Lista 3

Universidade Federal de Minas Gerais

Departamento de Computação

Projeto e Análise de Algoritmos - 2024.2

Professor: Marcio Costa Santos

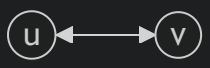
Exercício 1. Mostre que dado um grafo não orientado $G = (V, E)$ temos que $\sum_{v \in V} d(v) = 2|E(G)|$

Como por definição uma aresta (e) liga dois vértices (u, v), se utilizarmos uma função $split(e)$ que divide a aresta e nos dois vértices que ela liga, para cada aresta, teremos dois vértices.

Ao fazermos essa divisão para todas as arestas, teremos que a quantidade final de ocorrências de vértices (mesmo repetidos) será $2|E(G)|$.

Nessa concepção, a contagem de ocorrências de um determinado vértice nessa lista gerada por $split(e)$ será igual ao grau do vértice. Sendo assim, ao fazermos a soma de todos os graus dos vértices, teremos a quantidade de ocorrências de vértices na lista, que é $2|E(G)|$.

Prova por indução



- **Base:** $d(u) + d(v) = 2 \cdot 1 = 2$
- **Hipótese:** $\sum_{v \in V} d(v) = 2|E|$
- **Passo:** $G + u'k' = 2 \cdot |E| + 2 = 2(|E| + 1)$

Exercício 2. Prove que uma árvore com exatamente dois vértices de grau 1 é um caminho

---

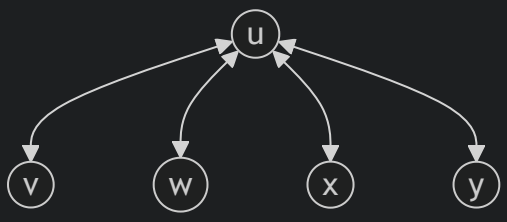
Como por definição, um caminho terá uma sequência de arestas tal que cada vértice tem grau 1 caso seja extremidade do caminho e grau 2 caso seja um vértice intermediário. Em um caminho trivial constituído por dois vértices, ambos terão grau 1, pois serão extremidades do caminho. E esse grafo, é, por definição, uma árvore.

~~Exercício 3. Prove que uma árvore $T = (V, E)$ com $\Delta(T) \geq k$ possui pelo menos k vértices~~

Exercício 3. Prove que uma árvore $T = (V, E)$ com $\Delta(T) \geq k$ possui pelo menos $k + 1$ vértices

---

- $\Delta(T)$ é o grau máximo de um vértice em T



Exercício 4. Definimos o complemento de um grafo simples $G = (V, E)$ não orientado, como o grafo $\overline{G} = (V, \overline{E})$ onde

- $V(\overline{G}) = V(G)$
- $E(\overline{G}) = \{uv | uv \neq E(G), u \neq v\}$

Apresente um algoritmo para essa operação e calcule a sua complexidade em cada uma das duas representações estudadas de um grafo.

L3Q4 - Matriz de Adjacência

```python
def complement_matrix(G):
    id = get_id_matrix(len(G)) # Filled with 0, but with 1 in the diagonal
    full_matrix = get_full_matrix(len(G)) # Filled with 1
    complement_matrix = get_zeroed_matrix(len(G)) # Filled with 0
    for i in range(len(G)):
        for j in range(len(G)):
            complement_matrix[i][j] = full_matrix[i][j] - G[i][j] - id[i][j]
    return complement_matrix
```

- **Complexidade:** $O(n^2)$

L3Q4 - Lista de Adjacência

```python
# G = {
#     0: [1, 2],
#     1: [0, 2],
#     2: [0, 1]
# }

def get_complement_list(G):
    complement_list = []
    for v in G.keys():
        new_adj_list = []
        for u in G[v]:
            if v != u and u not in G[v]:
                new_adj_list.append(u)
        complement_list.append(new_adj_list)
    return complement_list
```

- $X =$ complexidade de checagens de pertencimento ($\in$)
- **Complexidade:** $O(n^2 + X)$

```python
# Considere que cada par de vértices tem no máximo uma aresta
def get_complement_list(G):
    complement_list = {}
    vertices = set(G.keys())
    for v in vertices:
        adj_v = set(G[v])
        new_adj_list = []
        for u in adj_v:
            complement_set = vertices - adj_v - set(v)
        complement_list[v] = complement_set
    return complement_list
```

- $Y =$ complexidade de subtração de sets $(set(a) - set(b))$
- **Complexidade:** $O(n^2 + Y)$

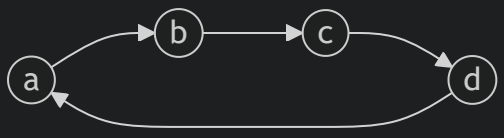
Exercício 5. Definimos a transposição de um grafo simples $G = (V, A)$ orientado, como o grafo $G^T = (V, A^T)$ onde

- $V(G^T) = V(G)$
- $E(G^T) =$ aos arcos de G com sentido trocado

Apresente um algoritmo para essa operação e calcule a sua complexidade em cada uma das duas representações estudadas de um grafo.

L3Q5 - Grafos

L3Q5 - Grafo $G = (V, A)$



L3Q5 - Grafo Transposto $G^T = (V, A^T)$

L3Q5 - Matriz de Adjacência

$$G_M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cong \begin{bmatrix} G_M & a & b & c & d \\ a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 1 & 0 \\ c & 0 & 0 & 0 & 1 \\ d & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$T(G_M) = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```python
def get_transposed_matrix(G):
    transpose_matrix = get_zeroed_matrix(len(G))
    for i in range(len(G)):
        for j in range(len(G)):
            transpose_matrix[i][j] = G[j][i]
    return transpose_matrix
```

- **Complexidade:** $O(n^2)$

L3Q5 - Lista de Adjacência

$$G_L = \begin{cases} a : [b] \\ b : [c] \\ c : [d] \\ d : [a] \end{cases}$$

$$T(G_L) = \begin{cases} a : [d] \\ b : [a] \\ c : [b] \\ d : [c] \end{cases}$$

```python
# Considere que cada par de vértices tem no máximo uma aresta
def get_transposed_list(G):
    transposed_list = [len(G) * []]
    for v in range(len(G)):
        for u in G[v]:
            transposed_list[u].append(v)
    return transposed_list
```

Exercício 6. Definimos o quadrado de um grafo simples $G = (V, A)$ não orientado, como o grafo

$G^2 = (V, A^2)$ onde

- $V(G^2) = V(G)$
- $E(G^2) = uv | \exists z \in V(G), z \neq u \neq v, uz \in E(G), vz \in E(G)$

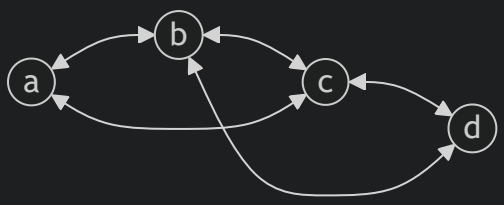
Apresente um algoritmo para essa operação e calcule a sua complexidade em cada uma das duas representações estudadas de um grafo.

Serão 3 loops. O primeiro para percorrer todos os vértices (v). O segundo para obter os vizinhos desse vértice ($Viz(v)$). O terceiro para encontrar os vizinhos dos vizinhos.

L3Q6 - Grafos

L3Q6 - Grafo $G = (V, A)$

Grafo Quadrado $G^2 = (V, A^2)$



L3Q6 - Matriz de Adjacência

$$G_M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$G_M^2 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

```
def get_square_matrix(G):
    GG = G
    for i in range(len(G)):
        for j in range(len(G)):
            if G[i][j] == 1:
                for k in range(len(G)):
                    if G[j][k] == 1:
                        GG[i][k] = 1
    return square_matrix
```

- **Complexidade:** $O(n^3)$

L3Q6 - Lista de Adjacência

$$G_L = \begin{cases} a : [b] \\ b : [a, c] \\ c : [b, d] \\ d : [c] \end{cases}$$

$$G_L^2 = \begin{cases} a : [b, c] \\ b : [a, c, d] \\ c : [b, d, a] \\ d : [c, b] \end{cases}$$

```
def get_square_list(G):
    GG = G                                  # O(n^2)
    for v in range(len(G)):                 # O(V)
        for u in G[v]:                      # O(\Delta(d))
            for k in G[u]:                  # O(\Delta(d))
                if k != v and k not in G[v]: # O(\Delta(d))
                    GG[v].append(k)         # O(1)
    return GG
```

- **Complexidade:** $O(n^4)$
  - $O(n^2) + O(V) \cdot O(\Delta(d)) \cdot O(\Delta(d)) \cdot O(\Delta(d)) \cdot O(1)$
  - $O(n^2) + O(V) \cdot O(V) \cdot O(V) \cdot O(V) \cdot O(1)$

- $O(n^2) + O(n^4)$
- $O(n^4)$

Exercício 7. Descreva como podemos modificar as duas estruturas de representação de grafo para acomodar grafos ponderados (grafos que possuem valores numéricos associados com suas arestas ou arcos)

L3Q7 - Matriz de Adjacência

Para a matriz de adjacência, onde temos valor inteiros positivos, podemos adicionar o peso da aresta na posição M_{ij} da matriz.

Se permitirmos valores negativos, podemos adicionar um valor que represente a ausência de aresta, como $NULL$ ou $False$.

Uma outra abordagem seria adicionar uma nova dimensão na matriz, onde indicaríamos que no índice M_{ij0} teríamos um valor booleano que indicaria se a aresta existe ou não, e no índice M_{ij1} teríamos o valor do peso da aresta.

L3Q7 - Lista de Adjacência

Para a lista de adjacência, podemos criar uma tupla que represente a aresta, onde o primeiro valor seria o vértice de destino e o segundo valor seria o peso da aresta.

Poderíamos também criar uma lista de adjacência complementar, onde os valores nos mesmos índices da lista de adjacência original seriam os pesos das arestas.

Exercício 8. Mostre que uma aresta uv é

1. uma aresta de árvore ou de avanço se e somente se $i[u] < i[v] < f[v] < f[u]$;

2. uma aresta de retorno se e somente se $i[v] \leq i[u] < f[u] \leq f[v]$; e
3. uma aresta de passagem se e somente se $i[v] < f[v] < i[u] < f[u]$.

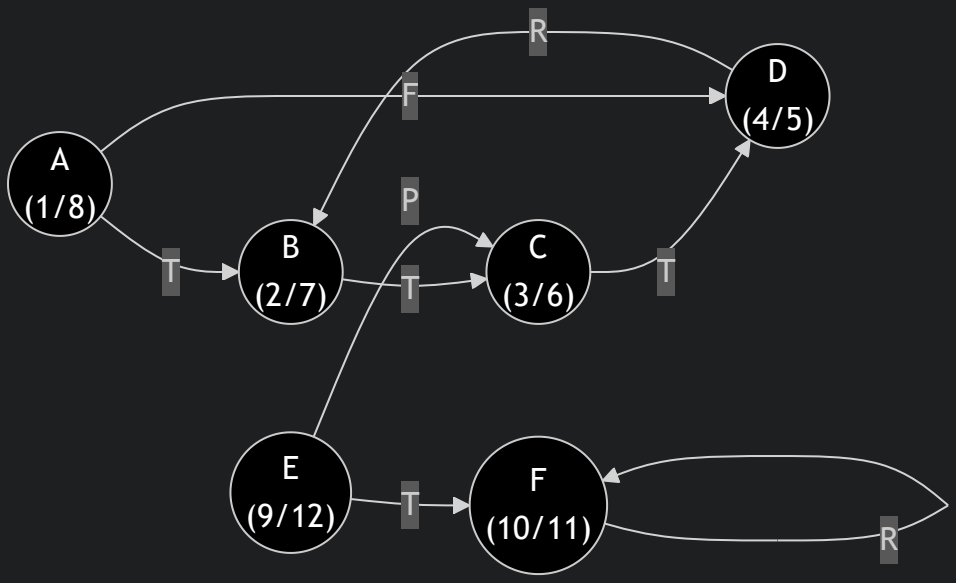
[JV: Eles fizeram, mas eu me perdi fazendo a pintura do grafo com Mermaid]

Pintura do grafo

- Cores:
  - Branco: `#fff`
  - Cinza: `#999`
  - Preto: `#000`
- Arestas
  - T: Árvore || Tree: are edges in depth-first forest G_π. $Edge(u, v)$ is a tree edge if v was first discovered by exploring edge (u, v).
  - R: Retorno || Back: are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
  - F: Avanço || Forward: are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
  - P: Passagem || Cross: are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

▶ (1)
▶ (1 (2))
▶ (1 (2 (3)))
▶ (1 (2 (3 (4))))
▶ (1 (2 (3 (4 (5)))))
▶ (1 (2 (3 (4 (5 6)))))
▶ (1 (2 (3 (4 (5 6) 7))))
▶ (1 (2 (3 (4 (5 6) 7) 8)))
▶ (1 (2 (3 (4 (5 6) 7) 8) (9)))
▶ (1 (2 (3 (4 (5 6) 7) 8) (9)))
▶ (1 (2 (3 (4 (5 6) 7) 8) (9 (10)))
▶ (1 (2 (3 (4 (5 6) 7) 8) (9 (10)))
▶ (1 (2 (3 (4 (5 6) 7) 8) (9 (10 11))
▶ (1 (2 (3 (4 (5 6) 7) 8) (9 (10 11) 12)

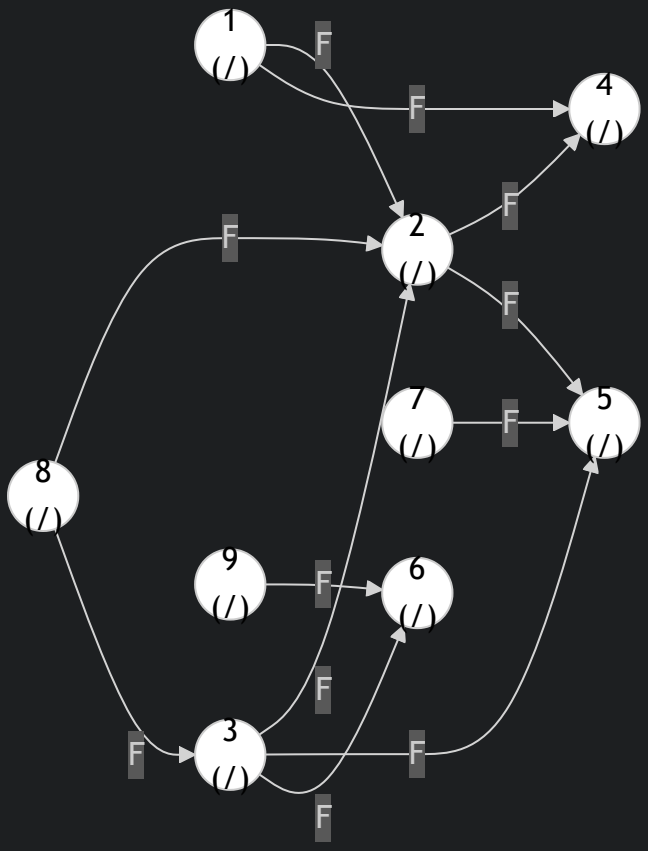
Grafo Final: (1 (2 (3 (4 (5 6) 7) 8) (9 (10 11) 12)



Exercício 9. Na descrição da busca em profundidade vista em sala classificamos as arestas do grafo de entrada de acordo após a execução da busca em profundidade. É possível realizar essa classificação durante a execução da busca? Como?

[JV: Eles fizeram, mas eu me perdi fazendo a pintura do grafo com Mermaid]

Exercício 10. Aplique o algoritmo de busca em profundidade para o grafo abaixo

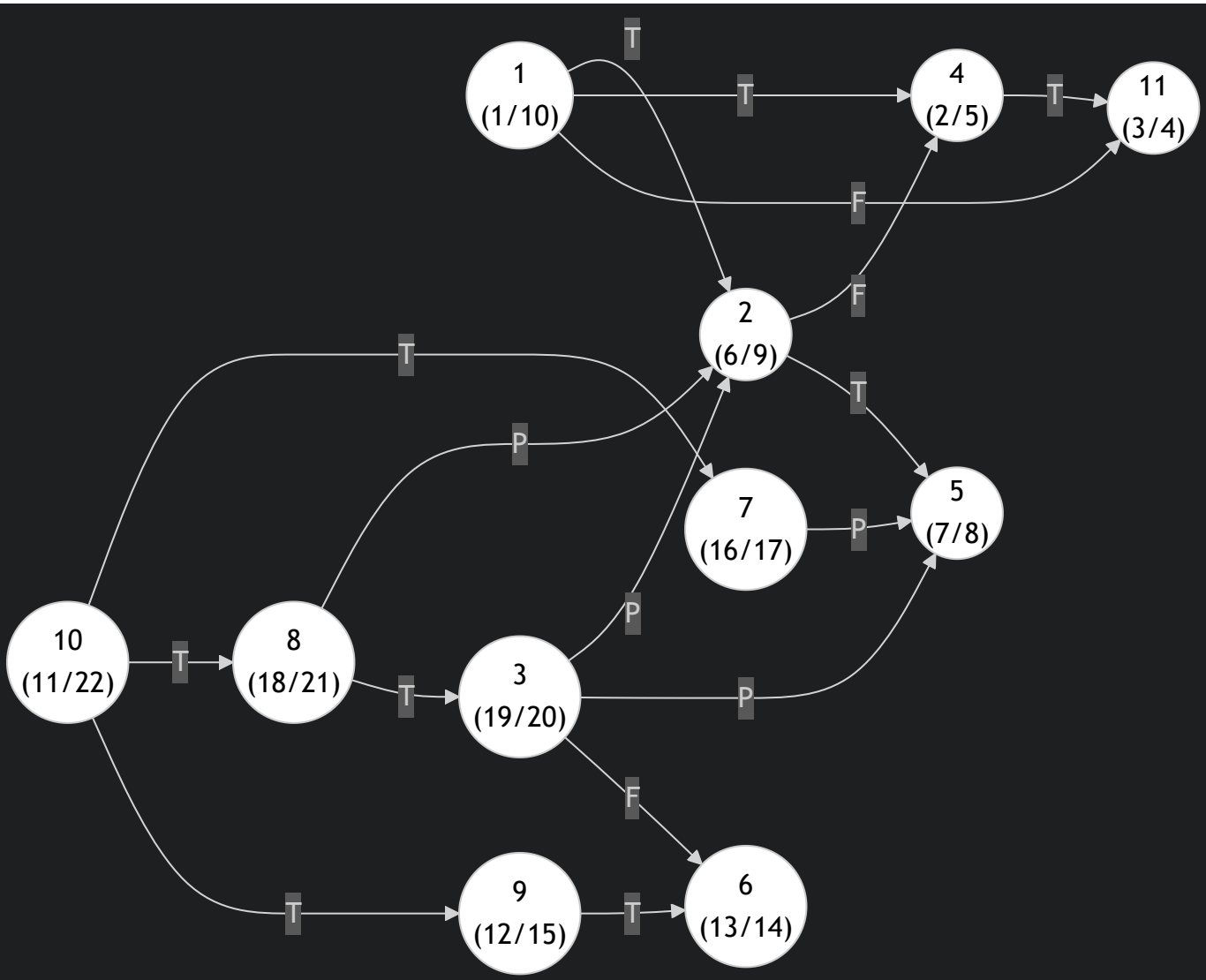


Exercício 11. Apresente uma versão não recursiva para o algoritmo de busca em profundidade que possua a mesma complexidade da versão recursiva

```python
def dfs_iterative(G, v): # DFS usa pilha || BFS usa fila
    visited = []
    marked = [False] * len(G)
    stack = [v] # Pilha || queue = [v] # Fila
    while len(stack) > 0:
        v = stack.pop() # Pilha || v = queue.pop(0) # Fila
        if not marked[v]:
            visited.append(v)
            marked[v] = True
            for w in Vizinhos(v):
                if not marked[w]:
                    stack.append(w) # Pilha || queue.append(w) # Fila
    return visited
```

Exercício 12. Aplique o algoritmo de ordenação topológica no DAG abaixo

---



Ordenação topológica: Ordenar os vértices em ordem decrescente do tempo de finalização. Ou seja, inverte a lista.

DFS = [1, 4, 11, 2, 5, 9, 6, 7, 8, 3, 10]
DFS invertida = [10, 3, 8, 7, 6, 9, 5, 2, 11, 4, 1]

Exercício 13. O que acontece quando utilizamos o algoritmo de ordenação topológica em um grafo orientado que contém ciclos? A ordenação obtida faz sentido?

A ordenação topológica representa a ordem de execução de tarefas, onde uma tarefa só pode ser executada após a execução de outra tarefa. Se houver um ciclo, a ordenação topológica não faz sentido, pois a tarefa A depende da tarefa B, que depende da tarefa C, que depende da tarefa A.

Uma ordenação topológica coerente resultaria em uma sequência de tarefas que podem ser executadas sem necessitar de outras tarefas que ainda não foram executadas.

Quando ocorre o ciclo, existirá uma tarefa que depende que uma tarefa ainda não executada. Dessa forma, mesmo que o algoritmo resulte em uma ordenação topológica, ela não faz sentido.

Exercício 14. Apresente um algoritmo que determina se um grafo não direcionado possui ou não um ciclo. Qual a complexidade desse algoritmo?

```python
# O algoritmo deve rodar uma DFS e, quando uma aresta de retorno (um nó cinza encontrar outro n
def has_cycle(G):
    # [JV: Não verifiquei esse código. O Copilot que cuspiu isso pra mim]
    def dfs(G, v, parent, visited):
        visited[v] = True
        for w in G[v]:
            if not visited[w]:
                if dfs(w, v):
                    return True
            elif w != parent:
                return True
        return False

    visited = [False] * len(G)
    for v in range(len(G)):
        if not visited[v]:
            if dfs(G, v, -1, visited):
                return True
    return False
```

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

Exercício 15. É possível fazer o algoritmo da questão anterior executar em $O(|V|)$ para toda entrada? Se sim, como?

[JV: Segundo eles...]

Sim. Usando Union-Find.

[JV: Segundo Copilot...]

Sim, é possível. Basta adicionar um vetor de pais para cada vértice. Quando um vértice for visitado, o pai do vértice será atualizado. Se um vértice cinza encontrar outro vértice cinza, basta verificar se o vértice encontrado é o pai do vértice atual. Se for, o grafo não possui ciclo. Se não for, o grafo possui ciclo.

Exercício 16. Como o número de componentes fortemente conexas de um grafo pode mudar pela adição de um arco?

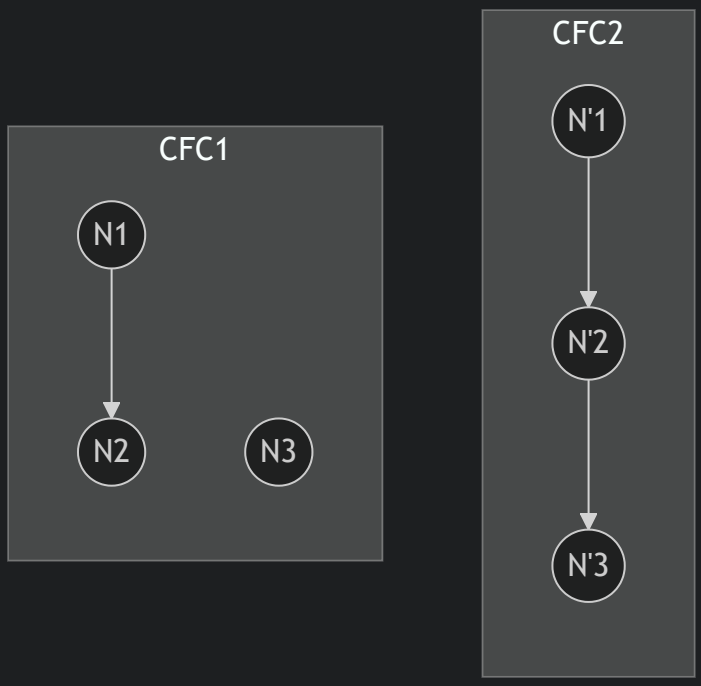
A partir da adição de um arco, o número de componentes fortemente conexas pode aumentar, se manter ou diminuir.

Aumentando: Se o arco adicionado pode conectar dois nós que não estavam conectados anteriormente, assim criando uma nova componente fortemente conexa.

Exemplo:

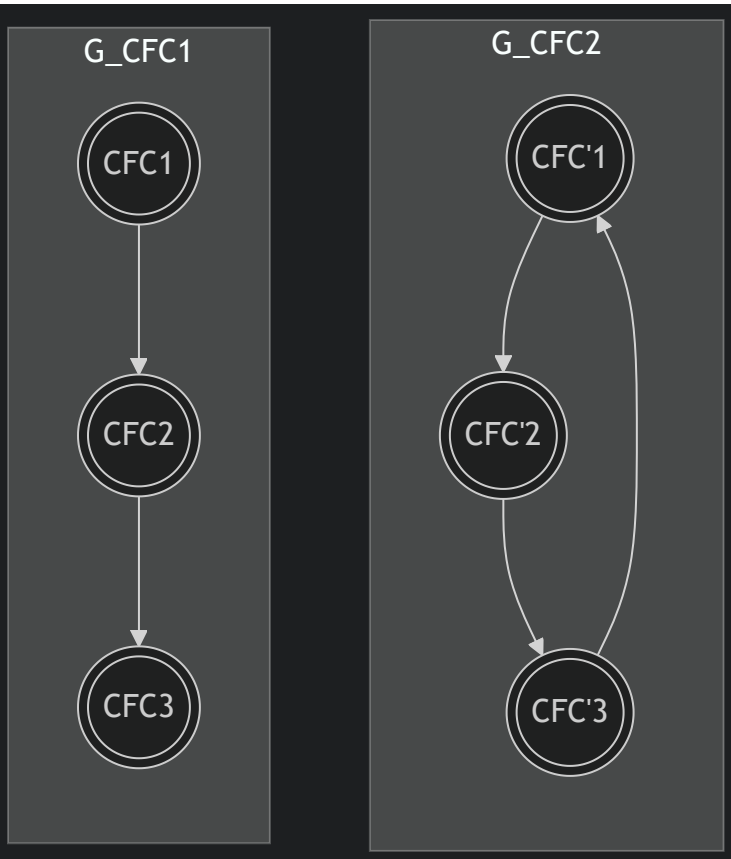
Mantendo: Se o arco adicionado não alterar a conexão entre os nós, mantendo as componentes fortemente conexas.

Exemplo:



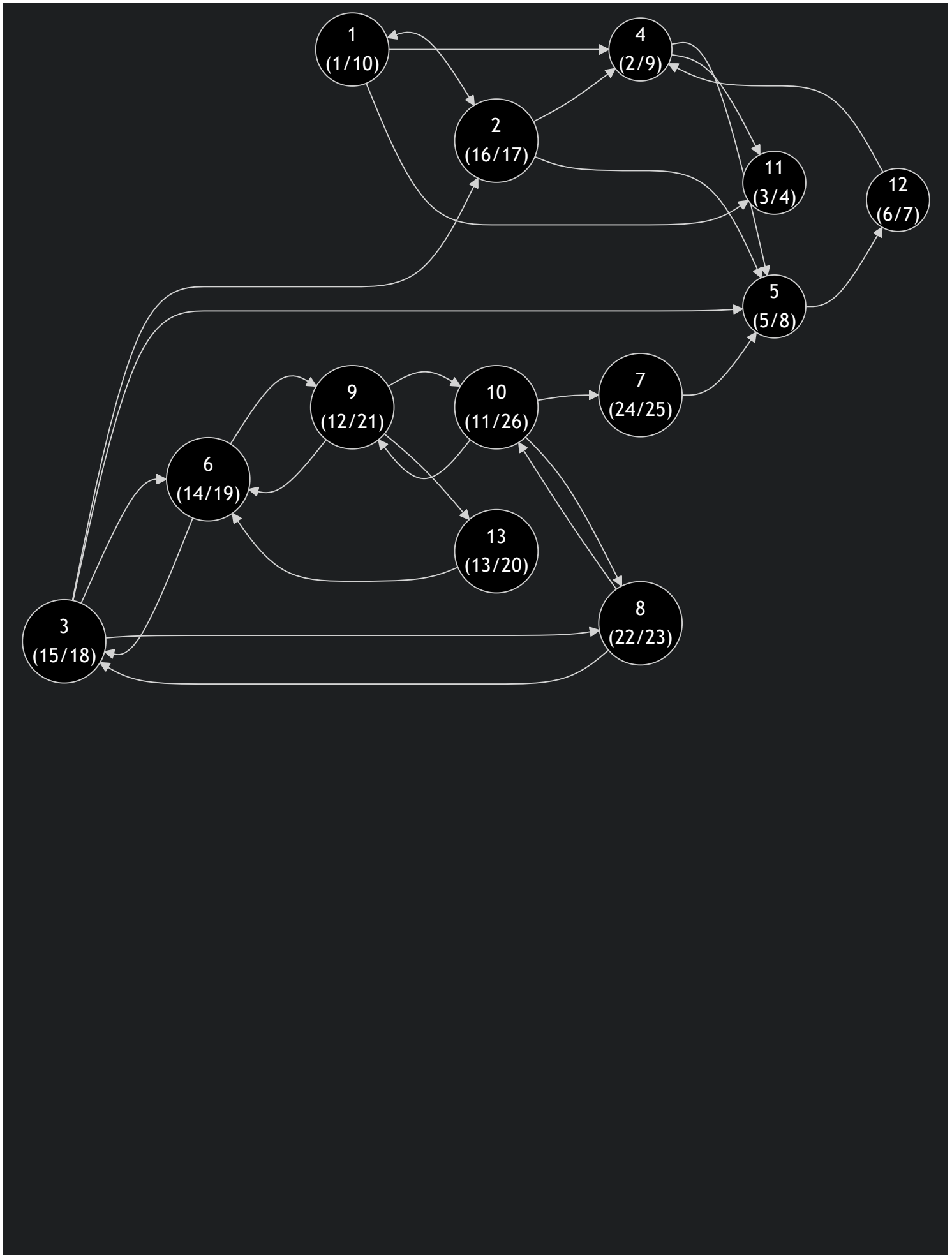
Diminuindo: Se o arco adicionado pode desconectar dois nós que estavam conectados anteriormente, assim diminuindo o número de componentes fortemente conexas.

Exemplo:



Exercício 17. Aplique o algoritmo para determinar componentes fortemente conexas no grafo orientado abaixo

[JV: não prestei muita atenção nisso]


1
(1/10)
4
(2/9)
2
(16/17)
11
(3/4)
12
(6/7)
5
(5/8)
9
(12/21)
10
(11/26)
7
(24/25)
6
(14/19)
13
(13/20)
3
(15/18)
8
(22/23)