Aula 11 · Ordenação Topológica, Componentes Conexos e Biconexos

Desafios de Programação

Fernando Kiotheka Vinícius Tikara Date

UFPR

31/05/2023

Componentes Conexos e Ordenação Topológica

O que é um componente?

- Um componente do grafo é uma subgrafo onde todos os vértices são conectados (existe um caminho de todos os vértices para todos os vértices).
- Da mesma forma que existem grafos fortemente conexos e grafos fracamente conexos, existem componentes fortemente conexos e componentes fracamente conexos.
- Como descobrir componentes em um grafo não direcionado? Uma busca em profundidade é o jeito mais simples.

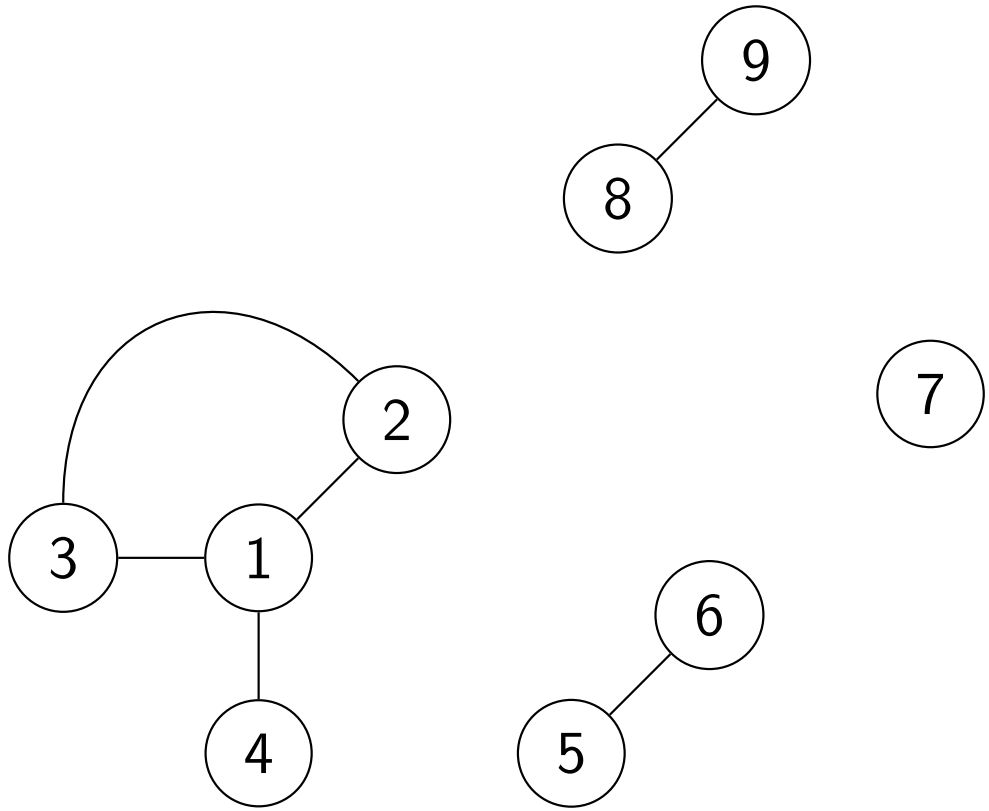
Busca de componentes em um grafo não direcionado

Código `rep.h`

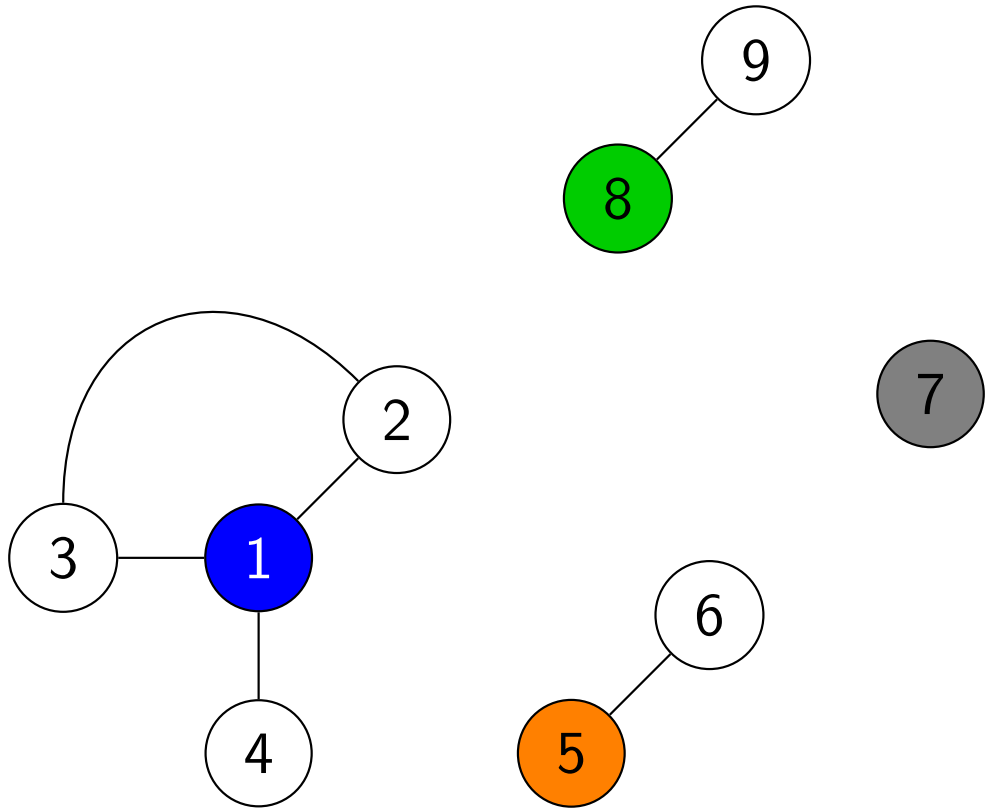
```
vector<vector<int>> g (N);
vector<int> rep (N);

void mark_component (int u, int r) {
    if (vis[u] == cts) { return; }
    vis[u] = cts;
    rep[u] = r;
    for (int v : g[u]) { mark_component(v, r); }
}
```

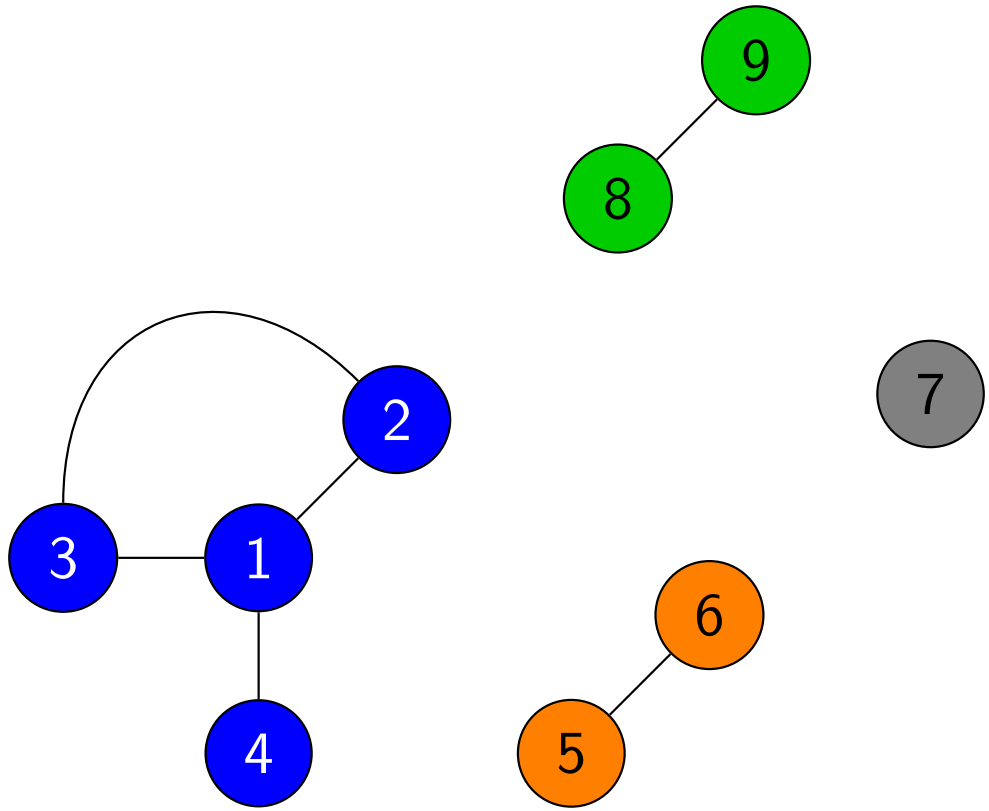
Ideia para achar componentes em um grafo não direcionado



Ideia para achar componentes em um grafo não direcionado



Ideia para achar componentes em um grafo não direcionado



Achando componentes em um grafo não direcionado

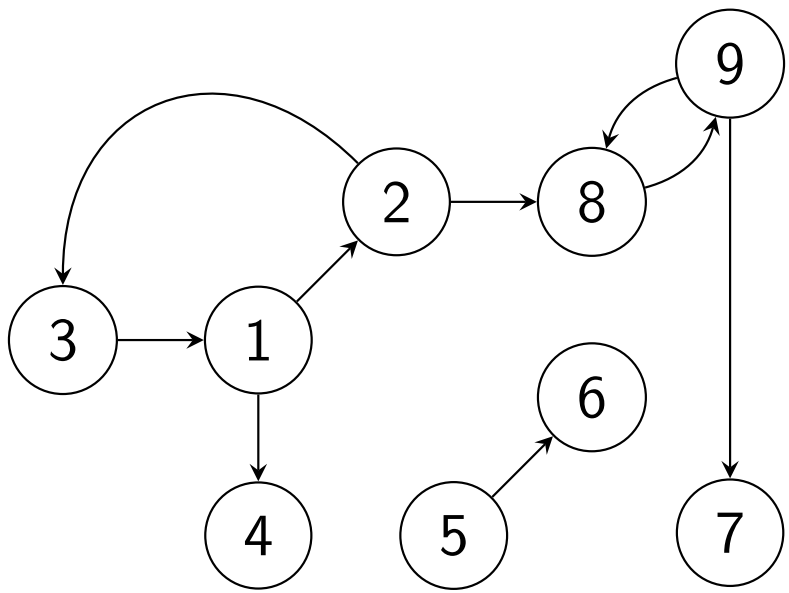
Código `rep.cpp`

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15; int cts = 1; vector<int> vis (N);
#include "rep.h"
int main() { int n, m; cin >> n >> m;
    while (m--) { int u, v; cin >> u >> v; u--; v--;
        g[u].push_back(v); g[v].push_back(u); }
    for (int u = 0; u < n; u++) mark_component(u, u);
    for (int u = 0; u < n; u++) cout << rep[u]+1 << " ";
}
```

Entrada	Saída
9 5 1 2 1 3 1 4 8 9 5 6	1 1 1 1 5 5 7 8 8

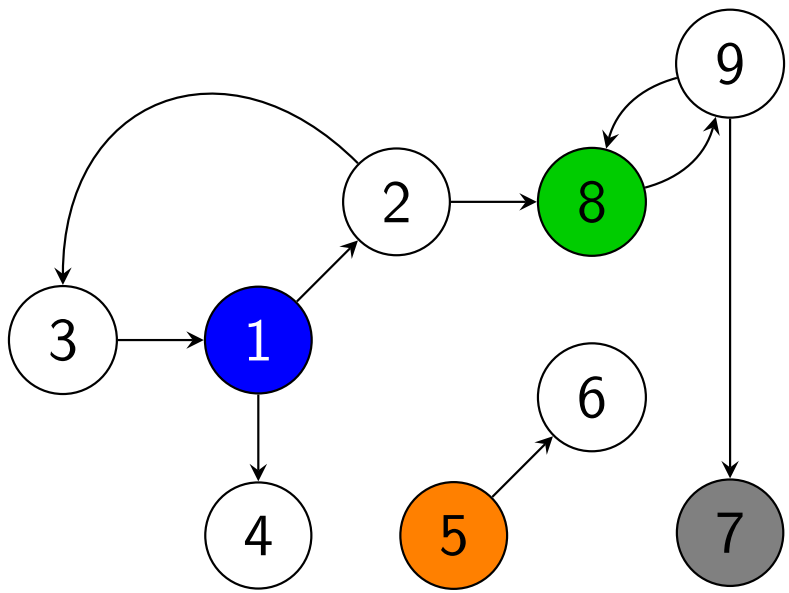
E em grafos direcionados?

- Se quisermos achar os componentes fracamente conexos de um grafo direcionado, é só criar um grafo não direcionado com base nele e procurar usando o método anterior.
- Mas se quisermos achar os componentes fortemente conexos, não podemos fazer em qualquer ordem, senão dá errado!



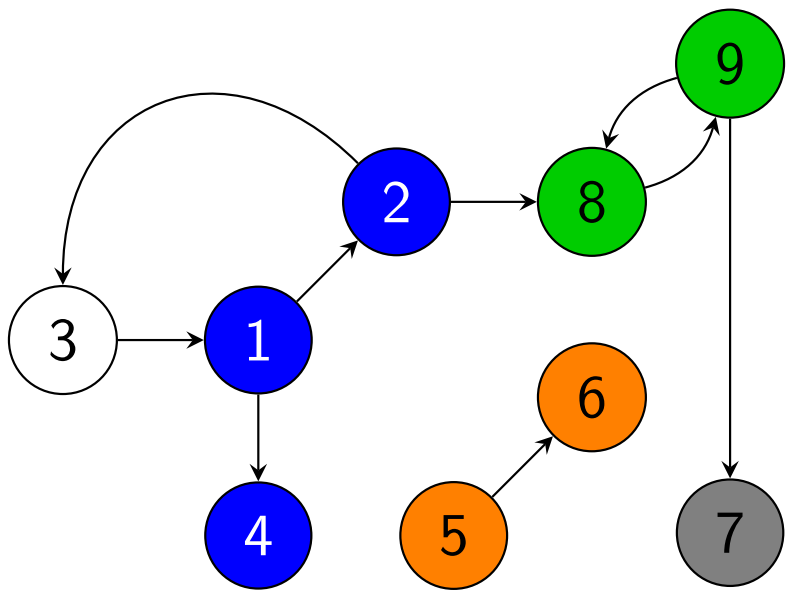
E em grafos direcionados?

- Se quisermos achar os componentes fracamente conexos de um grafo direcionado, é só criar um grafo não direcionado com base nele e procurar usando o método anterior.
- Mas se quisermos achar os componentes fortemente conexos, não podemos fazer em qualquer ordem, senão dá errado!



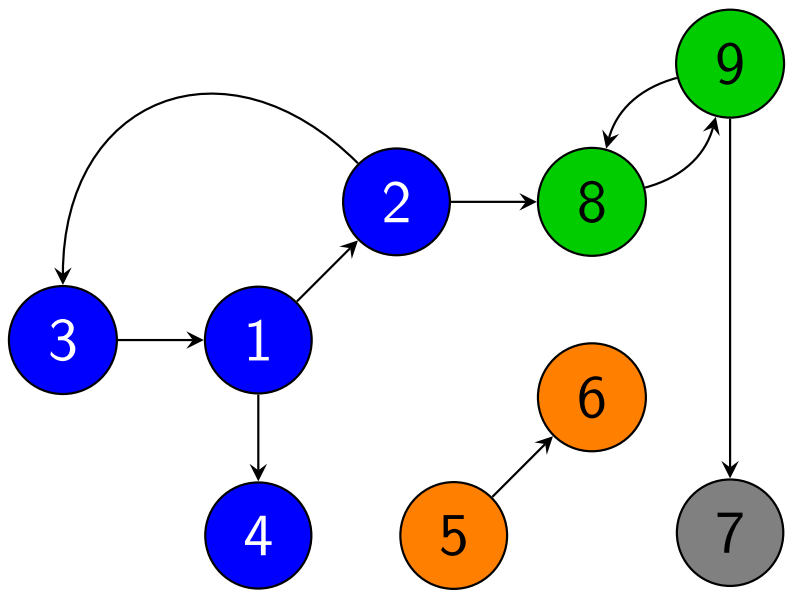
E em grafos direcionados?

- Se quisermos achar os componentes fracamente conexos de um grafo direcionado, é só criar um grafo não direcionado com base nele e procurar usando o método anterior.
- Mas se quisermos achar os componentes fortemente conexos, não podemos fazer em qualquer ordem, senão dá errado!



E em grafos direcionados?

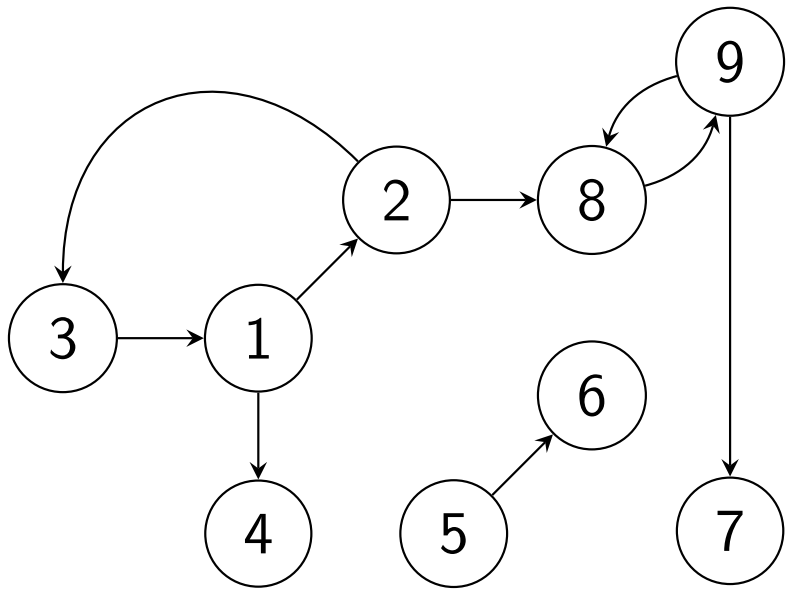
- Se quisermos achar os componentes fracamente conexos de um grafo direcionado, é só criar um grafo não direcionado com base nele e procurar usando o método anterior.
- Mas se quisermos achar os componentes fortemente conexos, não podemos fazer em qualquer ordem, senão dá errado!



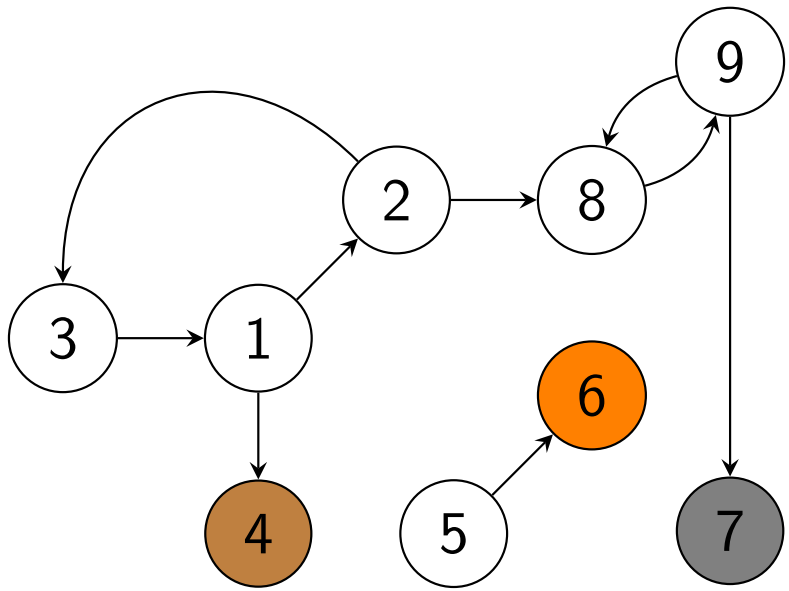
Então qual ordem é a ideal?

- A ordem ideal é a de iterar primeiro sobre os **sumidouros**.
- Um sumidouro é um vértice que só tem arestas de entrada.
- O contrário são os **fontes**, que só tem arestas de saída.
- Em um grafo direcionado *acíclico*, obrigatoriamente existe pelo menos uma fonte e um sumidouro.
- Então a ideia é iterar sobre os sumidouros até eles *sumirem*.
- Daí só sobrarão os ciclos, que podem ser percorridos em qualquer ordem.

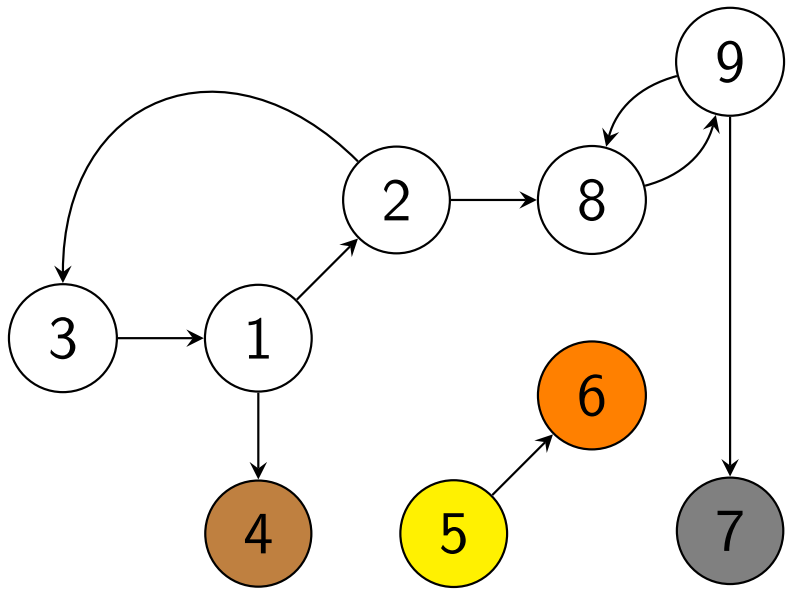
Iterando sobre sumidouros e então ciclos



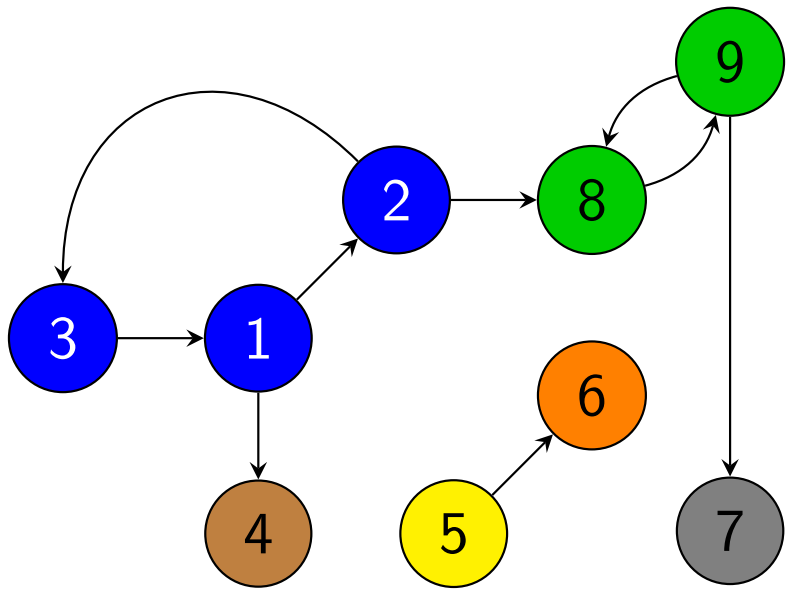
Iterando sobre sumidouros e então ciclos



Iterando sobre sumidouros e então ciclos



Iterando sobre sumidouros e então ciclos

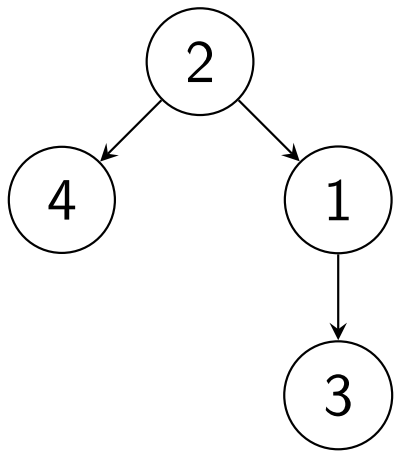


Leve *detour*: Ordenação topológica

- O problema da ordenação topológica é aquele de encontrar uma ordem para os vértices de um grafo direcionado que faça sentido em relação às arestas.
- A ordenação que obteremos nos permitirá resolver dependências. Sempre que uma coisa depender de outra, ela só será executada se todas as suas dependências estiverem satisfeitas.
- Em um grafo direcionado acíclico, para toda aresta (u, v), u deve aparecer antes de v na ordenação topológica.
- Caso existam ciclos, eles serão feitos em ordem arbitrária (mas as dependências dos ciclos serão resolvidas).
- Podemos fazer isso com uma busca em profundidade.

Ordenação topológica usando busca em profundidade

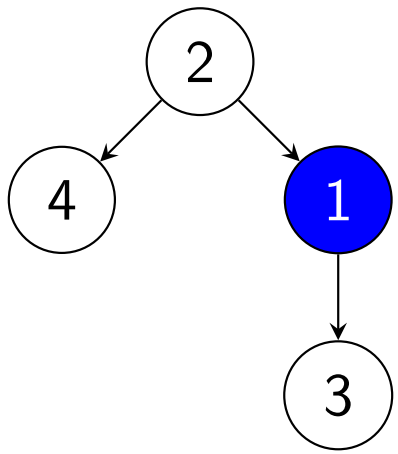
- Vértices que visitamos primeiro devem vir primeiro.
- Porém, não podemos fazer isso ao entrar na busca!
- No grafo abaixo, obteríamos a ordem 1 3 2 4 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Porém 1 vem depois de 2, como arrumar?

Ordenação topológica usando busca em profundidade

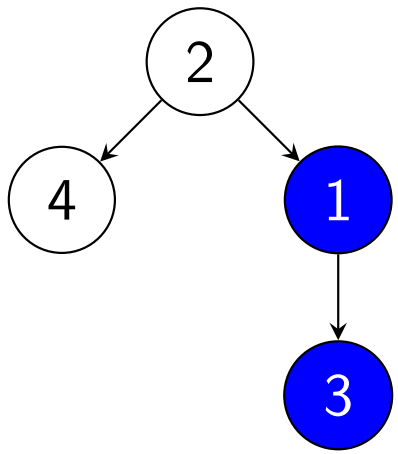
- Vértices que visitamos primeiro devem vir primeiro.
- Porém, não podemos fazer isso ao entrar na busca!
- No grafo abaixo, obteríamos a ordem 1 3 2 4 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Porém 1 vem depois de 2, como arrumar?

Ordenação topológica usando busca em profundidade

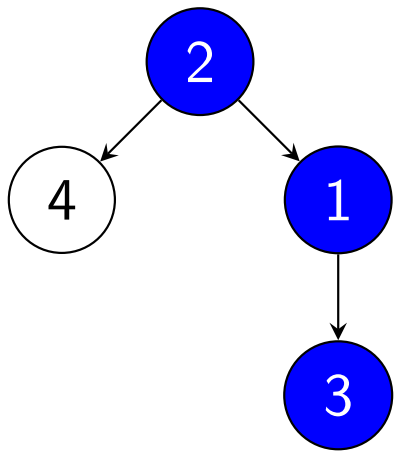
- Vértices que visitamos primeiro devem vir primeiro.
- Porém, não podemos fazer isso ao entrar na busca!
- No grafo abaixo, obteríamos a ordem 1 3 2 4 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Porém 1 vem depois de 2, como arrumar?

Ordenação topológica usando busca em profundidade

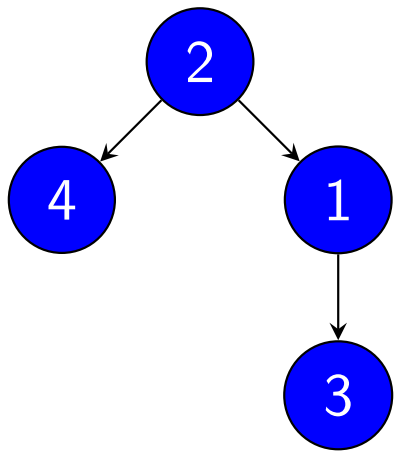
- Vértices que visitamos primeiro devem vir primeiro.
- Porém, não podemos fazer isso ao entrar na busca!
- No grafo abaixo, obteríamos a ordem 1 3 2 4 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Porém 1 vem depois de 2, como arrumar?

Ordenação topológica usando busca em profundidade

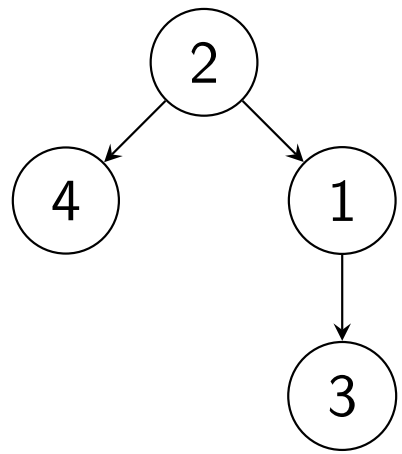
- Vértices que visitamos primeiro devem vir primeiro.
- Porém, não podemos fazer isso ao entrar na busca!
- No grafo abaixo, obteríamos a ordem 1 3 2 4 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Porém 1 vem depois de 2, como arrumar?

Ordenação topológica usando busca em profundidade

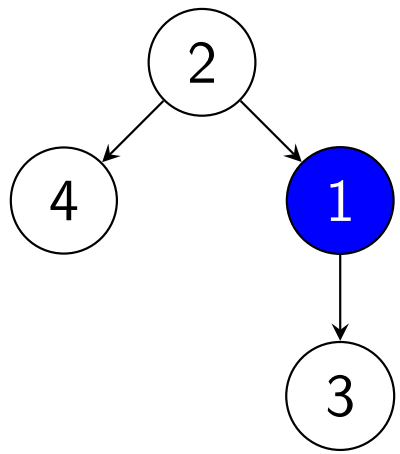
- Em vez de um vetor, guardamos os vértices numa pilha.
- Pegamos o resultado da pilha ao **final** da função.
- No grafo abaixo, obtemos a ordem empilhada 3 1 4 2 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Ao desempilhar, obtemos a ordem 2 4 1 3 que é uma ordem topológica válida!

Ordenação topológica usando busca em profundidade

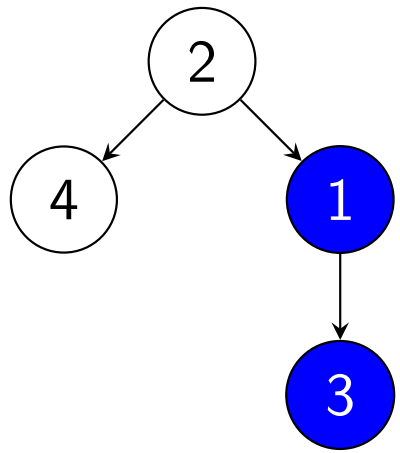
- Em vez de um vetor, guardamos os vértices numa pilha.
- Pegamos o resultado da pilha ao **final** da função.
- No grafo abaixo, obtemos a ordem empilhada 3 1 4 2 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Ao desempilhar, obtemos a ordem 2 4 1 3 que é uma ordem topológica válida!

Ordenação topológica usando busca em profundidade

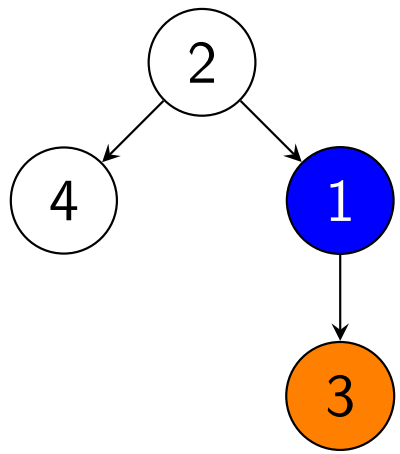
- Em vez de um vetor, guardamos os vértices numa pilha.
- Pegamos o resultado da pilha ao **final** da função.
- No grafo abaixo, obtemos a ordem empilhada 3 1 4 2 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Ao desempilhar, obtemos a ordem 2 4 1 3 que é uma ordem topológica válida!

Ordenação topológica usando busca em profundidade

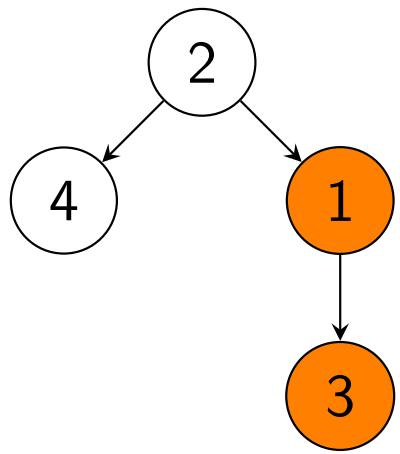
- Em vez de um vetor, guardamos os vértices numa pilha.
- Pegamos o resultado da pilha ao **final** da função.
- No grafo abaixo, obtemos a ordem empilhada 3 1 4 2 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Ao desempilhar, obtemos a ordem 2 4 1 3 que é uma ordem topológica válida!

Ordenação topológica usando busca em profundidade

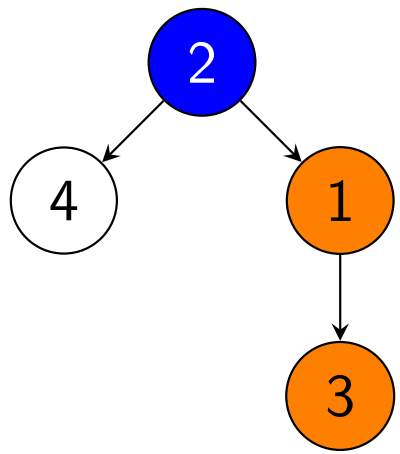
- Em vez de um vetor, guardamos os vértices numa pilha.
- Pegamos o resultado da pilha ao **final** da função.
- No grafo abaixo, obtemos a ordem empilhada 3 1 4 2 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Ao desempilhar, obtemos a ordem 2 4 1 3 que é uma ordem topológica válida!

Ordenação topológica usando busca em profundidade

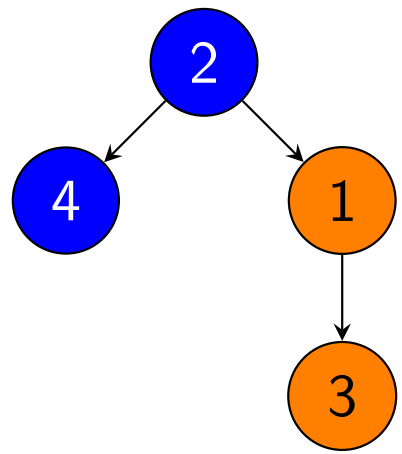
- Em vez de um vetor, guardamos os vértices numa pilha.
- Pegamos o resultado da pilha ao **final** da função.
- No grafo abaixo, obtemos a ordem empilhada 3 1 4 2 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Ao desempilhar, obtemos a ordem 2 4 1 3 que é uma ordem topológica válida!

Ordenação topológica usando busca em profundidade

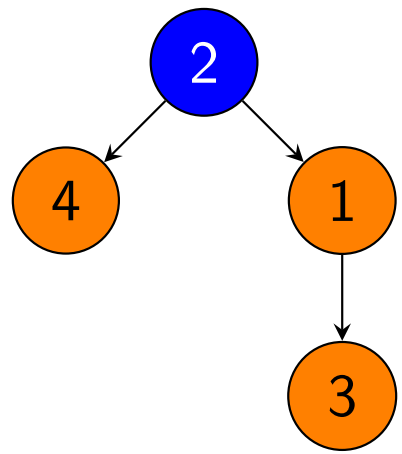
- Em vez de um vetor, guardamos os vértices numa pilha.
- Pegamos o resultado da pilha ao **final** da função.
- No grafo abaixo, obtemos a ordem empilhada 3 1 4 2 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Ao desempilhar, obtemos a ordem 2 4 1 3 que é uma ordem topológica válida!

Ordenação topológica usando busca em profundidade

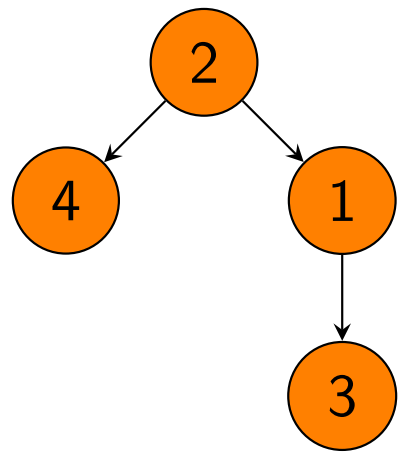
- Em vez de um vetor, guardamos os vértices numa pilha.
- Pegamos o resultado da pilha ao **final** da função.
- No grafo abaixo, obtemos a ordem empilhada 3 1 4 2 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



- Ao desempilhar, obtemos a ordem 2 4 1 3 que é uma ordem topológica válida!

Ordenação topológica usando busca em profundidade

- Em vez de um vetor, guardamos os vértices numa pilha.
- Pegamos o resultado da pilha ao **final** da função.
- No grafo abaixo, obtemos a ordem empilhada 3 1 4 2 se começarmos uma busca por 1 (que revela 3) e então por 2 (que revela 4).



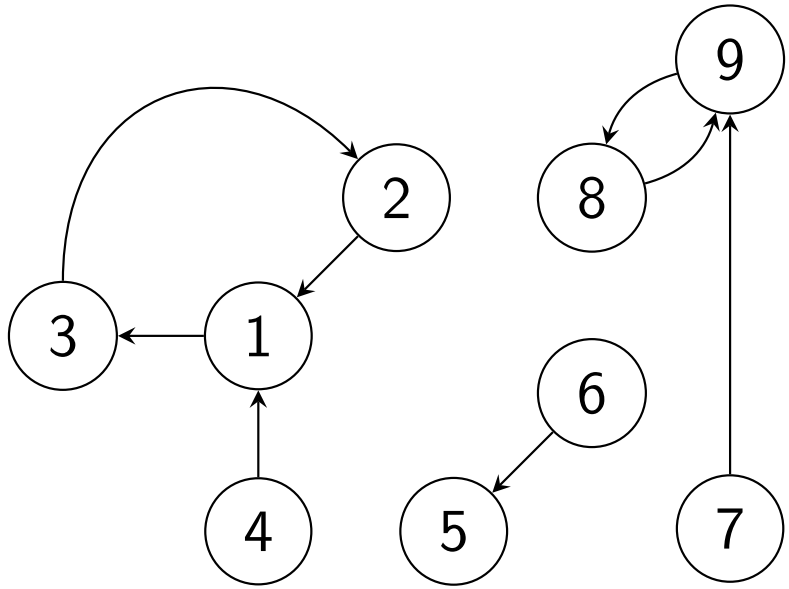
- Ao desempilhar, obtemos a ordem 2 4 1 3 que é uma ordem topológica válida!

Implementação da ordenação topológica

Código topo.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15;
vector<vector<int>> g (N); vector<bool> vis (N);
stack<int> st;
void dfs_topo(int u) {
    if (vis[u]) { return; } vis[u] = 1;
    for (int v : g[u]) { dfs_topo(v); }
    st.push(u); }
vector<int> toposort(int n) {
    vector<int> topo;
    for (int u = 0; u < n; u++) { dfs_topo(u); }
    while (!st.empty()) { int v = st.top(); st.pop();
        topo.push_back(v); }
    return topo; }
int main() { int n, m; cin >> n >> m;
    while (m--) { int u, v; cin >> u >> v; u--; v--;
        g[u].push_back(v); }
    for (int u : toposort(n)) { cout << u+1 << " "; }
}
```

Implementação da ordenação topológica (continuado)



Entrada	Saída
9 8 2 1 3 2 1 3 4 1 9 8 8 9 6 5 7 9	7 9 8 6 5 4 1 3 2

Então de volta ao problema... Será que essa ordem serve?

- Estávamos pensando anteriormente que seria bom iterar sobre os sumidouros primeiro para obter os nossos componentes fortemente conexos no grafo direcionado.
- O problema é que a ordem topológica nos dá a ordem começando pelas fontes, e não sumidouros...
- Mas tem um jeito simples de tornar sumidouros fontes e fontes sumidouros: Inverter todas as arestas do grafo!
- A melhor parte é que os ciclos continuam sendo os mesmos nesse grafo chamado de transposto.
- Assim, saberemos onde estão os sumidouros no grafo original e poderemos rodar o algoritmo na ordem correta!

Implementação da ordem dos sumidouros

Código `stack.h`

```
vector<vector<int>> g_t (N);
stack<int> sinks;

void fill_stack(int u) {
    if (vis[u] == cts) { return; }
    vis[u] = cts;
    for (int v : g_t[u]) { fill_stack(v); }
    sinks.push(u);
}
```

Juntando tudo: O Algoritmo de Kosaraju

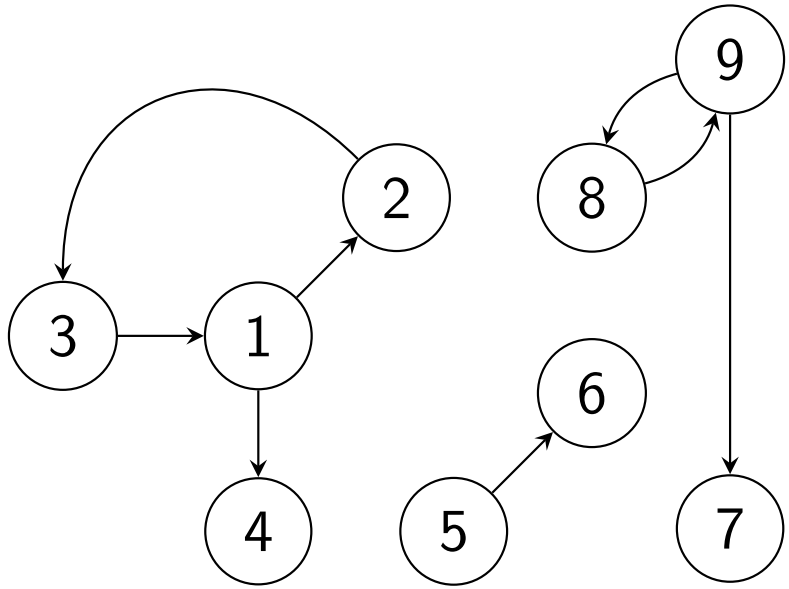
- Juntando a parte de achar a ordem dos sumidouros com a marcação das componentes, inventamos o Algoritmo de Kosaraju!
- Só que a gente meio que fez o Algoritmo de Kosaraju ao contrário, preenchendo a pilha com os vértices do grafo transposto ao invés de preencher os componentes no grafo transposto.
- Porém, não importa, afinal o Algoritmo de Kosaraju se baseia justamente nesse conceito: O grafo transposto tem exatamente os mesmos componentes fortemente conexos do grafo original.
- Só tome esse cuidado ao encontrar esse algoritmo na Internet, ele vai estar diferente do disposto aqui mas o funcionamento é idêntico.

Implementação do Kosaraju

Código kosaraju.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15; int cts = 1; vector<int> vis (N);
#include "stack.h"
#include "rep.h"
vector<int> kosaraju(int n) {
    for (int u = 0; u < n; u++) { fill_stack(u); } cts++;
    vector<int> topo_components;
    while (!sinks.empty()) {
        int u = sinks.top(); sinks.pop();
        mark_component(u, u);
        if (rep[u] == u) topo_components.push_back(u); }
    return topo_components;
}
int main() { int n, m; cin >> n >> m;
    while (m--) { int u, v; cin >> u >> v; u--; v--;
        g[u].push_back(v); g_t[v].push_back(u); }
    kosaraju(n);
    for (int u = 0; u < n; u++) cout << rep[u]+1 << " ";
}
```

Implementação do Kosaraju (continuado)



Entrada	Saída
9 8 1 2 2 3 3 1 1 4 8 9 9 8 5 6 9 7	1 1 1 4 5 6 7 9 9

A função *lowlink*

Guardaremos:

- O momento em que uma DFS chega pela primeira vez a cada vértice (t)
- O menor momento que conseguimos alcançar a partir de cada vértice, visitando seus vizinhos (l)

Inicialmente t e l são iguais (quando chegamos ao vértice pela primeira vez), mas l é atualizado de acordo com os valores de l para os vizinhos do vértice.

A função *lowlink*

Se o vizinho v do vértice atual u não foi visitado ainda, fazemos a chamada recursiva e:

- Se $t_u \leq l_v$, não pode ser alcançado a partir de v nenhum vértice que veio antes de u **(ponto de articulação)**
- Se $t_u < l_v$, não pode ser alcançado a partir de v nenhum vértice que veio antes de u nem o próprio u **(ponte)**
- $l_u \leftarrow \min(l_u, l_v)$

Caso contrário, basta fazer $l_u \leftarrow \min(l_u, t_v)$

O algoritmo de Tarjan

Com a função *lowpoint* podemos fazer um algoritmo de componentes conexos:

- Note que, para toda fonte u, $l_u = t_u$.
- Basta então manter uma pilha dos vértices recentemente visitados que é atualizada no **início** da DFS.
- Quando $l_u = t_u$, o componente fortemente conexo será composto por todos os vértices que aparecem depois de u na pilha.

Implementação do Tarjan

Código `tarjan.h`

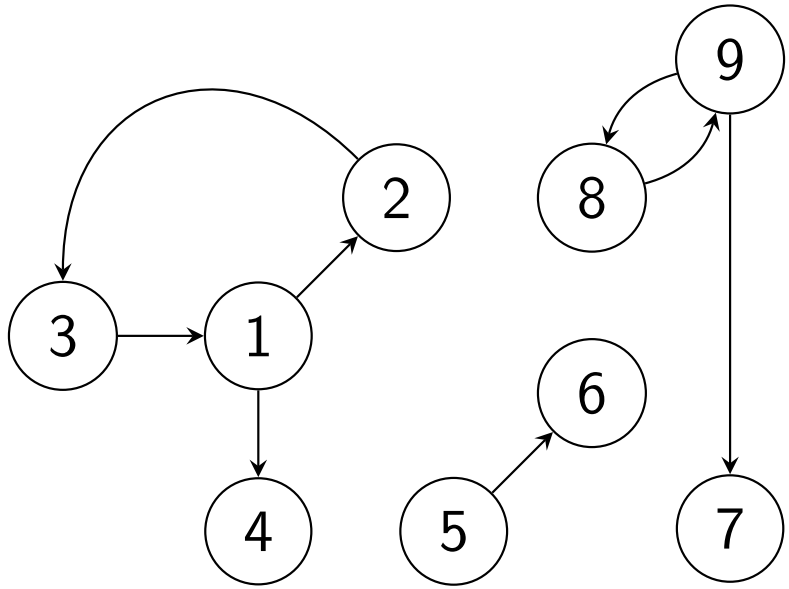
```
int tk = 1; vector<int> tin (N, -1), low (N);
stack<int> st; vector<int> rep (N), onst (N);
void dfs_tarjan(int u) {
    if (tin[u] != -1) { return; }
    tin[u] = low[u] = onst[u] = ++tk;
    st.push(u);
    for (int v : g[u]) {
        dfs_tarjan(v);
        if (onst[v]) low[u] = min(low[u], low[v]);
    }
    if (low[u] == tin[u]) {
        int v; do {
            v = st.top(); st.pop(); onst[v] = 0;
            rep[v] = u;
        } while (u != v);
    }
}
void tarjan(int n) {
    for (int u = 0; u < n; u++) { dfs_tarjan(u); }
}
```

Implementação do Tarjan (continuado)

Código `tarjan.cpp`

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15; int cts = 1;
vector<vector<int>> g (N);
#include "tarjan.h"
int main() { int n, m; cin >> n >> m;
    while (m--) { int u, v; cin >> u >> v; u--; v--;
        g[u].push_back(v); }
    tarjan(n);
    for (int u = 0; u < n; u++) cout << rep[u]+1 << " ";
}
```

Implementação do Tarjan (continuado)



Entrada	Saída
9 8 1 2 2 3 3 1 1 4 8 9 9 8 5 6 9 7	1 1 1 4 5 6 7 8 8

Grafo condensado

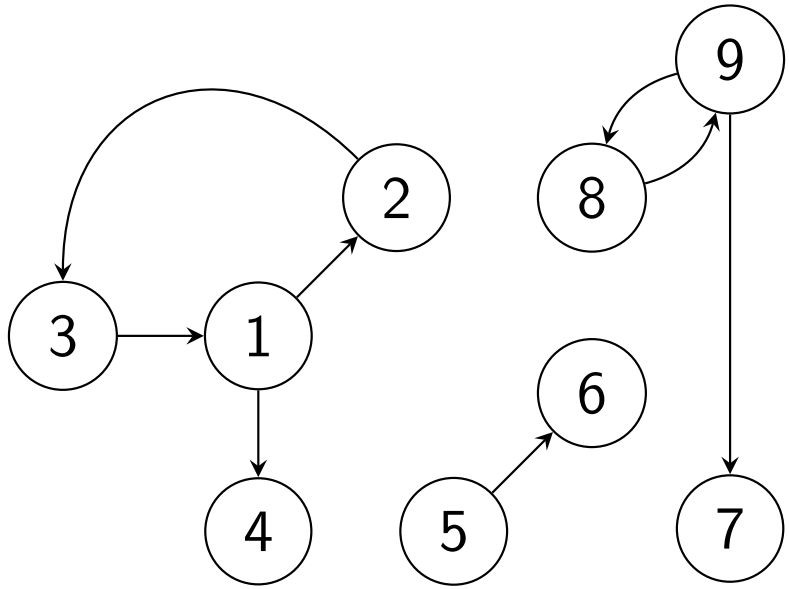
- Por vezes é útil pegar um grafo direcionado com ciclos e transformá-lo em um DAG correspondente.
- Podemos fazer isso criando um **grafo condensado**.
- Cada vértice representa uma componente fortemente conexa.
- Assim, podemos aplicar soluções de DAG em grafos cíclicos!
- Vamos fazer isso pegando cada um dos vértices e vendo se os seus vizinhos pertencem a componentes diferentes.
- Se sim, criamos uma aresta entre as duas componentes.
- Cuidado que no final das contas você poderá obter um multigrafo com várias arestas repetidas.
- Isso geralmente não é problema, mas é melhor evitar o ganho de complexidade tirando arestas duplicadas.

Implementação de grafo condensado

Código condensed.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15; vector<int> vis (N); int cts = 1;
vector<vector<int>> g (N);
vector<set<int>> cg (N);
#include "tarjan.h"
int main() { int n, m; cin >> n >> m;
    while (m--) { int u, v; cin >> u >> v; u--; v--;
        g[u].push_back(v); }
    tarjan(n);
    for (int u = 0; u < n; u++)
        for (int v : g[u]) if (rep[u] != rep[v])
            cg[rep[u]].insert(rep[v]);
    for (int u = 0; u < n; u++) if (rep[u] == u) {
        for (int v : cg[u])
            cout << u+1 << " " << v+1 << "\n";
    }
}
```

Implementação do grafo condensado (continuado)



Entrada	Saída
9 8 1 2 2 3 3 1 1 4 8 9 9 8 5 6 9 7	1 4 5 6 8 7

Pontos de Articulação e Pontes

Definições

- **Articulação:** Vértice v em um grafo conectado G tal que, se o vértice v fosse removido, G não seria conectado.
- **Ponte:** Aresta e em um grafo conectado G tal que, se a aresta e fosse removida, G não seria conectado.

Implementação de articulações e pontes

Código `articbridges.h`

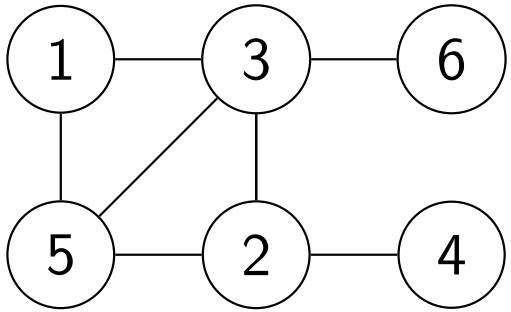
```
using ii = pair<int, int>;
int tk = 0; vector<int> tin (N, -1), low (N);
vector<ii> brid; set<int> arti;
void dfs(int u, int p) {
    tin[u] = low[u] = ++tk; int ch = 0;
    for (int v : g[u]) if (v != p) {
        if (tin[v] == -1) {
            dfs(v, u); ch++;
            if ((low[v] >= tin[u] && p != u) ||
                (ch >= 2 && p == u))
                arti.insert(u);
            if (low[v] > tin[u])
                brid.push_back(ii(u, v));
            low[u] = min(low[u], low[v]);
        } else { low[u] = min(low[u], tin[v]); }
    }
}
```

Utilizando articulações e pontes

Código articbridges.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+15;
vector<vector<int>> g (N);
#include "articbridges.h"
int main() {
    int n, m; cin >> n >> m;
    while (m--) {
        int u, v; cin >> u >> v; u--; v--;
        g[u].push_back(v); g[v].push_back(u);
    }
    for (int u = 0; u < n; u++) { dfs(u, u); }
    cout << "articulations:\n";
    for (int u : arti) { cout << u+1 << " "; }
    cout << "\n" << "bridges:\n";
    for (auto [u, v] : brid)
        cout << u+1 << " " << v+1 << "\n";
}
```

Utilizando articulações e pontes (continuado)



Entrada	Saída
6 8 1 3 1 5 2 3 2 4 2 5 3 2 3 5 3 6	articulations: 2 3 bridges: 2 4 3 6