



Representação de grafos

- Existem diversas formas de se representar grafos de maneira computacional.

- A escolha da estrutura de dados depende do tamanho do grafo e do modo como ele será processado.

- As três formas mais comuns de representação de grafos são:
  - Lista de adjacência
  - Matriz de adjacência
  - Lista de arestas

Lista de adjacência

- Em uma lista de adjacência, a cada nó X do grafo é atribuída uma lista de nós para os quais existe uma aresta a partir de X.

Lista de adjacência

- Em uma lista de adjacência, a cada nó X do grafo é atribuída uma lista de nós para os quais existe uma aresta a partir de X.

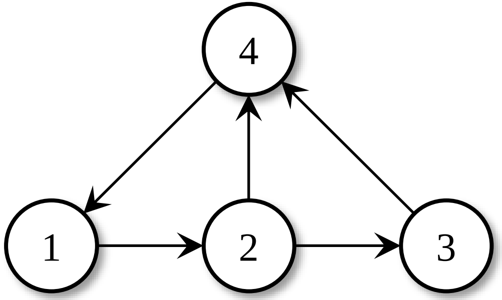
- Uma forma de armazenar a lista de adjacências é declarar um array de vectors (STL): `vector<int> adj[N];`

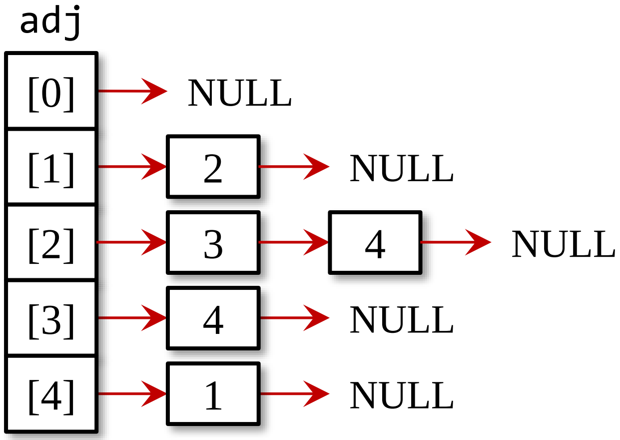
Lista de adjacência

- Em uma lista de adjacência, a cada nó X do grafo é atribuída uma lista de nós para os quais existe uma aresta a partir de X.

- Uma forma de armazenar a lista de adjacências é declarar um array de vectors (STL): `vector<int> adj[N];`

- Por exemplo:



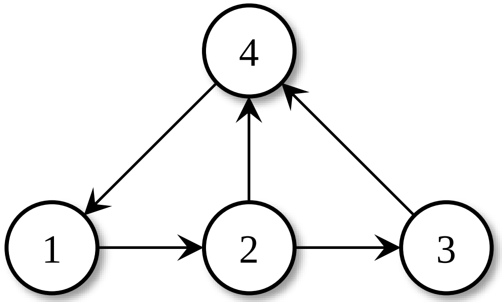


Lista de adjacência

- Em uma lista de adjacência, a cada nó X do grafo é atribuída uma lista de nós para os quais existe uma aresta a partir de X.

- Uma forma de armazenar a lista de adjacências é declarar um array de vectors (STL): `vector<int> adj[N];`

- Por exemplo:



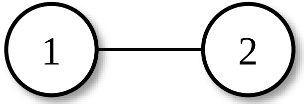

```
vector<int> adj[5];

adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

Lista de adjacência

- Em uma lista de adjacência, a cada nó X do grafo é atribuída uma lista de nós para os quais existe uma aresta a partir de X.

- Uma forma de armazenar a lista de adjacências é declarar um array de vectors (STL): `vector<int> adj[N];`

- Se o grafo é não-direcionado, as arestas são adicionadas em ambas as direções:




```
adj[1].push_back(2);
adj[2].push_back(1);
```

Lista de adjacência

- Se o grafo é ponderado, a representação da lista de adjacência pode ser estendida para: `vector<pair<int, int>> adj[N];`

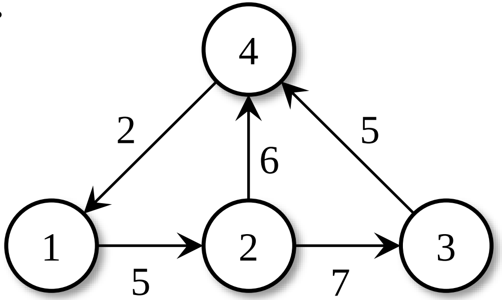
- Onde a lista de adjacência do nó A contém o par (B, *w*) quando existe uma aresta do nó A para o nó B, com peso *w*.

Lista de adjacência

- Se o grafo é ponderado, a representação da lista de adjacência pode ser estendida para: `vector<pair<int, int>> adj[N];`

- Onde a lista de adjacência do nó A contém o par (B, *w*) quando existe uma aresta do nó A para o nó B, com peso *w*.

- Por exemplo:




```
vector<pair<int, int>> adj[5];

adj[1].push_back( {2, 5} );
adj[2].push_back( {3, 7} );
adj[2].push_back( {4, 6} );
adj[3].push_back( {4, 5} );
adj[4].push_back( {1, 2} );
```

Lista de adjacência

- Através de uma lista de adjacência, pode-se, de forma eficiente, encontrar os nós alcançáveis a partir de um dado nó.

- Por exemplo, o loop a seguir permite obter todos os nós para os quais pode-se mover a partir do nó `s`:

```
for(auto u : adj[s])
{
        // processa o nó u
}
```

Matriz de adjacência

- A matriz de adjacência contém a informação a respeito das arestas do grafo.
  - Representação: `int adj[N][N];`

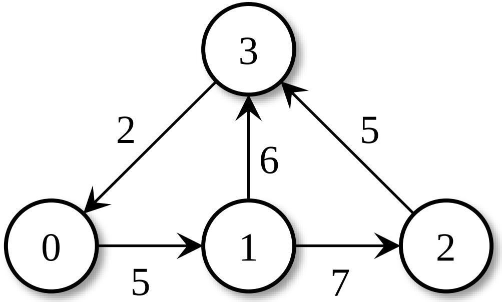
- Cada posição `adj[u][v]` da matriz indica se o grafo contém uma aresta que parte do nó `u` e chega ao nó `v`.
  - Se esta aresta existe no grafo, `adj[u][v] = 1`.
  - Caso contrário, `adj[u][v] = 0`.

- Para todo grafo não-direcionado, a matriz de adjacência será simétrica: somente os elementos da diagonal principal e baixo dela devem ser armazenados.

- Em grafos ponderados, se existe uma aresta entre os vértices `u` e `v`, `adj[u][v]` contém o peso desta aresta.

Matriz de adjacência

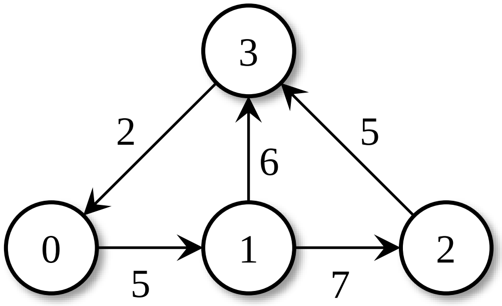
- Por exemplo:



	[0]	[1]	[2]	[3]
[0]		5		
[1]			7	6
[2]				5
[3]	2			

Matriz de adjacência

- Por exemplo:

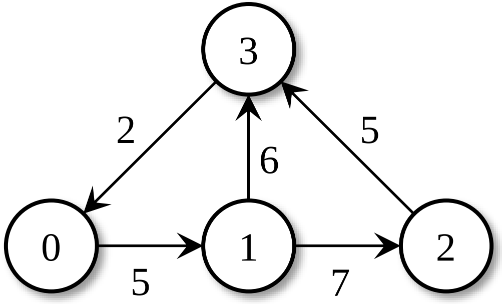


	[0]	[1]	[2]	[3]
[0]	0	5		
[1]		0	7	6
[2]			0	5
[3]	2			0

Distância de um vértice para si próprio é zero.

Matriz de adjacência

- Por exemplo:



	[0]	[1]	[2]	[3]
[0]	0	5	-1	-1
[1]	-1	0	7	6
[2]	-1	-1	0	5
[3]	2	-1	-1	0

Se não há aresta entre dois nós,
sua distância é infinita (-1).

Lista de arestas

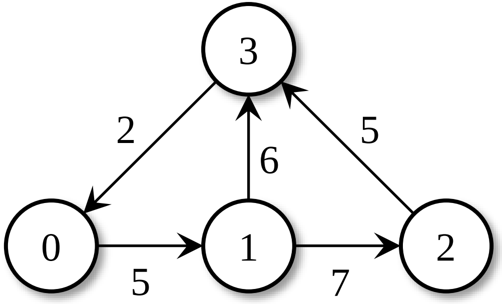
- Uma lista de arestas contém todas as arestas do grafo, dispostas em alguma ordem aleatória.

- Esta forma de representação é conveniente se o algoritmo que processa o grafo manipula todas as arestas, sem a necessidade de encontrar as arestas que partem de um dado vértice.

- De forma similar à lista de adjacência, uma lista de arestas pode ser representada através da estrutura vector:
  - Grafos não ponderados: o par (u, v) denota uma aresta do nó u até v.
  - Grafos ponderados: a tupla (u, v, *w*) denota uma aresta do nó u até v, com peso *w*.

Lista de arestas

- Por exemplo:




```
vector<tuple<int, int, int>> edg;

edg.push_back( {0, 1, 5} );
edg.push_back( {1, 2, 7} );
edg.push_back( {1, 3, 6} );
edg.push_back( {2, 3, 5} );
edg.push_back( {3, 0, 2} );
```

- C++ tuple: *https://en.cppreference.com/w/cpp/utility/tuple*

Qual a melhor forma de representação?

- Depende da relação entre as quantidades de vértices e arestas do grafo.

- O número de arestas (|A|) pode ser tão pequeno quanto o número de vértices (ou menor); ou tão grande quanto $|V|^2$.
  - Quando |A| estiver perto do limite superior desse intervalo, o grafo é denso. Neste caso, a melhor opção para sua representação é a matriz de adjacência.
  - Quando |A| for próximo a |V|, o grafo é esparso. Neste caso, a melhor opção é usar a lista de adjacência.
  - Quando o algoritmo que processa o grafo não se preocupa com os vértices (ex. algoritmo de Kruskal – *próximas aulas*), a melhor representação é a lista de arestas.

Percurso em grafos

- Existem várias técnicas para se percorrer um grafo.
- As técnicas de percurso fundamentais são a Busca em Profundidade e a Busca em Largura.
- Ambos os algoritmos iniciam a busca a partir de um nó inicial e visitam todos os nós do grafo que podem ser alcançados a partir deste nó inicial.
- A diferença entre os dois tipos de busca está na ordem em que os nós do grafo são visitados.

Busca em Profundidade

- A Busca em Profundidade (DFS – Depth-First Search) inicia a partir de um dado nó do grafo e o atravessa sempre se aprofundando através de suas arestas.

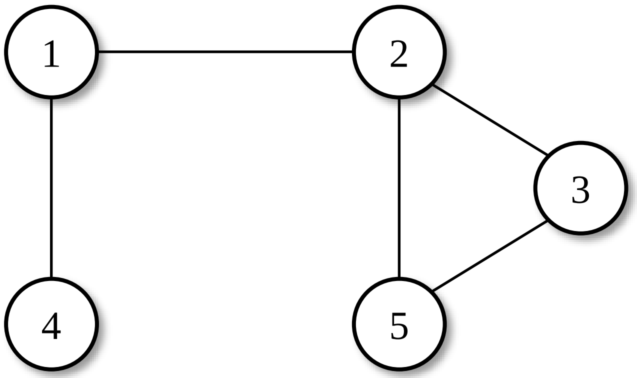
- A cada nó que o algoritmo analisa, se este nó possuir mais de um vizinho, a busca escolhe uma destas arestas e visitará esse nó vizinho.

- O algoritmo repete esse processo até que um nó em que não é possível continuar caminhando seja encontrado.

- Quando isso acontece, o algoritmo retrocede a um vizinho anterior e continua a busca a partir dele.

Busca em Profundidade

- DFS a partir do vértice 1:

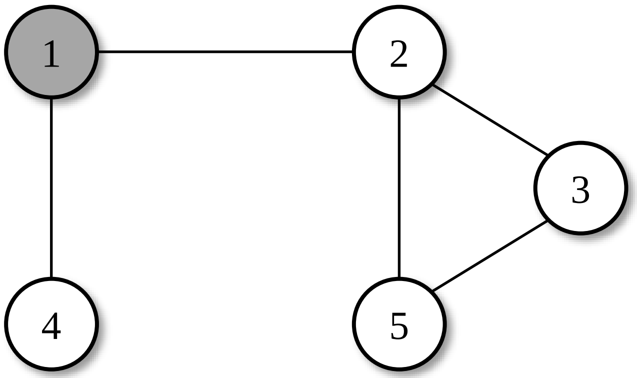


Busca em Profundidade

- DFS a partir do vértice 1:

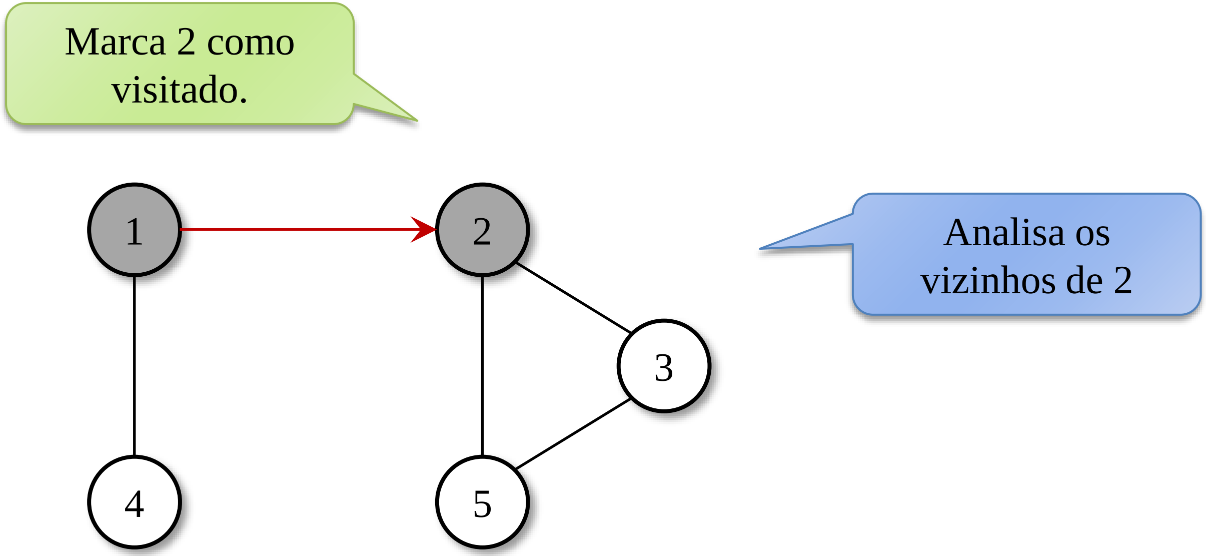
Analisa os vizinhos de 1

Marca 1 como visitado.



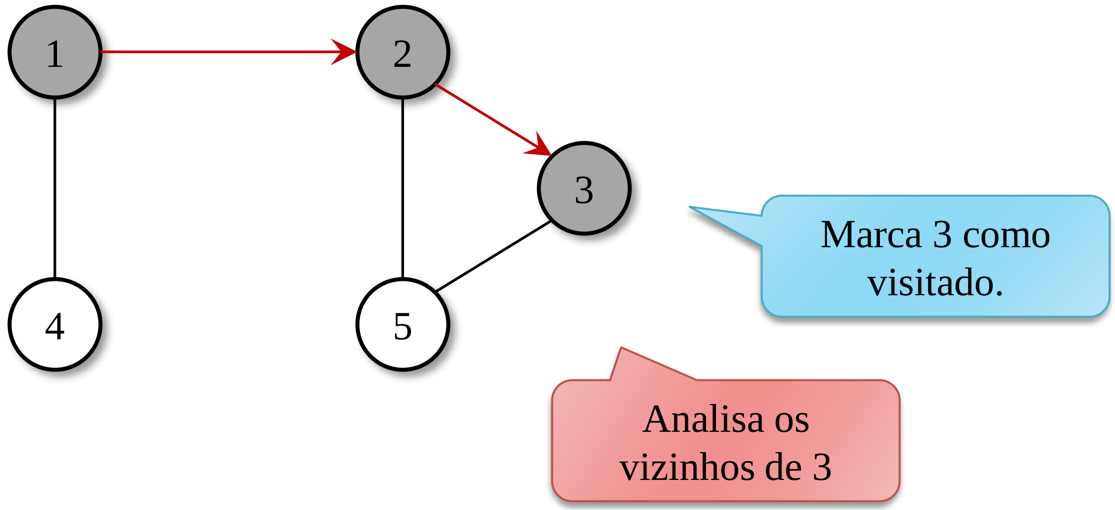
Busca em Profundidade

- DFS a partir do vértice 1:



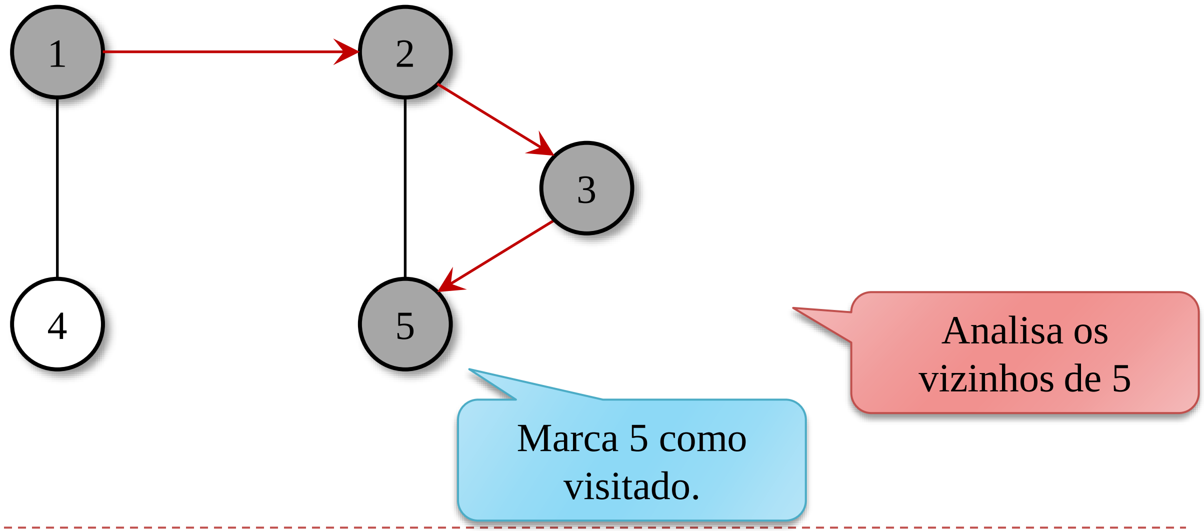
Busca em Profundidade

- DFS a partir do vértice 1:



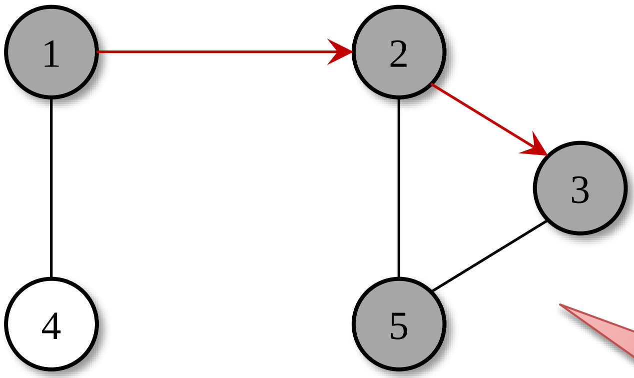
Busca em Profundidade

- DFS a partir do vértice 1:



Busca em Profundidade

- DFS a partir do vértice 1:

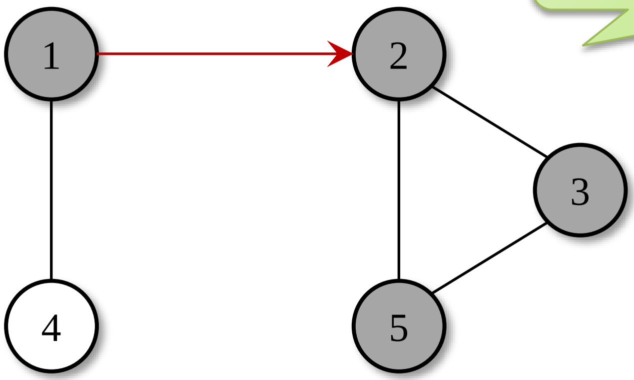


Como os vizinhos de 5 já foram visitados, retrocede ao 3.

Busca em Profundidade

- DFS a partir do vértice 1:

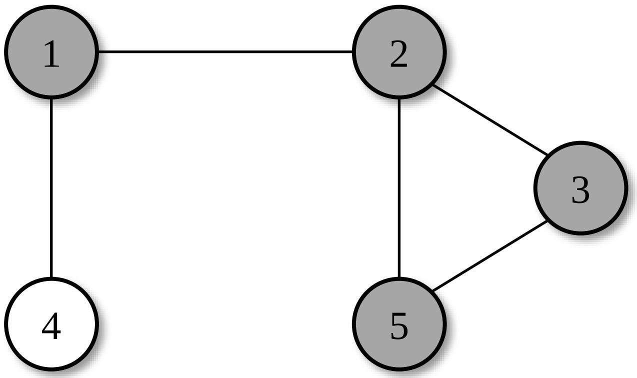
Como os vizinhos de 3 já foram visitados, retrocede ao 2.



Busca em Profundidade

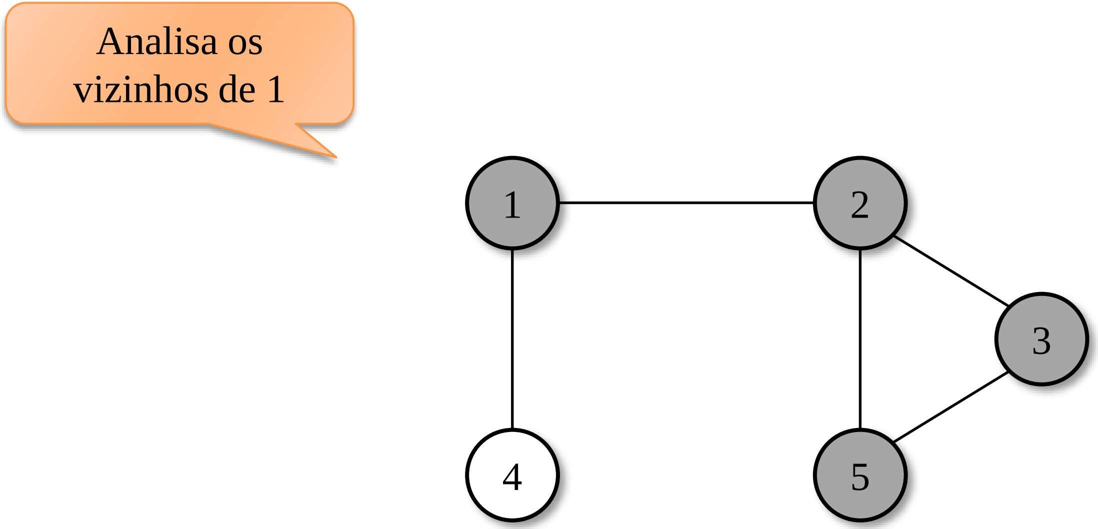
- DFS a partir do vértice 1:

Como os vizinhos de 2 já foram visitados, retrocede ao 1.



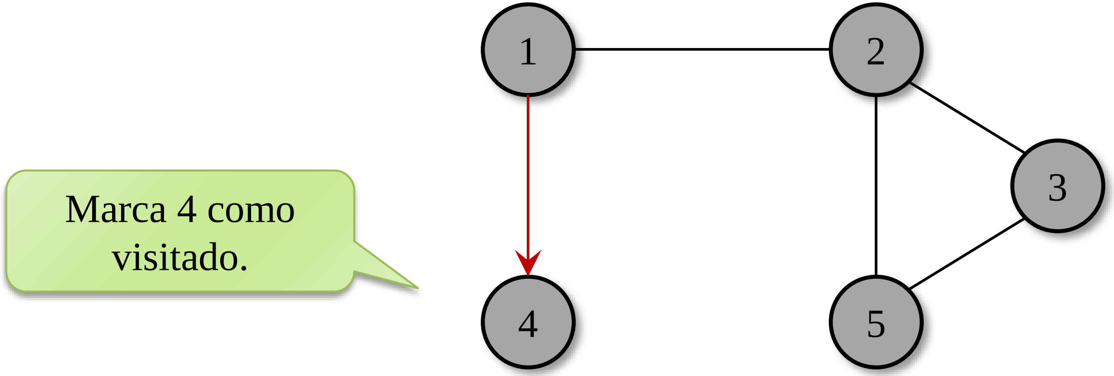
Busca em Profundidade

- DFS a partir do vértice 1:



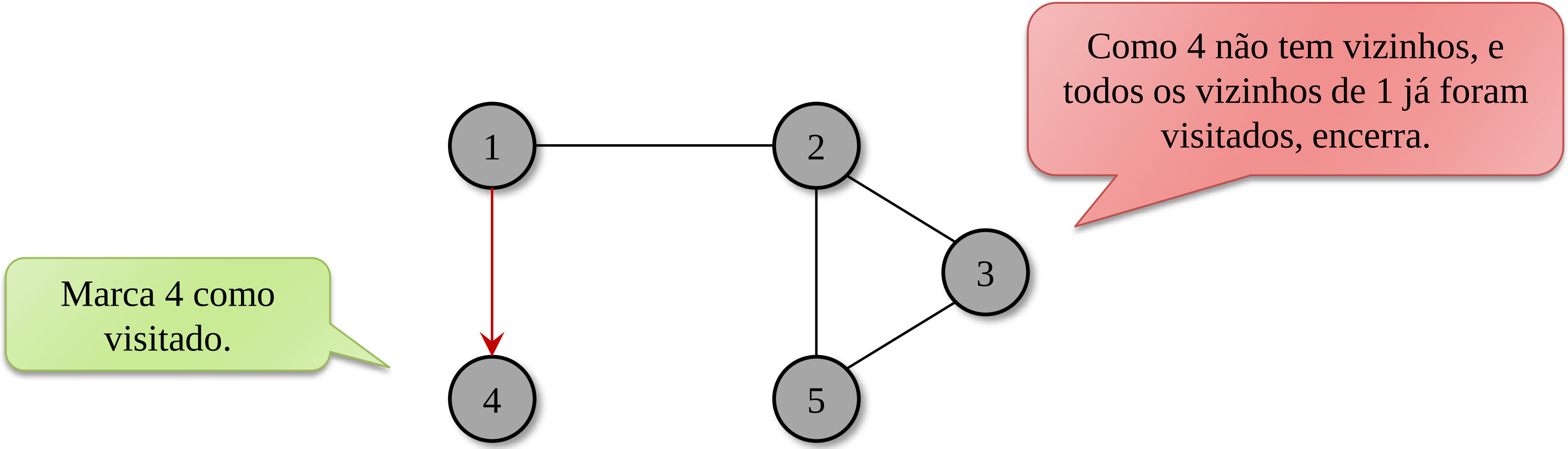
Busca em Profundidade

- DFS a partir do vértice 1:



Busca em Profundidade

- DFS a partir do vértice 1:



Busca em Profundidade

- Implementação recursiva com matriz de adjacência:

```
// Definicoes gerais
int N = 10; // qtd de vertices
int m[N][N]; // a matriz de adjacencia
int visitado[N]; // marca os nos visitados. Inicializar com 0.
int anterior[N]; // marca o no anterior na busca. Inicializar com -1.

```

Busca em Profundidade

- Implementação recursiva com matriz de adjacência:

```c
// u: no inicial.
void dfs_rec(int u)
{
    int i;

    if(!visitado[u])
    {
        visitado[u] = 1;
        printf("%d ", u);
    }
    for(i = 0; i < N; i++)
    {
        if(m[u][i] && !visitado[i])
        {
            anterior[i] = u;
            dfs_rec(i);
        }
    }
}
```

Busca em Profundidade

- Implementação não-recursiva com matriz de adjacência:

```
// Definicoes gerais
int N = 10; // qtd de vertices
int m[N][N]; // a matriz de adjacencia
int visitado[N]; // marca os nos visitados. Inicializar com 0.
int anterior[N]; // marca o no anterior na busca. Inicializar com -1.
stack<int> pilha; // pilha para marcar o caminho de volta

```

Busca em Profundidade

- Implementação não-recursiva com matriz de adjacência:

```
// s: no inicial.
void dfs(int s)
{
    int u, i;

    pilha.push(s);

    while(!pilha.empty())
    {
        u = pilha.top();
        pilha.pop();
        if(!visitado[u])
        {
            visitado[u] = 1;
            printf("%d ", u);
        }

        for(i = 0; i < N; i++)
        {
            if(m[u][i] && !visitado[i])
            {
                pilha.push(i);
                anterior[i] = u;
            }
        }
    }
}
```

Busca em Profundidade

- Implementação recursiva com lista de adjacência:

```
vector<int> adj[N]; // lista de adjacencia
int visitado[N]; // marca os nos visitados. Inicializar com 0.

// u: no inicial
void dfs_list(int u)
{
    if(!visitado[u])
    {
        visitado[u] = 1;
        printf(“%d ”, u);

        for(auto i: adj[u])
            dfs_list(i);
    }
}
```

Busca em Largura

- A Busca em Largura (BFS – Breadth-First Search) visita os nós de um grafo em ordem crescente de níveis para o nó inicial da busca.

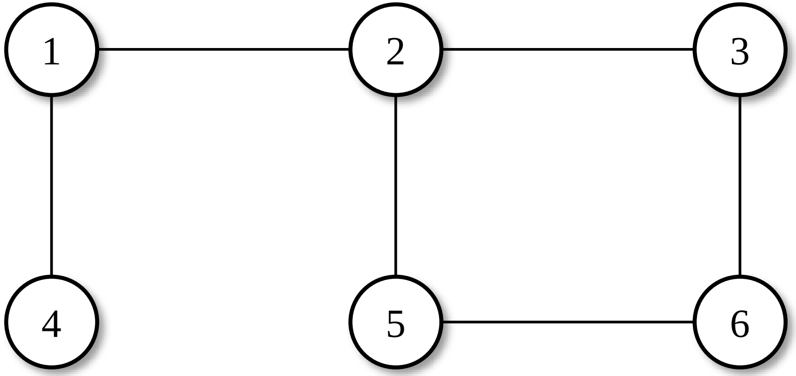
- Dessa forma, é possível calcular a distância de um nó inicial para todos os outros nós do grafo usando BFS.

- Entretanto, sua implementação inclui detalhes adicionais em relação à DFS.

- A BFS analisa, primeiro, os nós que são vizinhos diretos do nó inicial. A seguir, parte para os vizinhos dos vizinhos, e assim sucessivamente, até que todos os nós tenham sido visitados.

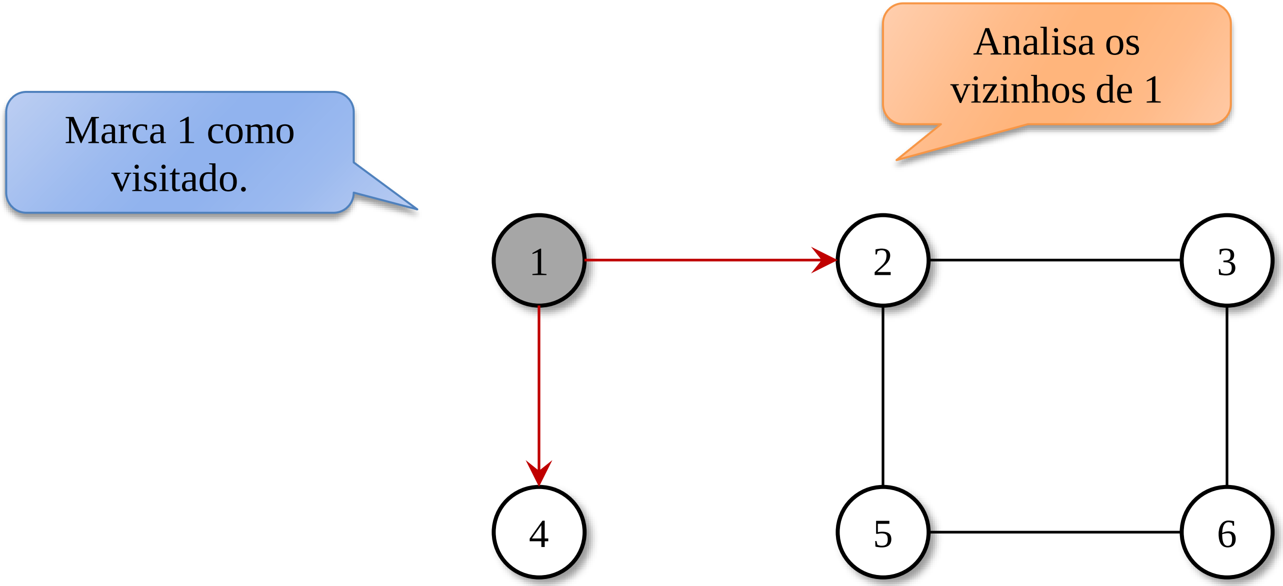
Busca em Largura

- BFS a partir do vértice 1:



Busca em Largura

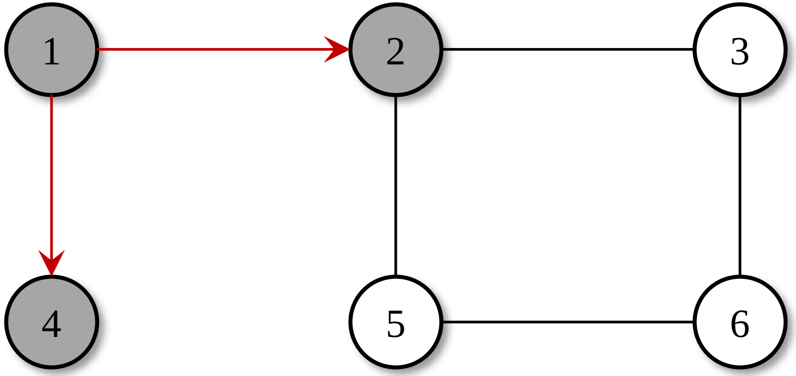
- BFS a partir do vértice 1:



Busca em Largura

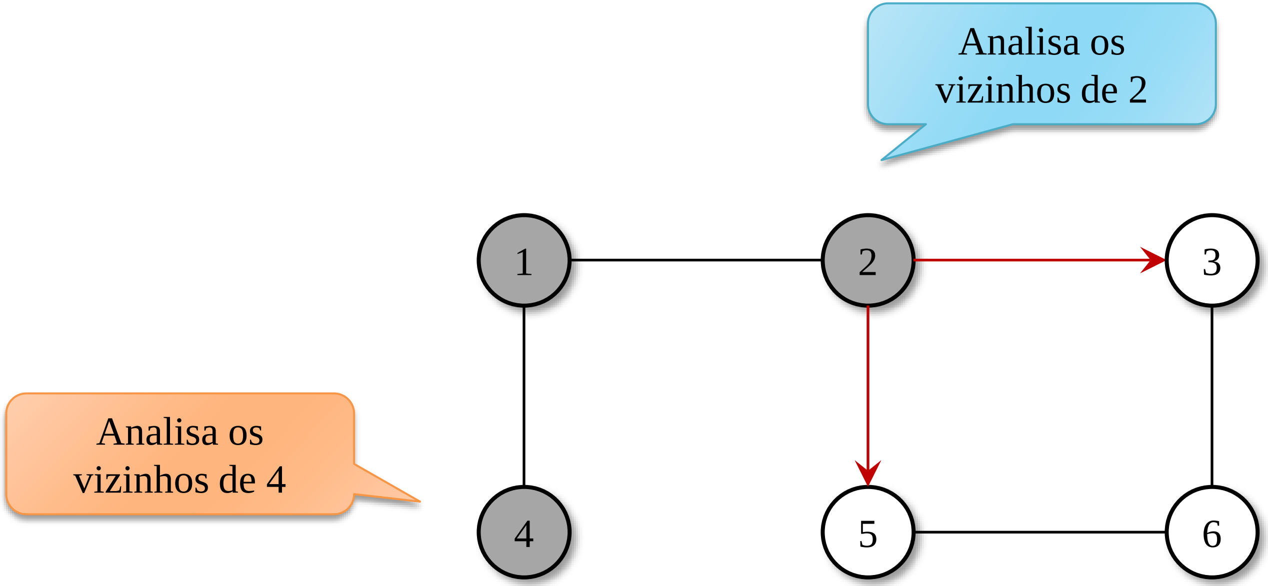
- BFS a partir do vértice 1:

Marca 2 e 4 como visitados.



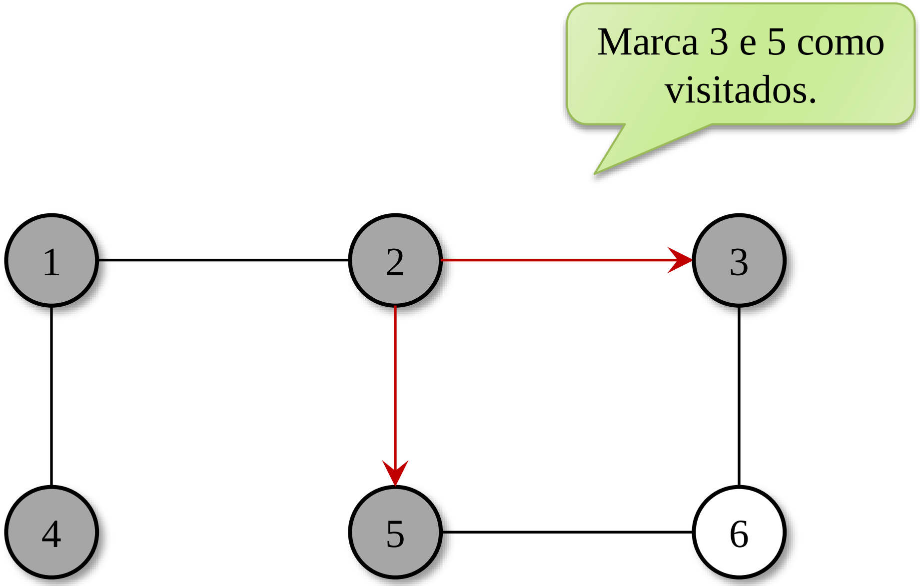
Busca em Largura

- BFS a partir do vértice 1:



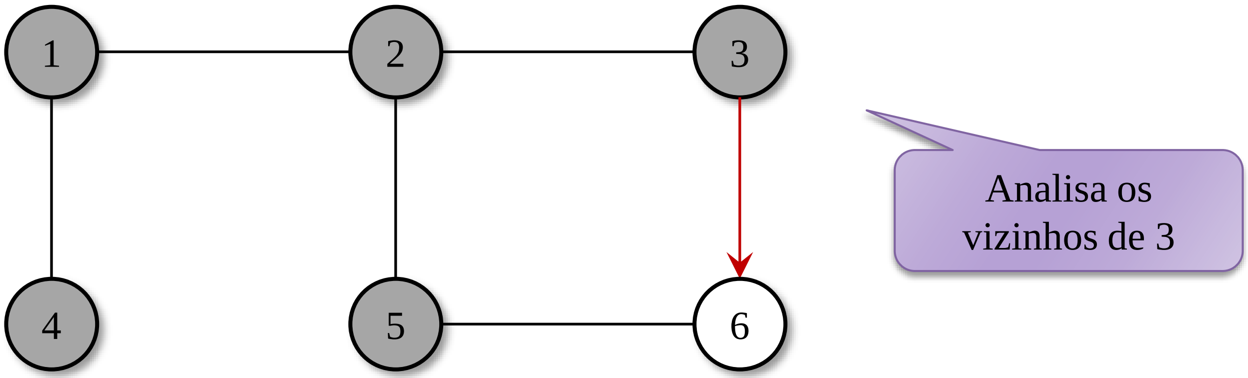
Busca em Largura

- BFS a partir do vértice 1:



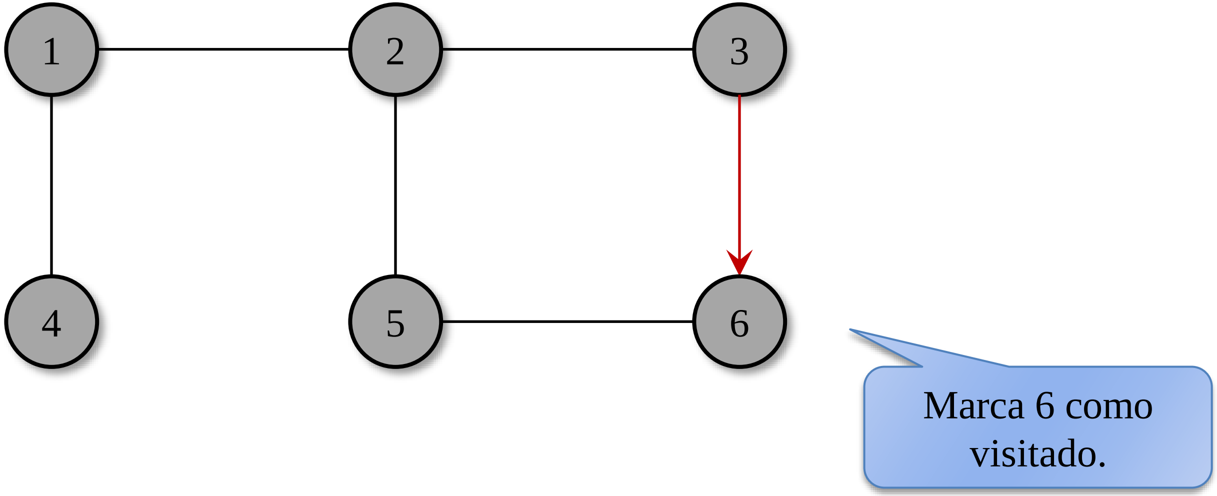
Busca em Largura

- BFS a partir do vértice 1:



Busca em Largura

- BFS a partir do vértice 1:



Busca em Largura

- A Busca em Largura é mais complexa de ser implementada.
- A razão para isso é que o algoritmo visita nós em diferentes partes do grafo.
- Uma implementação típica faz uso da estrutura de fila para armazenar os nós de cada nível do grafo.
- A cada passo, os nós vizinhos ao nó analisado são enfileirados para que possam ser processados.
- Na implementação a seguir, o vetor `dist[]`, além de manter a distância, também é usado para identificar se um nó já foi visitado.

Busca em Largura

- Implementação com matriz de adjacência:

```
// Definicoes gerais
int INF -1;
int N = 10; // qtd de vertices
int m[N][N]; // a matriz de adjacencia
int dist[N]; // Distancias a partir do inicio. Inicializar com infinito.
queue<int> fila; // estrutura de fila para percorrer os niveis.

```

Busca em Largura

- Implementação com matriz de adjacência:

```
void bfs(int s)
{
    int i, u;

    dist[s] = 0;
    fila.push(s);

    while(!fila.empty())
    {
        u = fila.front();
        fila.pop();
        printf("%d ", u);

        for(i = 0; i < N; i++)
        {
            if(dist[i] == INF && m[u][i])
            {
                dist[i] = dist[u] + 1;
                fila.push(i);
            }
        }
    }
}
```

Aplicações

- Ambos os algoritmos de busca apresentados permitem verificar diversas propriedades de um grafo.

- Conectividade: é possível verificar se um grafo é conexo se, a partir de um nó inicial, é possível alcançar todos os demais nós.

- Detecção de ciclos: um grafo contém um ciclo se, durante a sua travessia, encontramos um nó cujo vizinho tenha sido visitado.

- Grafo bipartido: um grafo é bipartido se for possível colorir seus nós usando duas cores, de forma que dois nós adjacentes não tenham a mesma cor.

Aplicações

- Grafo bipartido (cont.): detectar se um grafo é, ou não, bipartido, é simples.

- Considerando duas cores, X e Y, colore-se o nó inicial da busca com X, seus vizinhos com Y, os vizinhos de seus vizinhos com X, e assim por diante.

- Se, em algum ponto da busca, detectarmos que dois nós adjacentes estão com a mesma cor, o grafo não é bipartido.

- Visualização das Buscas em Profundidade e Largura:
  - https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

Dúvidas?

Aula 7:
Introdução a Grafos

Disciplina: Maratona de Programação 1

Profs. Edmilson Marmo e Luiz Olmes
edmarmo@unifei.edu.br, olmes@unifei.edu.br

UNIFEI
UNIVERSIDADE FEDERAL DE ITAJUBÁ