Introdução à Teoria dos Grafos

Gustavo Henrique Aragão Silva

UNIVERSIDADE FEDERAL DE SERGIPE

Greedy <algorithm>

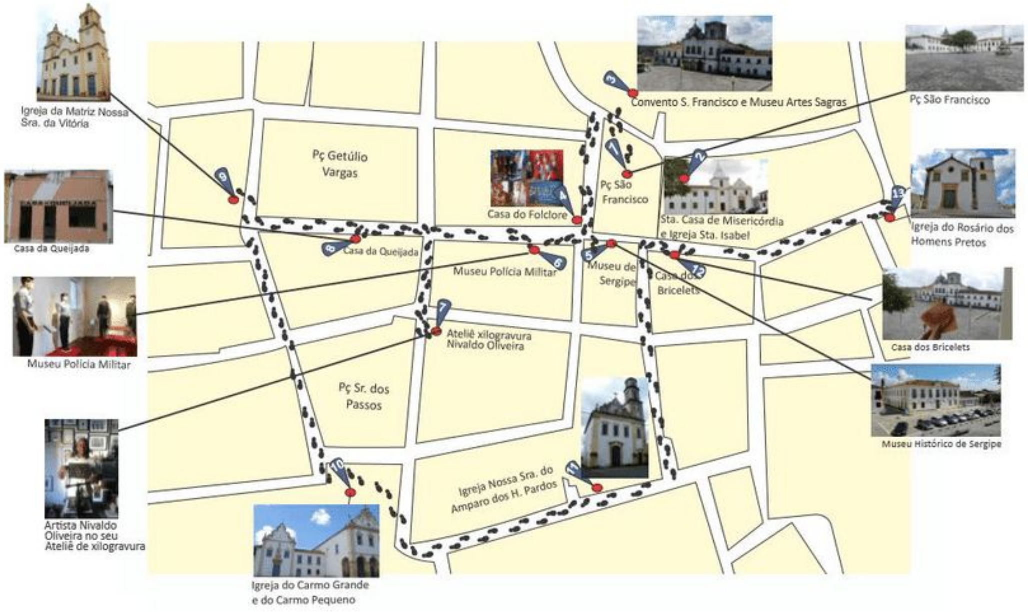
Departamento de Computação/UFS

Agenda

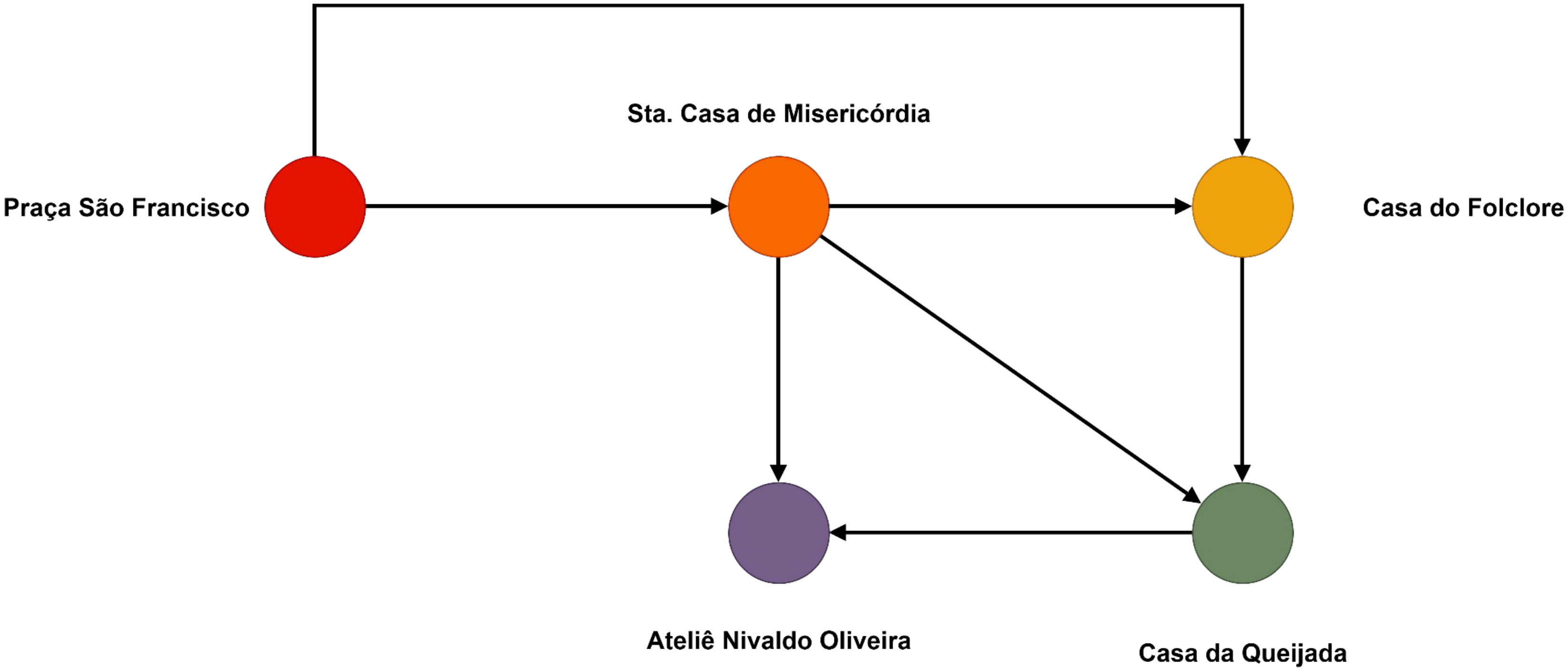
- **Motivação**
- **Definição de Grafo**
- **Caminho e Ciclo**
- **Conectividade**
- **Componentes Conexos**
- **Representação de Grafos:** Lista de Adjacências
- **Busca em Profundidade (DFS)**
- **Busca em Largura (BFS)**

Motivação

- Para resolver problemas, é conveniente abstrair e extrair apenas as partes necessárias
- Com frequência é possível pensar em um problema como elementos e suas relações



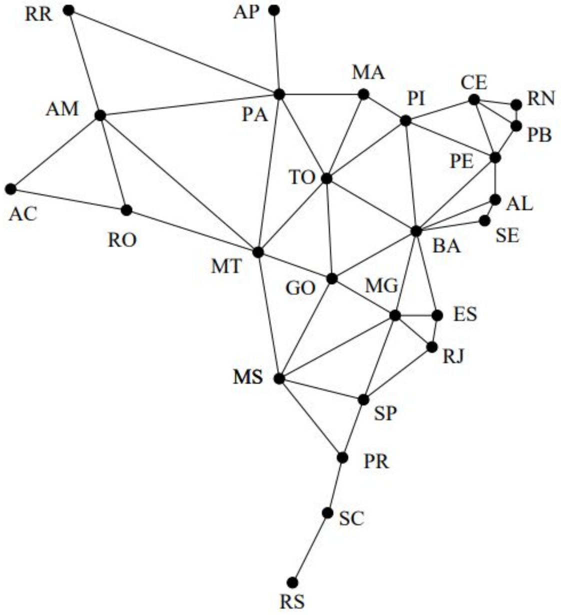
Motivação



Sta. Casa de Misericórdia
Praça São Francisco
Casa do Folclore
Ateliê Nivaldo Oliveira
Casa da Queijada

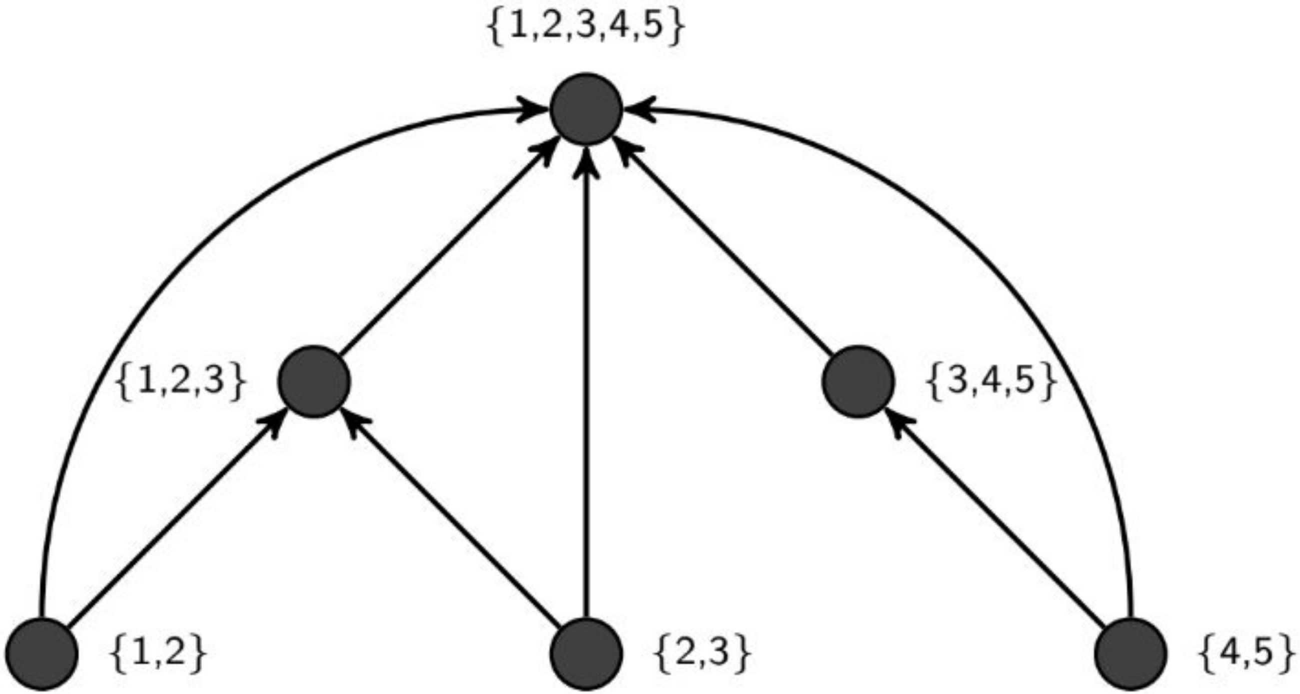
Motivação

Um grafo que representa as fronteiras dos estados do Brasil.



Motivação

Dados dois conjuntos *v* e *w* de números inteiros, **v** → **w** se **v** ⊆ **w**.

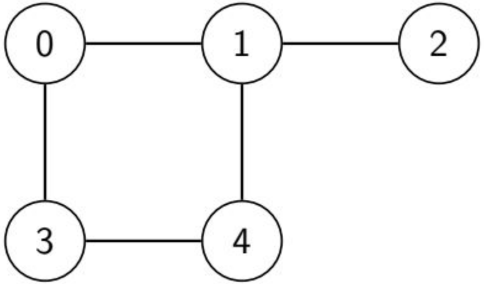


Definição de Grafo

Um **grafo** *G* é um par de conjuntos $G = (V, E)$, sendo **V** um conjunto de **vértices** (pontos) e *E* um conjunto de **arestas** (linhas).

As arestas serão representadas como um par (v, w)

- **ordenado**, quando G é **direcionado**.
- **não-ordenado**, quando G é **não-direcionado**.



Uma aresta (v, w) pode ser denotada por *vw* ou por *wv*.

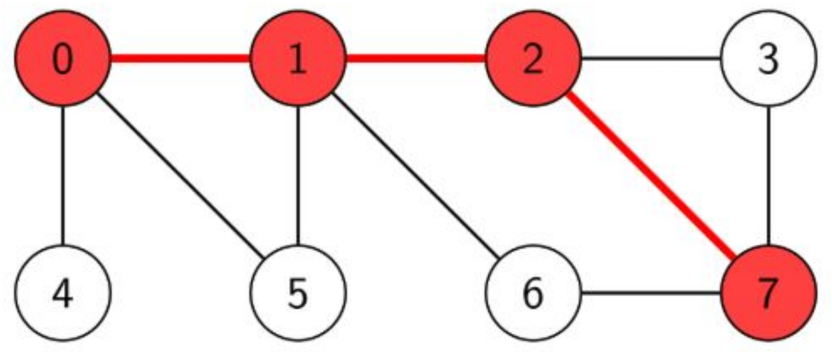
Se *vw* é uma aresta, dizemos que *v* e *w* são **vizinhos** ou **adjacentes**.

- $V = \{0, 1, 2, 3, 4\}$
- $E = \{(0, 1), (0, 3), (1, 2), (1, 4), (3, 4)\}$

Caminho

Um caminho P de x_0 para x_k é um grafo não vazio da forma $V = \{x_0, x_1, \ldots, x_k\}$ e $E = \{(x_0, x_1), (x_0, x_1), \ldots, (x_{k-1}, x_k)\}$ em que todos os x_i são distintos (exceto pelos vértices inicial e final, eventualmente).

O **comprimento** de um caminho é definido pela norma de E, ou seja, |E|.

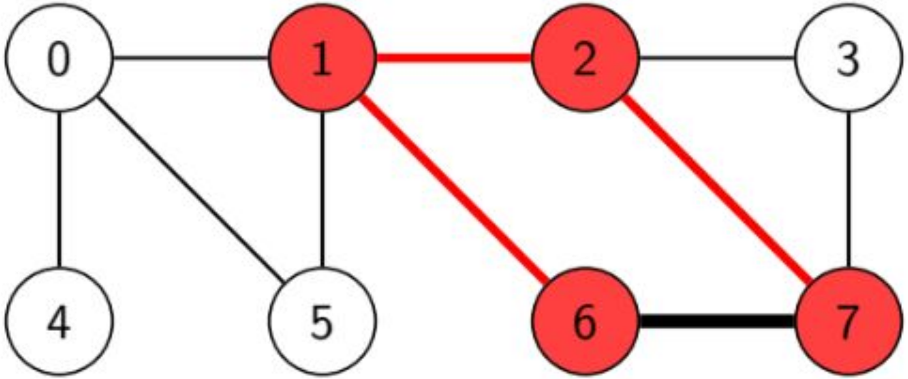


P é um caminho de 0 para 7 de comprimento 3.

- V(P) = {0, 1, 2, 7} e E(P) = {(0, 1), (1, 2), (2,7)}.

Ciclo

Sendo P um caminho de x_0 para x_k de comprimento maior ou igual a 2, um **ciclo** C é definido como $P + (x_k, x_0)$ (adiciona uma aresta, fechando o ciclo).



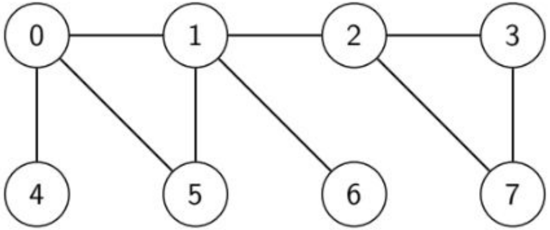
P é um caminho de 6 para 7 de comprimento 3.

- V(P) = {6, 1, 2, 7} e E(P) = {(6, 1), (1, 2), (2,7)}.
- V(C) = {6, 1, 2, 7} e E(C) = {(6, 1), (1, 2), (2,7), (7, 6)}.

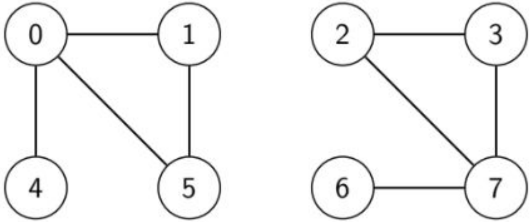
Conectividade

Um grafo G é dito **conexo** se para todo par de vértices *v*, *w* ∈ G, existe um caminho P em G de v para w e **desconexo** caso o contrário.

- Exemplo de grafo **conexo**



- Exemplo de grafo **desconexo**



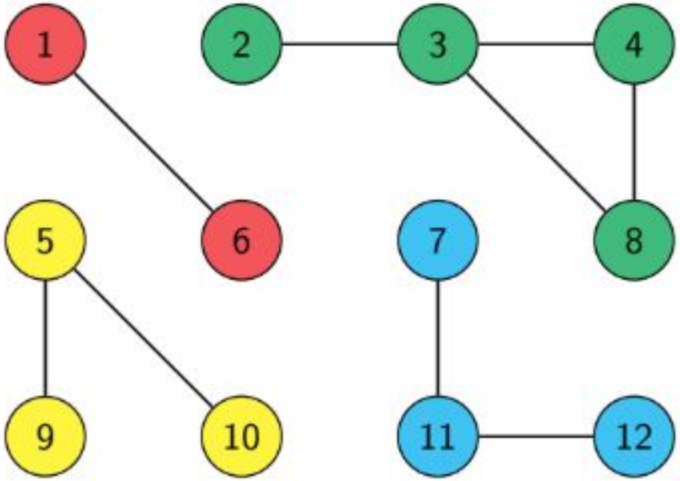
Problema Motivador: Gincana (OBI 2011)

Em uma sala com *N* alunos numerados de 1 a N, temos *M* relações de amizade. Uma relação de amizade é dada na forma de um par *(i, j)* que indica que tanto *i* é amigo de *j* quanto *j* é amigo de *i*.

A professora de Juquinha quer separar a sala de maneira a ter o máximo número de times possível.

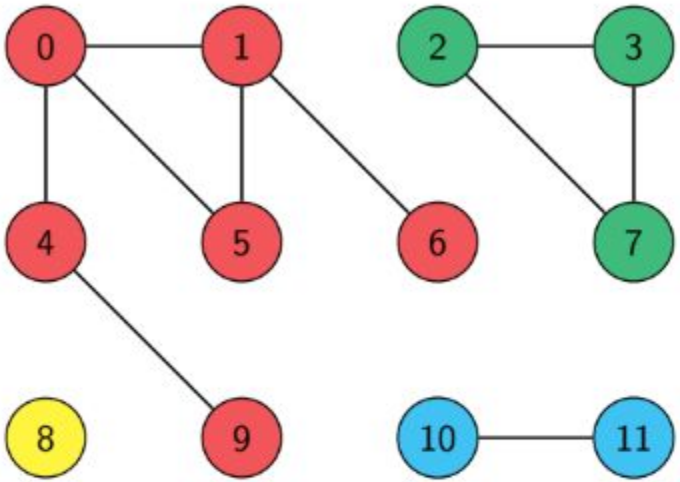
Responda **quantos times a professora pode formar**.

n	Aluno
1	Carlos
2	Fernanda
3	Ana
4	Maria
5	João
6	André
7	Antônio
8	Eduarda
9	Flávio
10	José
11	Isabela
12	Luísa



Componentes Conexos

Um **subgrafo conexo maximal** de um Grafo é chamado de **componente conexo**.



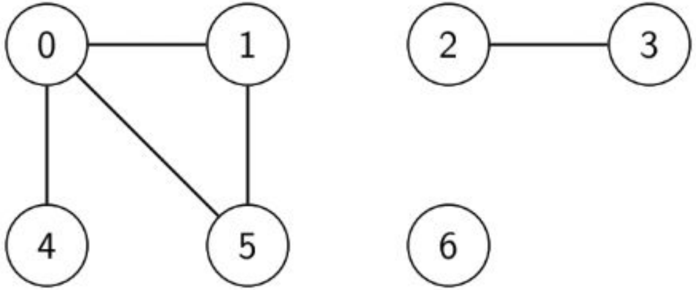
O grafo G acima tem **4 componentes conexos**, marcados por diferentes cores.

Representação de Grafos: Lista de Adjacências

É natural guardar, para cada vértice, os vértices que podemos atingir por meio de uma aresta, ou seja, os **vizinhos**.

O número de vizinhos de um vértice é chamado de **grau** do vértice.

Vértice	Vizinhos	Grau
0	{1,4,5}	3
1	{0,5}	2
2	{3}	1
3	{2}	1
4	{0}	1
5	{0,1}	2
6	{}	0

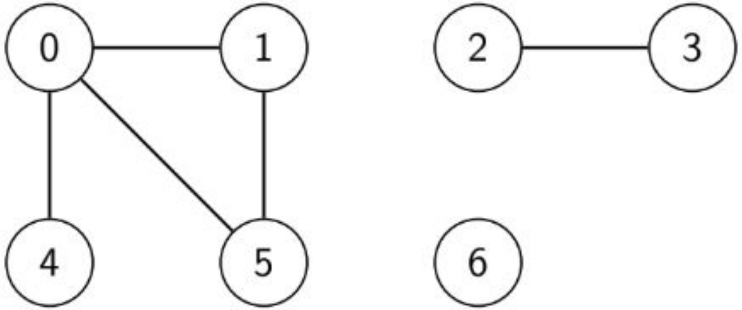


Representação de Grafos: Lista de Adjacências

Podemos usar um **vector<vector<int>>** para guardar os vizinhos de cada vértice.

Desta maneira podemos representar um grafo em $O(|V| + |E|)$ de memória.

```
v[0] = {1,4,5}
v[1] = {0,5}
v[2] = {3}
v[3] = {2}
v[4] = {0}
v[5] = {0,1}
v[6] = {}
```



Busca em Profundidade (DFS)

É útil saber **como caminhar** em um Grafo.

- Verificar conectividade
- Encontrar ciclos
- Coloração de grafos
- **Encontrar componentes conexos**
- **...**

Dessa necessidade, surge o nosso primeiro algoritmo para percorrer um grafo, o **Depth-first Search (DFS)** ou **Busca em Profundidade**.

Busca em Profundidade (DFS)

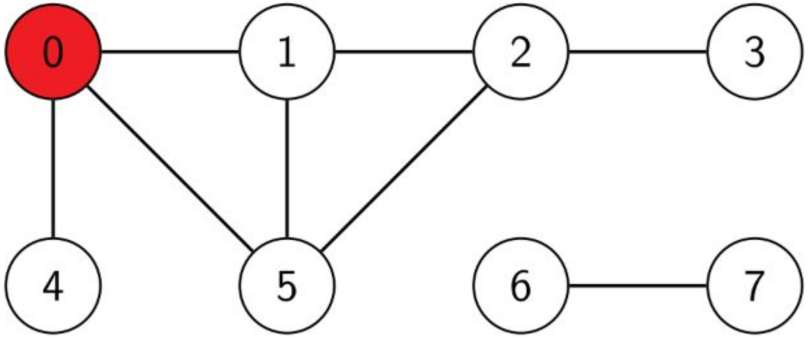
A ideia é começar a busca de *v* e, enquanto tiver vértices alcançáveis não visitados, continuar “percorrendo o grafo”, andando de vizinho em vizinho sucessivamente.

- **Complexidade:** $O(|V| + |E|)$
- **Pseudocódigo:**

```
DFS(vértice v)  {
    visitado[v] = verdadeiro
    Para todo vizinho w de v:
        Se w ainda não foi visitado:
            DFS(w)
}
```

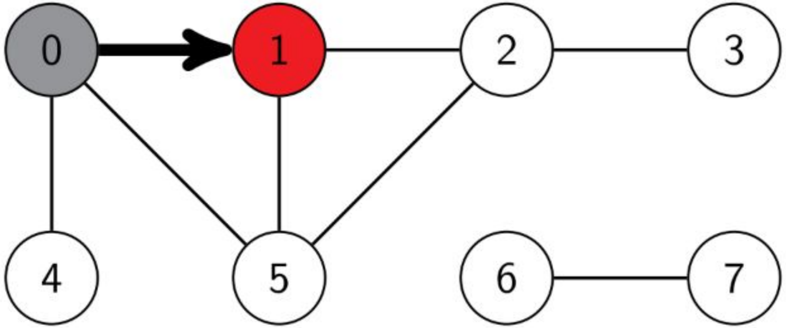
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	N
2	{1,3,5}	N
3	{2}	N
4	{0}	N
5	{0,1,2}	N
6	{7}	N
7	{6}	N



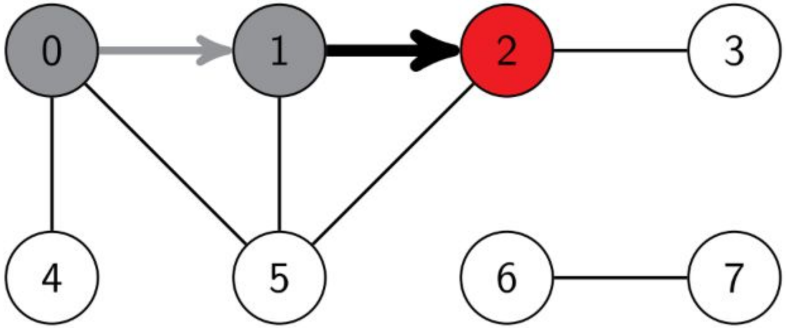
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	N
3	{2}	N
4	{0}	N
5	{0,1,2}	N
6	{7}	N
7	{6}	N



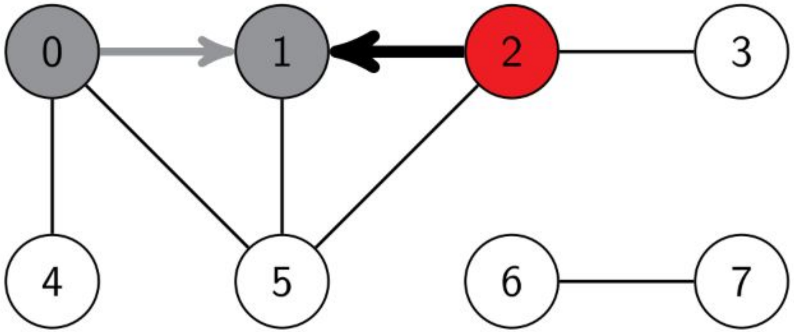
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	N
4	{0}	N
5	{0,1,2}	N
6	{7}	N
7	{6}	N



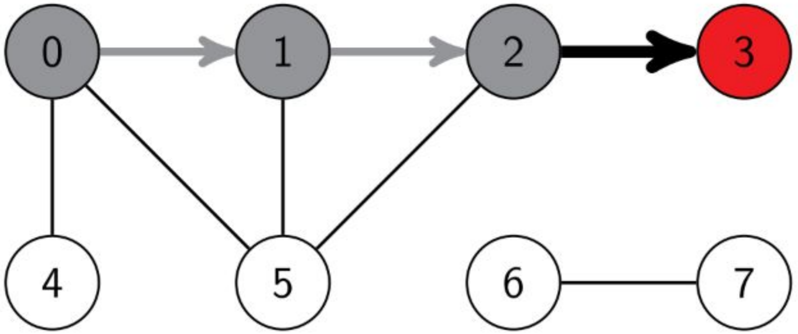
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	N
4	{0}	N
5	{0,1,2}	N
6	{7}	N
7	{6}	N



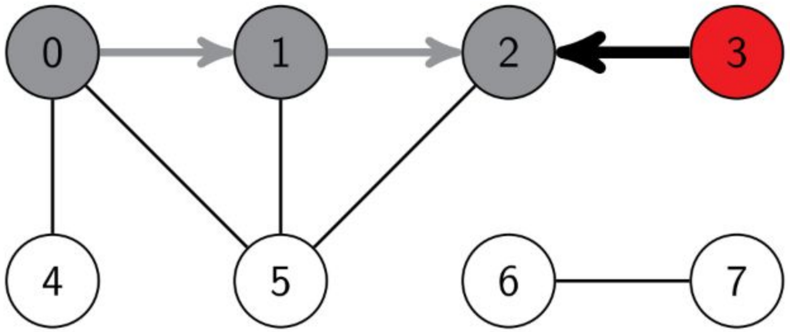
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	N
6	{7}	N
7	{6}	N



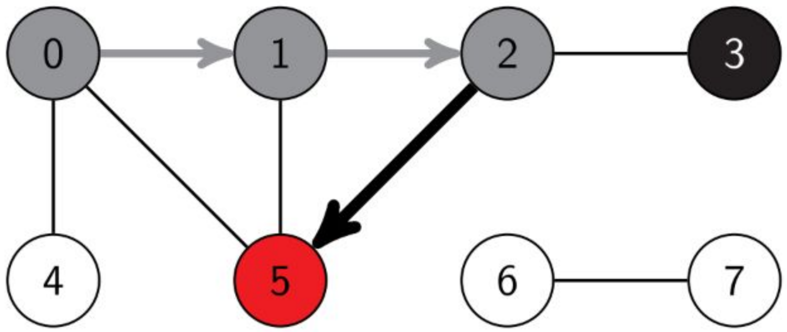
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	N
6	{7}	N
7	{6}	N



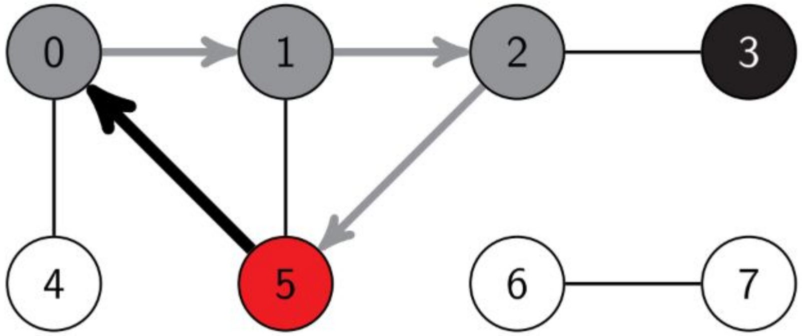
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	S
6	{7}	N
7	{6}	N



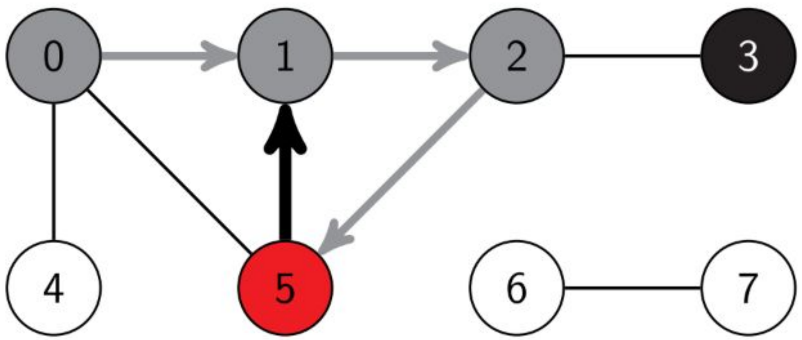
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	S
6	{7}	N
7	{6}	N



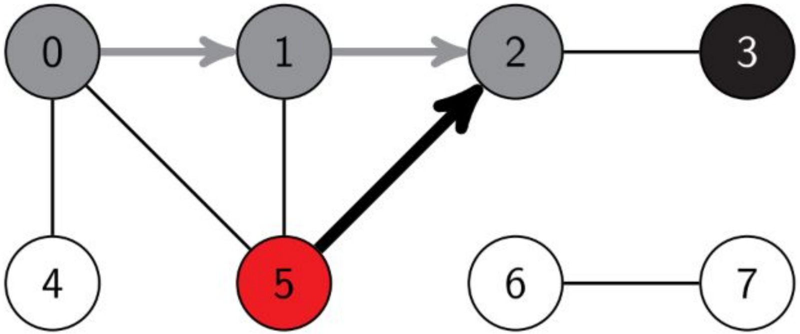
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	S
6	{7}	N
7	{6}	N



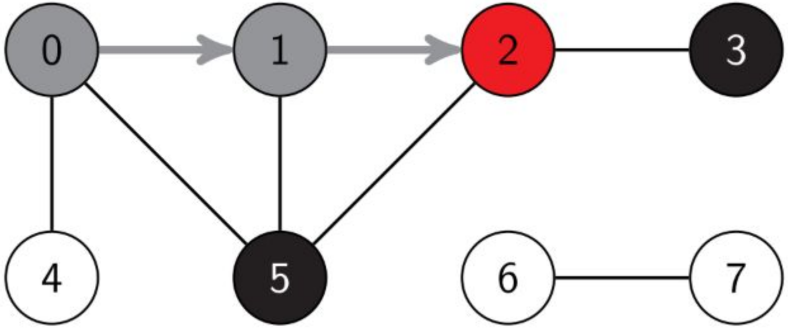
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	S
6	{7}	N
7	{6}	N



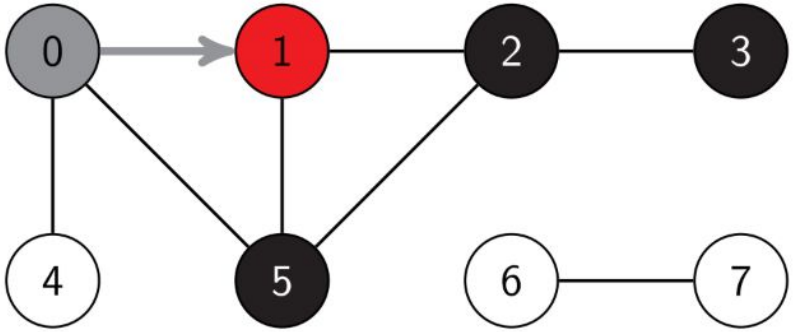
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	S
6	{7}	N
7	{6}	N



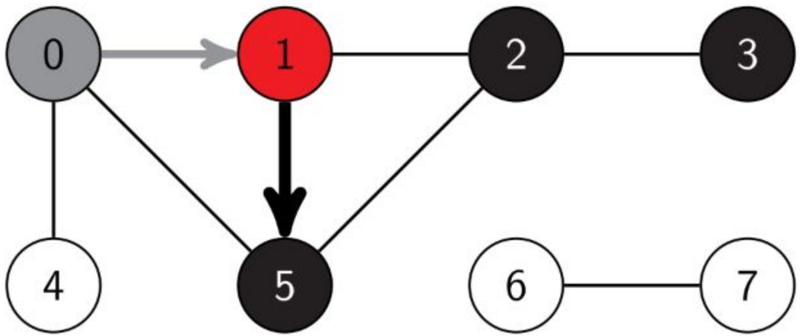
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	S
6	{7}	N
7	{6}	N



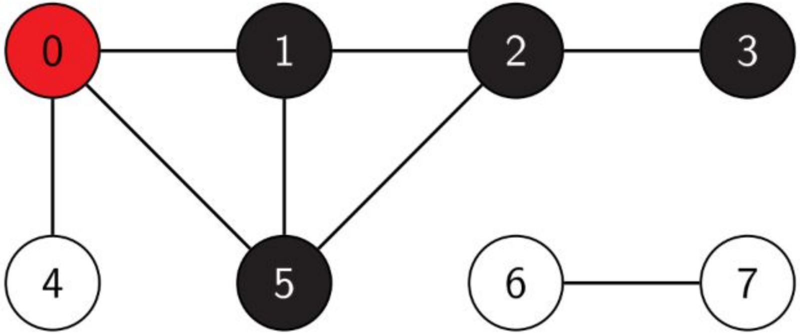
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	S
6	{7}	N
7	{6}	N



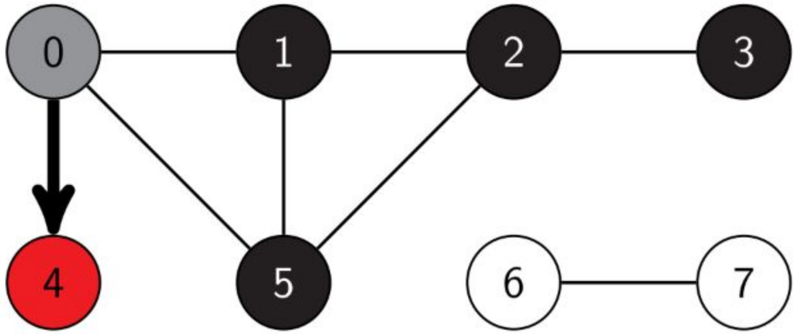
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	N
5	{0,1,2}	S
6	{7}	N
7	{6}	N



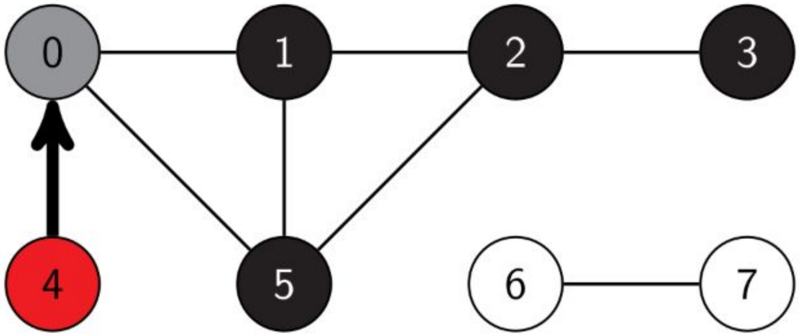
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	S
5	{0,1,2}	S
6	{7}	N
7	{6}	N



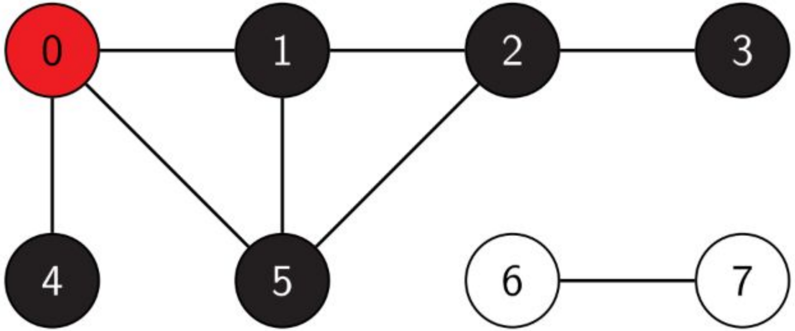
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	S
5	{0,1,2}	S
6	{7}	N
7	{6}	N



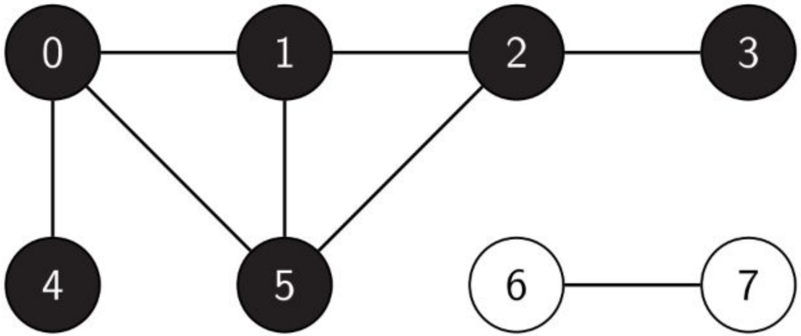
Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	S
5	{0,1,2}	S
6	{7}	N
7	{6}	N



Busca em Profundidade (DFS)

Vértice	Vizinhos	Visitado
0	{1,4,5}	S
1	{0,2,5}	S
2	{1,3,5}	S
3	{2}	S
4	{0}	S
5	{0,1,2}	S
6	{7}	N
7	{6}	N



Busca em Profundidade (DFS)

Implementação

```cpp
const int MAX = 1e5+10; // Número máximo de vértices

vector<vector<int>> adj(MAX);
vector<bool> vis(MAX);

void dfs(int v) {
    vis[v] = true;
    for (auto u : adj[v]) if (!vis[u]) {
        dfs(u);
    }
}
```

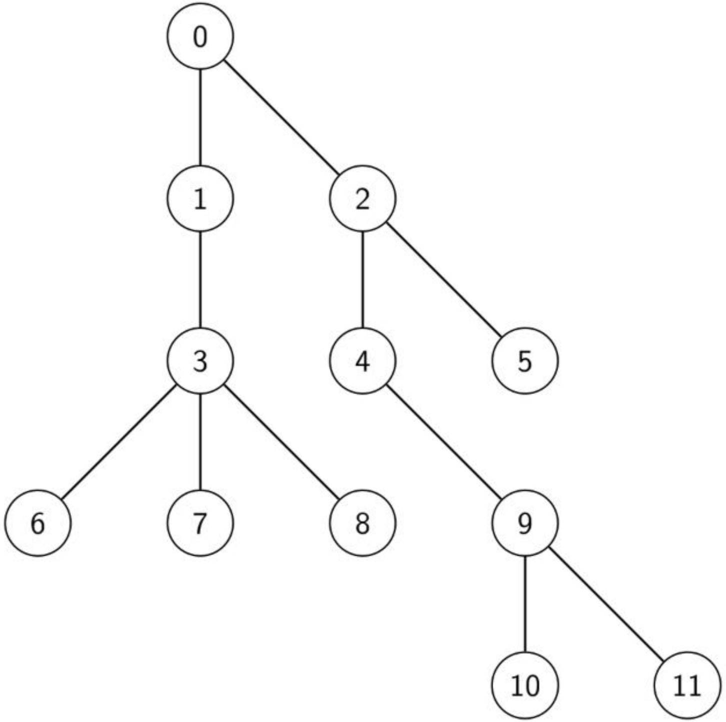
Solução: Gincana (OBI 2011)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define endl '\n'
#define pb push_back

const int MAX = 1e3 + 10;

int n, m;
vector<int> adj[MAX]; bool vis[MAX];

void dfs(int v) {
    vis[v] = true;
    for (auto u : adj[v]) if (!vis[u]) {
        dfs(u);
    }
}
```

```cpp
signed main(){
    ios_base::sync_with_stdio(0);cin.tie(0);
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u, v; cin >> u >> v; u--, v--;
        adj[u].pb(v);
        adj[v].pb(u);
    }
    int comp = 0;
    for (int v = 0; v < n; v++) {
        if (!vis[v]) {
            dfs(v);
            comp++;
        }
    }
    cout << comp << endl;
}
```

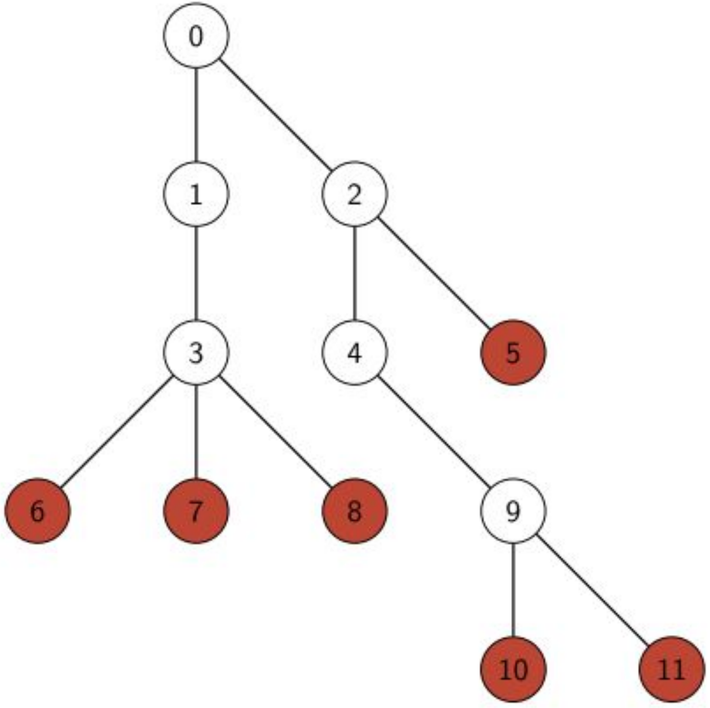
Árvores

- Um grafo conexo acíclico (sem ciclos) é chamado de **árvore**.



Árvores

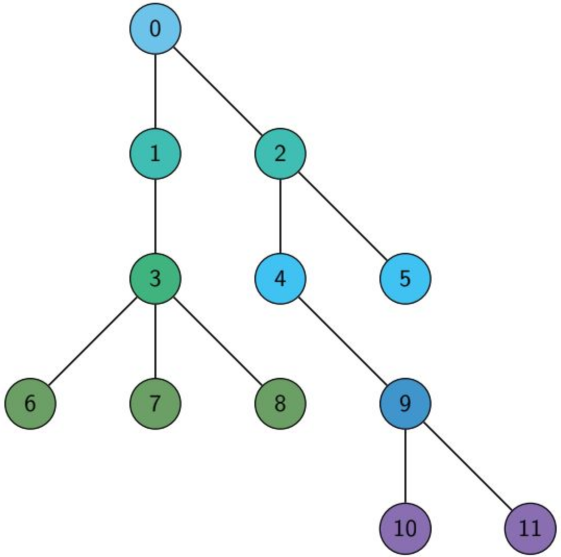
- Em um árvore, vértices de grau 1 são chamados **folhas**.



Árvores

- Uma árvore pode ser enraizada em algum vértice chamado **raiz** da árvore.

- A partir desse vértice, diremos que os vértices conectados por uma aresta são **pais** e **filhos**, sendo os vértices de baixo os filhos dos superiores.



Árvores

São afirmações equivalentes

- G é uma árvore.
- G é um grafo conexo acíclico.
- G é um grafo conexo e $|E| = |V| - 1$.
- G é um grafo em que para todo $v, w \in G$ existe exatamente um caminho de v para w.
- G é conexo e tirar qualquer aresta de G o torna desconexo.
- G não contém ciclos, mas se adicionarmos qualquer aresta à G criaria um ciclo.

Problema Motivador: Kefa and Park (Codeforces)

Kefa mora em um parque que pode ser representado como uma **árvore** com n vértices, sendo o vértice 1 a raiz, onde fica a casa dele. **Alguns vértices do parque têm gatos**.

Os restaurantes estão localizados nos **vértices folha**.

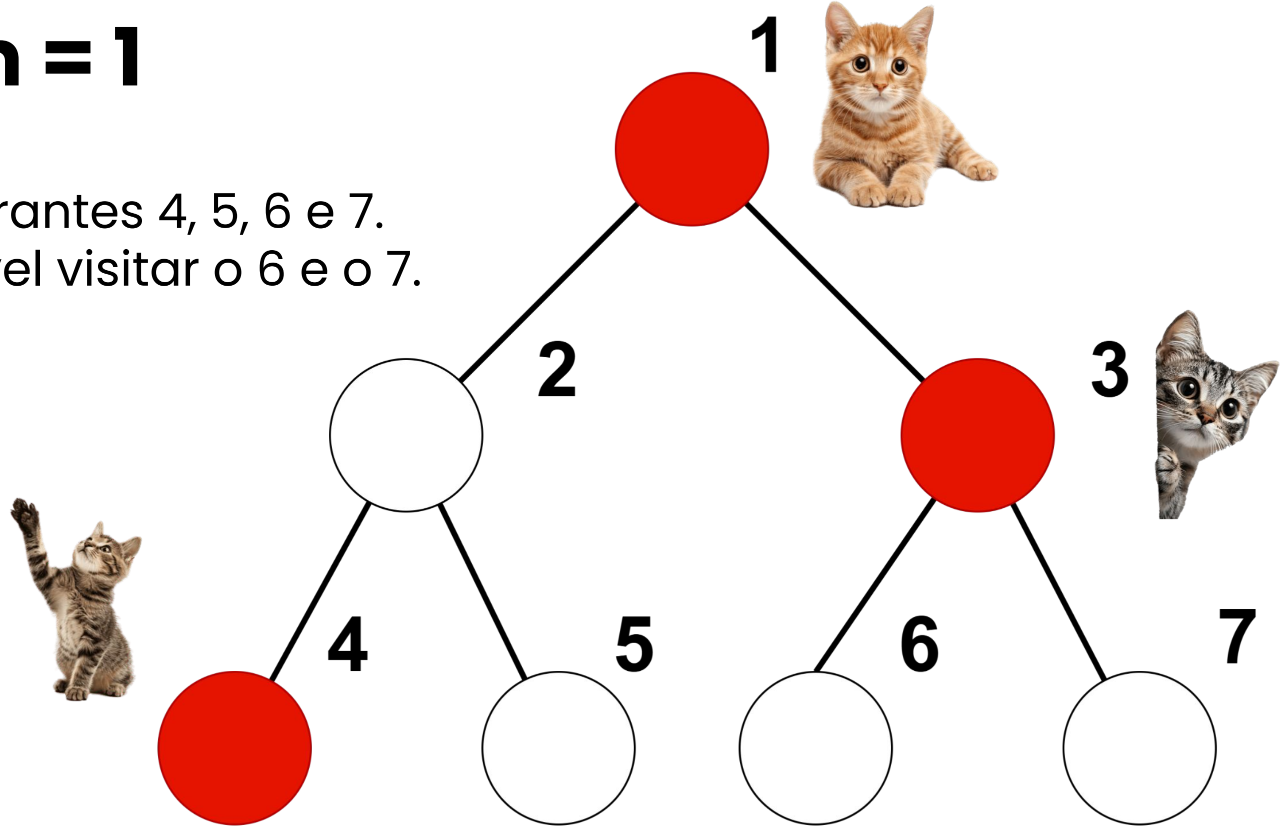
Kefa quer escolher um restaurante para ir, mas ele tem medo de gatos. Ele só irá a um restaurante se o caminho da casa dele até esse restaurante não tiver mais do que **m vértices consecutivos com gatos**.

Você deve determinar quantos restaurantes Kefa pode visitar, ou seja, quantos vértices folha têm um caminho válido até a raiz, respeitando o limite de m gatos consecutivos.

Problema Motivador: [Kefa and Park (Codeforces)]

m = 1

Dos restaurantes 4, 5, 6 e 7.
Não é possível visitar o 6 e o 7.



Solução: Kefa and Park (Codeforces)

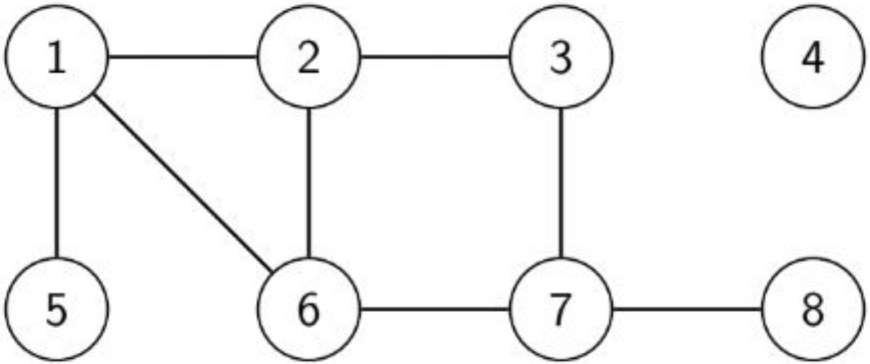
```cpp
#include <bits/stdc++.h>
using namespace std;

#define endl '\n'
#define pb push_back

const int MAXN = 2e5 + 10;
int n, m, ans;
vector<int> adj[MAXN];
bool cats[MAXN], vis[MAXN];

void dfs(int v, int cnt) { // cnt -> armazena o contador de gatos consecutivos
    vis[v] = true;
    // Verificar a presença de gato
    if (cats[v]) cnt++;
    else cnt = 0;

    // Finaliza a busca para cnt > m
    if (cnt > m) return;

    bool is_leaf = true;
    for (auto u : adj[v]) if (!vis[u]) {
        is_leaf = false;
        dfs(u, cnt);
    }

    // Chegou em uma folha?
    if (is_leaf) ans++;
}
```

```cpp
signed main(){
    ios_base::sync_with_stdio(0);cin.tie(0);
    cin >> n >> m;
    for (int i = 0; i < n; i++) cin >> cats[i];
    for (int i = 0; i < n-1; i++) {
        int u, v; cin >> u >> v;
        u--; v--;
        adj[u].pb(v);
        adj[v].pb(u);
    }
    dfs(0, 0);
    cout << ans << endl;
}
```

Problema Motivador: [Message Route (CSES)]

Dada uma rede de **N computadores** e **M conexões**, queremos saber se é possível enviar uma mensagem do computador 1 para o computador N.

Caso seja, qual o menor número de computadores nessa rota e quais são eles?



- Possíveis caminhos: {1, 2, 3, 7, 8}, {1, 2, 6, 7, 8} e **{1, 6, 7, 8}**.

- Como fazer um algoritmo para descobrir o menor caminho?

Busca em Largura (BFS)

- Da necessidade de um algoritmo que descobre o **menor caminho** entre vértices, surge o **Breadth First Search (BFS)** ou **Busca em Largura**.

- O algoritmo pode ser entendido como um fogo se espalhando pelo grafo.
    - Partindo de um vértice *s*, visitamos todos os vizinhos, depois todos os vizinhos dos vizinhos, e assim sucessivamente.

- Sempre visitamos os vértices a uma distância $d - 1$ antes de visitar os vértices a uma distância d.

- **Complexidade:** $O(|V| + |E|)$

Busca em Largura (BFS)

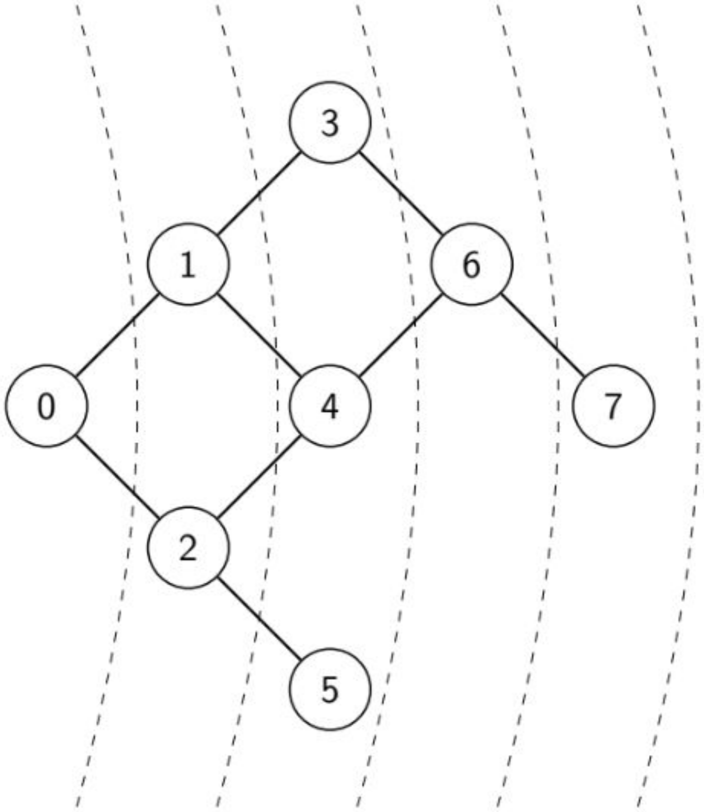
- Da necessidade de um algoritmo que descobre o **menor caminho** entre vértices, surge o **Breadth First Search (BFS)** ou **Busca em Largura**.

- O algoritmo pode ser entendido como um fogo se espalhando pelo grafo.
  - Partindo de um vértice *s*, visitamos todos os vizinhos, depois todos os vizinhos dos vizinhos, e assim sucessivamente.

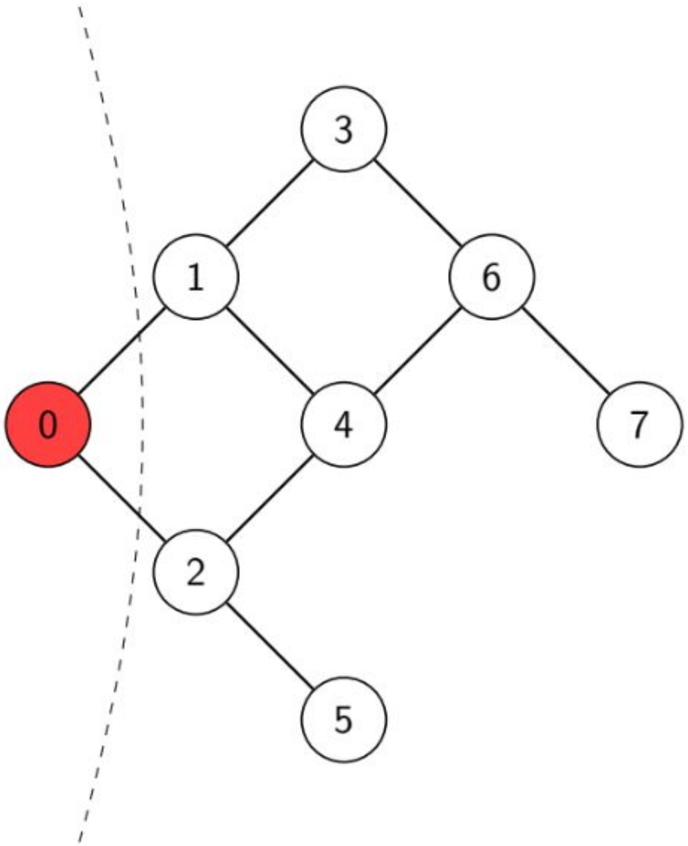
- Sempre visitamos os vértices a uma distância $d - 1$ antes de visitar os vértices a uma distância *d*.

- **Complexidade:** $O(|V| + |E|)$

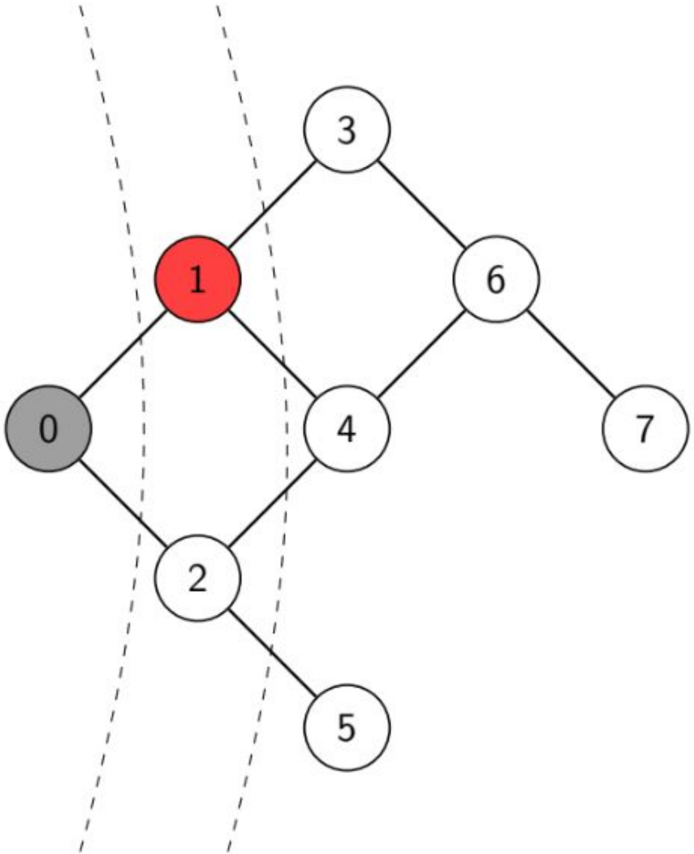
Busca em Largura (BFS)



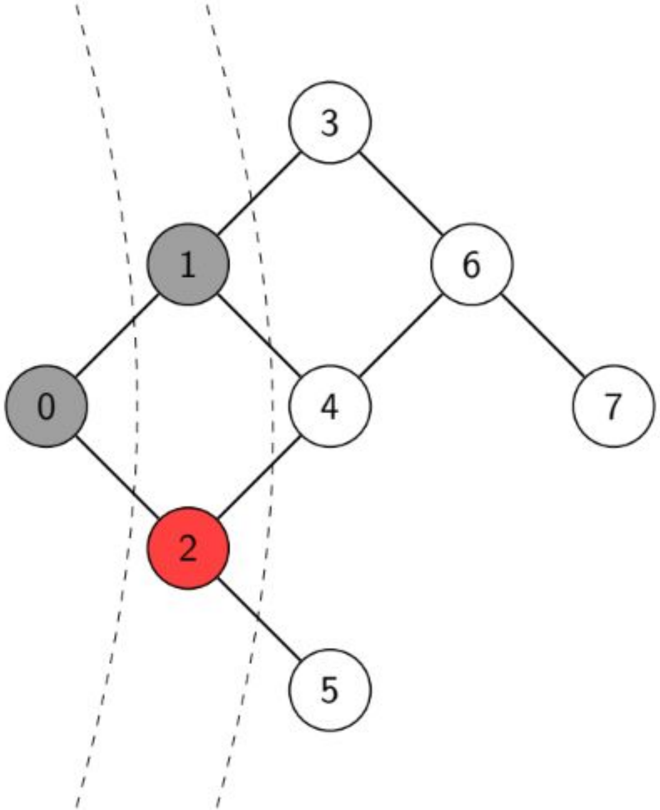
Busca em Largura (BFS)



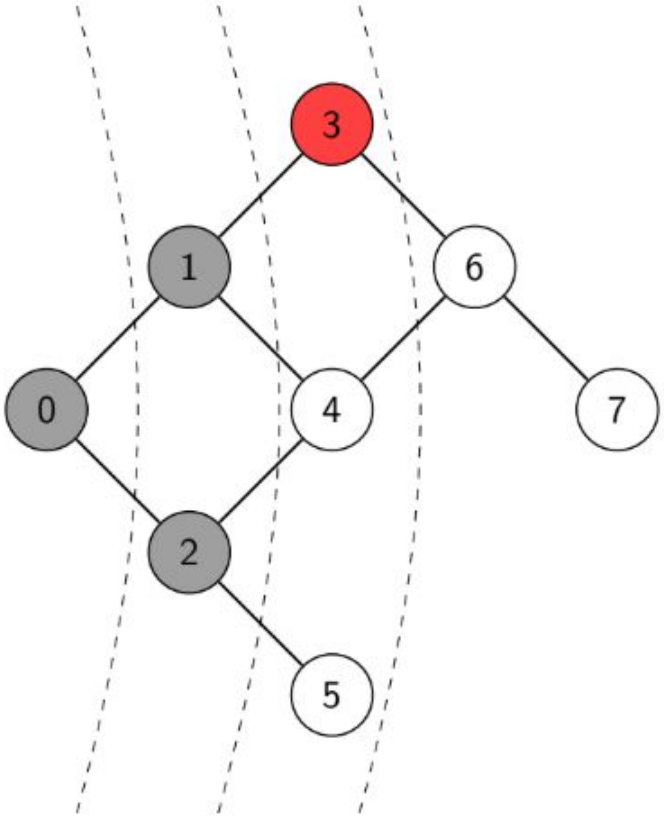
Busca em Largura (BFS)



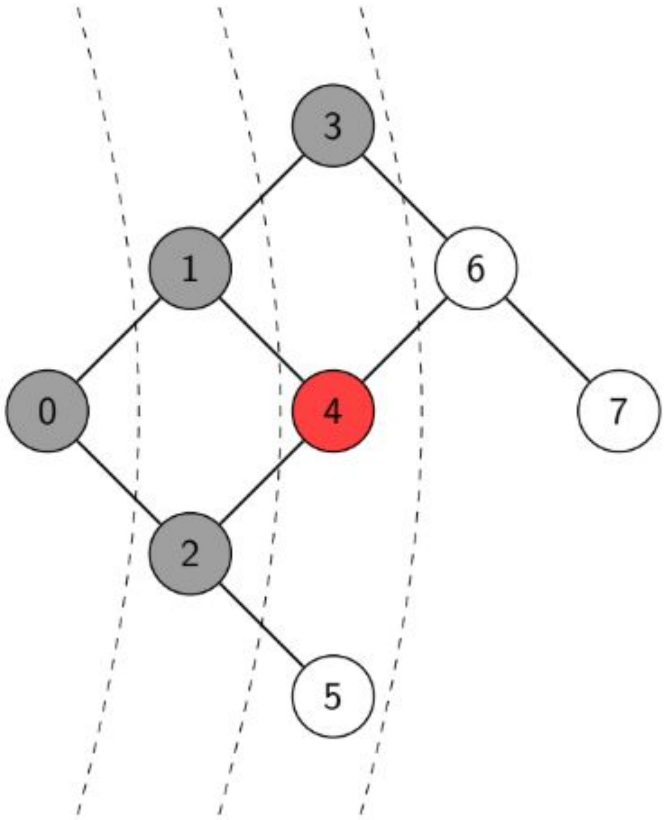
Busca em Largura (BFS)



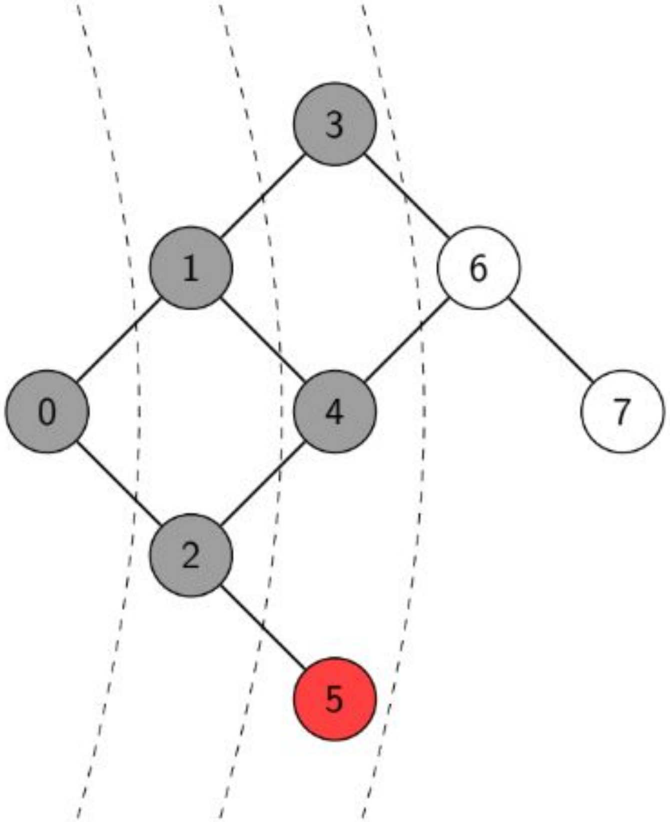
Busca em Largura (BFS)



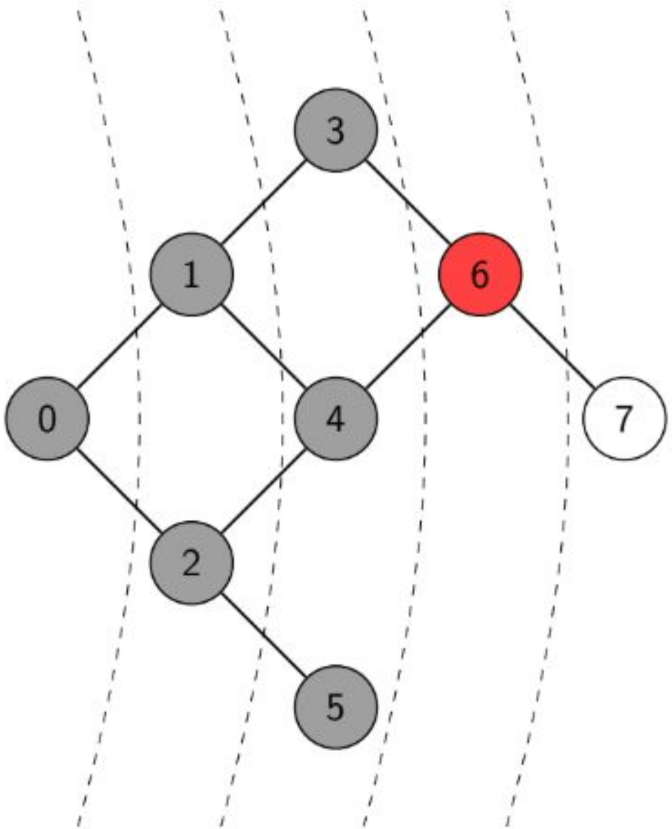
Busca em Largura (BFS)



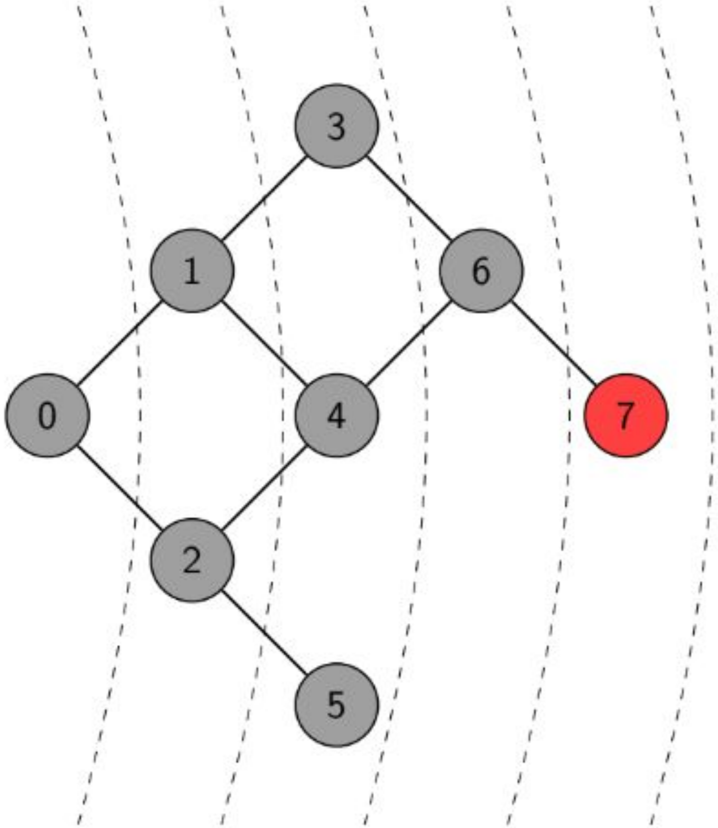
Busca em Largura (BFS)



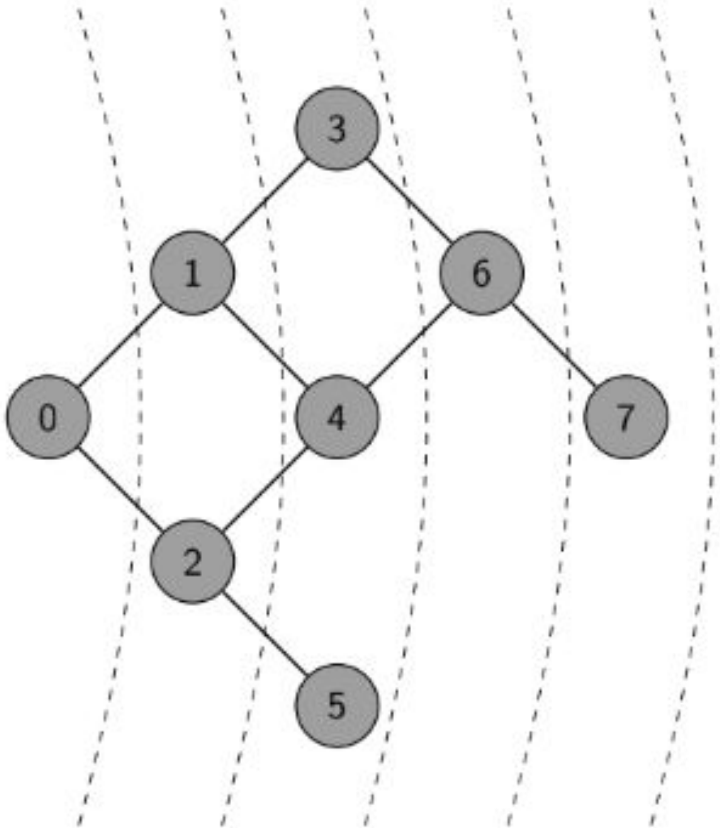
Busca em Largura (BFS)



Busca em Largura (BFS)



Busca em Largura (BFS)



Busca em Largura (BFS)

- Visitamos primeiro os vértices a uma distância *d* da fonte antes de visitar todos os vértices a uma distância *d* + 1.
- Precisamos de uma estrutura que prioriza os elementos que colocamos primeiro.
- Uma **fila** atende esses requisitos.
- A BFS só resolve o problema de **menor caminho** para grafos **não-ponderados**.

Busca em Largura (BFS)

```
BFS(vértice s)  {
     fila.insere(s)
     visitado[s] = verdadeiro
     Enquanto fila não estiver vazia:
          v = fila.frente()
          fila.remove()
          visitado[v] = verdadeiro
          Para todo vizinho w de v:
               Se w ainda não foi visitado:
                    fila.insere(w)
                    visitado[w] = verdadeiro
}
```

Busca em Largura (BFS)

Implementação

```cpp
const int MAXN = 1e5 + 10;

int n, m;
vector<vector<int>> adj(MAXN);
vector<bool> vis(MAXN);

void bfs(int s) {
    queue<int> q;
    q.push(s); vis[s] = true;
    while (!q.empty()) {
        int v = q.front(); q.pop();
        for (auto u : adj[v]) if (!vis[u]) {
            q.push(u); vis[u] = true;
        }
    }
}
```

Solução: Message Route (CSES)

```cpp
#include <bits/stdc++.h>
using namespace std;

#define endl '\n'
#define pb push_back
#define int long long int

const int MAXN = 1e5 + 10;

int n, m;
vector<int> adj[MAXN], dist(MAXN, -1), parent(MAXN, -1);
bool vis[MAXN];

void bfs(int s) {
    queue<int> q;
    q.push(s); vis[s] = true;
    dist[s] = 0;
    parent[s] = -1;
    while (!q.empty()) {
        int v = q.front(); q.pop();
        for (auto u : adj[v]) if (!vis[u]) {
            dist[u] = dist[v] + 1;
            q.push(u); vis[u] = true;
            parent[u] = v;
        }
    }
}
```

```cpp
signed main(){
    ios_base::sync_with_stdio(0);cin.tie(0);
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u, v; cin >> u >> v;
        u--; v--;
        adj[u].pb(v);
        adj[v].pb(u);
    }
    bfs(0);
    if (!vis[n-1]) {
        cout << "IMPOSSIBLE" << endl;
    } else {
        cout << (dist[n-1] + 1) << endl;
        vector<int> path;
        int v = n-1;
        for (int i = 0; i <= dist[n-1]; i++) {
            path.pb(v);
            v = parent[v];
        }
        reverse(path.begin(), path.end());
        for (auto x : path) cout << x+1 << " ";
        cout << endl;
    }
}
```

Referências

- Grafos: conceitos fundamentais, , algoritmos e aplicações
- Uma Introdução Sucinta à Teoria dos Grafos
- Breadth First Search - Algorithms for Competitive Programming
- Depth First Search - Algorithms for Competitive Programming
- Slides da Maratona UFMG

Dúvidas?

Obrigado pela atenção