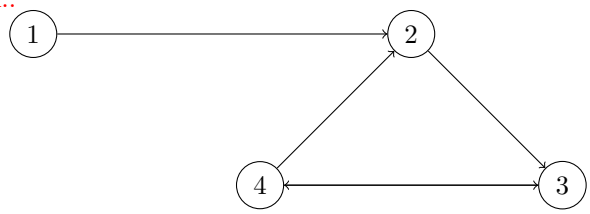
Programação Competitiva

Isaias Castro

Maio 2025

1 Componentes Fortemente Conexos

Um grafo dirigido (digrafo) é constituido de vértices conectados por arcos.
Ex.:



Um caminho dirigido é uma sequência alternada entre vértices e arcos, que começa e termina num vértice. Além disso, todos os vértices tem que ser distintos, exceto, possivelmente, o primeiro e o último vértice (se eles forem iguais teremos um caminho dirigido fechado também chamado de circuito dirigido). Além disso, se $\vec{a}$ faz parte do caminho, então $\vec{a}$ deve ter como origem o vértice que o antecede no caminho e como destino o vértice que o sucede no caminho.

Um digrafo é dito fortemente conexo se para cada par (u,v) de vértices do digrafo existe um caminho de u para v. Uma componente fortemente conexa de um digrafo $\vec{G}$ é um subgrafo de G que é fortemente conexo e é maximal. O algoritmo de Kosaraju permite encontrar as componentes conexas de um digrafo.

1.1 Algoritmo de Kosaraju

1. Dado um digrafo $\vec{G}$, calcule o grafo reverso $\vec{G}^R$ invertendo o sentido de cada arco de $\vec{G}$
2. Faça uma DFS (Depth-First-Search) em $\vec{G}^R$, obtendo uma ordenação $v_1, v_2, ..., v_n$ em pós-ordem dos vértices de $\vec{G}^R$.
3. Faça uma DFS em $\vec{G}$ percorrendo os vértices na ordem $v_n, v_{n-1}, ..., v_1$. Em cada etapa da DFS, o conjunto dos vértices percorridos constituem uma componente conexa de $\vec{G}$.

OBS: uma ordenação em pós-ordem ocorre quando o nó só é inserido na ordenação após todos os nós que dependem dele já tiverem sido inseridos, ou seja, todos os nós alcançados pelos arcos que saem do vértice já foram inseridos.

1.2 Implementação

1. Primeiro vamos calcular o grafo $\vec{G^R}$ percorrendo cada aresta e invertendo-a.

```
vector<vector<int>> GReverse(vector<vector<int>>& G){
    vector<vector<int>> _G (G.size());

    for(int i = 0; i < G.size(); ++i){
        for(int j = 0; j < G[i].size(); ++j){
            _G[G[i][j]].push_back(i);
        }
    }

    return _G;
}
```

2. Agora já com nosso grafo $\vec{G^R}$, vamos fazer uma DFS para encontrar uma ordenação em pós-ordem grafo.

```
void DFS_PosOrder(int u, vector<vector<int>>& _G, vector<bool>& vis, vector<int>& order) {
    vis[u] = true;

    for (int viz : _G[u]) {
        if (!vis[viz]) {
            DFS_PosOrder(viz, _G, vis, order);
        }
    }

    order.push_back(u);
}
```

nesse caso teremos uma complementação na função **main** para percorremos todo o grafo _G que consistem em apenas um laço for para iterar os vértices.

```
for(int i = 0; i < V; ++i){
    if(!vis[i]) DFS_PosOrder(i, _G, vis, s);
}
```

3. Por fim, a parte mais complexa, faremos uma DFS em G na ordem reversa da ordenação em pós-ordem achada e contar a quantidade de subgrafos fortemente conexos nos temos.

```
int Counter(vector<vector<int>>& G, vector<int>& stackAux){
    set<int> vis;
    int count = 0;

    for(int i = stackAux.size() - 1; i >= 0; --i){
        int v = stackAux[i];
        if(vis.find(v) == vis.end()) {
            count++;
            stack<int> s;
            s.push(v);
            vis.insert(v);

            while(!s.empty()){
                int currVertice = s.top();
                s.pop();

                for(int adj : G[currVertice]){
                    if(vis.find(adj) == vis.end()){
                        s.push(adj);
                        vis.insert(adj);
                    }
                }
            }
        }
    }

    return count;
}
```

4. Como nesse caso a função **main** contêm parte da solução do algoritmo e não apenas código de entrada e saída, ela será mostrada para compreensão total do algoritmo.

```cpp
int main(){
    int V, E;
    cin >> V >> E;

    vector<vector<int>> G (V);

    for(int i = 0; i < E; ++i){
        int a, b;
        cin >> a >> b;

        G[a-1].push_back(b-1);
    }

    vector<vector<int>> _G = GReverse(G);

    vector<bool> vis(V, false);
    vector<int> s;

    for(int i = 0; i < V; ++i){
        if(!vis[i]) DFS_PosOrder(i, _G, vis, s);
    }

    int count = Counter(G, s);

    cout << count << "\n";
    cout << ((count == 1) ? "Yes\n" : "No\n");

    return 0;
}
```