Exercício 1. *Mostre que dado um grafo não orientado $G = (V, E)$ temos que*

$$\sum_{v \in V(G)} d(v) = 2|E(G)|$$

O exercício pede para demonstrar que, em um grafo não orientado G=(V,E)G = (V, E)G=(V,E), a soma dos graus de todos os vértices é igual ao dobro do número de arestas, ou seja:

$$\sum_{v \in V(G)} d(v) = 2|E(G)|$$

Este resultado é conhecido como **Teorema da Mão Dupla** ou **Teorema do Aperto de Mãos**. Vamos demonstrar:

1. **Definição de grau de um vértice**: O graud(v) de um vértice v em um grafo não orientado é o número de arestas que incidem sobre ele.
2. **Soma dos graus**: Se somarmos o grau de todos os vértices do grafo G, cada aresta será contada duas vezes (uma vez para cada extremidade da aresta). Portanto, a soma total dos graus de todos os vértices é igual a duas vezes o número total de arestas $|E(G)|$.
3. **Demonstração formal**:
    - Cada aresta e={u,v} contribui com 1 ao grau do vértice u e 1 ao grau do vértice v.
    - Como cada aresta é contada duas vezes na soma dos graus, temos que: $\sum v \in V(G) d(v) = 2 \times |E(G)|$

Exemplo:

Considere o grafo GGG com V={A,B,C} e E={{A,B},{B,C}}:

- d(A)=1
- d(B)=2
- d(C)=1

Soma dos graus: 1+2+1=4

Número de arestas: $|E(G)|=2$

Portanto, $\sum v \in V(G) d(v) = 2|E(G)|$

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

Demonstração:
Definição de árvore:

- Uma **árvore** é um grafo conexo e acíclico.
- Em uma árvore com nnn vértices, existem exatamente n−1 arestas.

Definição de caminho:

- Um **caminho** é um grafo em que existe exatamente uma sequência de vértices v1,v2,…,vk conectados por arestas, sem formar ciclos e sem bifurcações.

Agora, provemos que uma árvore com exatamente dois vértices de grau 1 é um caminho.

Passo 1: Grau dos vértices em uma árvore

- Os **vértices de grau 1** em uma árvore são chamados de **folhas**.
- Qualquer árvore com n vértices e n−1 arestas precisa ter pelo menos **uma folha** (vértice de grau 1).

Passo 2: Configuração com dois vértices de grau 1

- Suponha que temos uma árvore com **dois vértices de grau 1**: u e v.
- Como a árvore é conexa e acíclica, os vértices u e v precisam estar conectados por uma sequência de vértices u=v1,v2,…,vk, formando um **caminho simples**.

Passo 3: Prova por contradição

- Suponha, por contradição, que a árvore **não é um caminho**. Então, a árvore deve conter uma **bifurcação** (um vértice com grau maior que 2).
- Se isso acontecer, haverá mais de duas folhas ou múltiplos ramos, o que viola a condição de que há **exatamente dois vértices de grau 1**.
- Isso contradiz a hipótese inicial.

Conclusão

- Portanto, uma árvore com **exatamente dois vértices de grau 1** deve ser **um caminho**, pois não existe outra forma de conectar os vértices sem introduzir bifurcações ou ciclos.

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

Definições Importantes:

1. **Árvore**:

    - Um grafo conexo e acíclico.
    - Se T tem n vértices, ele terá exatamente n−1 arestas.
2. **Grau de um vértice**:
    - O grau de um vértice v, denotado por d(v), é o número de arestas incidentes a v.
3. ∆**(T)**:
    - Δ(T) é o **grau máximo** entre todos os vértices da árvore T.
    - $\Delta(T) = \max\{d(v) : v \in V\}.$

Demonstração:

Queremos provar que se Δ(T)≥k, então a árvore T possui pelo menos k vértices.

1. **Árvore com grau máximo Δ(T) ≥ k**:
    - Por definição, existe pelo menos um vértice v∈V tal que d(v)≥k.
    - Esse vértice está conectado a pelo menos **k outros vértices**, pois seu grau é o número de arestas conectadas a ele.
2. **Número mínimo de vértices**:
    - O vértice v contribui com k conexões. Como cada uma dessas conexões é com um vértice diferente (pois não existem loops em uma árvore), temos no mínimo k vértices distintos:
        - O próprio vértice v, com grau d(v)≥k.
        - Os k vértices conectados a v.
3. **Conclusão**:
    - Uma árvore T com Δ(T)≥k deve ter pelo menos k vértices, pois um vértice com grau k ou mais necessita de pelo menos k vértices (o próprio vértice e k−1 vizinhos).

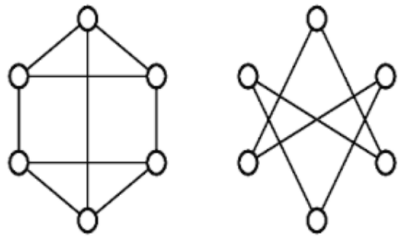
Exemplo:

- Considere uma árvore T com Δ(T)=3:
    - O vértice v com d(v)=3 está conectado a três outros vértices.
    - Assim, o número mínimo de vértices na árvore é 4 (v + 3 vértices conectados).

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

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

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



Definição do Complemento de um Grafo

Seja G=(V,E) um grafo simples e não orientado. O **complemento** de G, denotado como $\underline{G}$=(V,$\underline{E}$), é definido por:

- V($\underline{G}$)=V(G)(os vértices permanecem os mesmos).
- $\overline{E} = \{uv \mid uv \notin E(G), u \neq v\}$, ou seja, o conjunto de arestas do complemento contém todas as arestas que **não estão presentes em GGG**.

Objetivo:

1. Apresentar um algoritmo para construir o complemento de G.
2. Determinar a complexidade computacional do algoritmo considerando duas representações:
    - **Matriz de Adjacência**
    - **Lista de Adjacência**

---

1. Algoritmo para construir o complemento de um grafo

Entrada:

- Um grafo G=(V,E)G = (V, E)G=(V,E), representado por uma matriz ou lista de adjacência.

Saída:

- O grafo $\underline{G}$=(V,$\underline{E}$).

Algoritmo (usando matriz de adjacência):

Seja $n = |V|$, o número de vértices.

1. Crie uma matriz de adjacência M para G, onde M[i][j]=1 se existe uma aresta entre i e j, e 0 caso contrário.
2. Inicialize uma matriz de adjacência $\underline{M}$ para $\underline{G}$ com todos os valores iguais a 0.
3. Para cada par (u,v) com $1 \leq u,v \leq n$, e $u \neq v$:
    - Se M[u][v]=0, defina $\underline{M}^{-}[u][v]=1$.
    - Caso contrário, $\underline{M}[u][v]=0$.
4. Retorne $\underline{M}$.

Algoritmo (usando lista de adjacência):

1. Inicialize uma lista de adjacência L para G.
2. Para cada vértice $u \in V$, inicialize uma nova lista L[u].
3. Para cada vértice $v \in V$, com $v \neq u$:
    - Se $v \notin L[u]$, adicione v à lista L[u].
4. Retorne L

---

2. Complexidade do Algoritmo

a. Matriz de Adjacência

- A matriz de adjacência tem tamanho O(n^2), onde $n = |V|$.
- O algoritmo percorre todos os pares de vértices (u,v), verificando se existe uma aresta em G.
- **Complexidade**: O(n^2).

b. Lista de Adjacência

- Em uma lista de adjacência, o algoritmo percorre todos os vértices e, para cada vértice, verifica os $n-1$ possíveis vértices adjacentes.
- Verificar se $v \notin L[u]$ pode ser feito em O(deg(u)).
- No pior caso, percorremos todas as $n(n-1)/2$ arestas ausentes.
- **Complexidade**: O(n^2).

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.

Para um grafo orientado, a transposição deste se dá apenas pela inversão do sentido das arestas. Logo, em uma matriz de adjacência, basta transpor a matriz (inverter o índice i pelo j).

Exercício 6. *Definimos o quadrado de um grafo simples $G = (V, A)$ não orientado, como o grafo $G^2 = (V, A^2)$ onde*

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

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

O **quadrado de um grafo** G=(V,E), denotado por G^2. é definido como um grafo que possui os mesmos vértices de G, mas as arestas de G^2 conectam dois vértices u e v se:

1. Existe uma aresta uv∈E(G), ou
2. Existe um vértice w tal que uw∈E(G) e wv∈E(G). Em outras palavras, u e v estão a uma distância no máximo 2 em G.

Objetivo

1. Apresentar um algoritmo para calcular G^2.
2. Determinar a complexidade computacional do algoritmo para duas representações:
    - **Matriz de Adjacência**
    - **Lista de Adjacência**

---

1. Algoritmo para calcular G^2

Entrada:

- Um grafo G=(V,E), representado por uma matriz ou lista de adjacência.

Saída:

- O grafo G^2 = (V, E^2).

Algoritmo (Matriz de Adjacência)

1. Seja n=|V|.
2. Inicialize a matriz M de adjacência de G, onde M[u][v]=1 se uv∈E(G), e 0 caso contrário.
3. Crie uma matriz M^2 de tamanho n×n, inicialmente zerada.
4. Para cada par de vértices (u,v):
    - Se u≠v:
        - Se M[u][v]=1, então M^2[u][v]=1.
        - Caso contrário, verifique se existe algum vértice w tal que M[u][w]=1 e M[w][v]=1.
            - Se sim, defina M^2[u][v]=1.
5. Retorne M^2.

Algoritmo (Lista de Adjacência)

1. Inicialize a lista de adjacência L para G.
2. Crie uma nova lista de adjacência L^2 para G^2.
3. Para cada vértice u∈V:
    - Adicione a L^2[u] todos os vértices v∈L[u] (arestas diretas).
    - Para cada w∈L[u], adicione os vértices v∈L[w] em L^2[u], exceto o próprio u.
    - Garanta que não há duplicatas.
4. Retorne L^2.

---

2. Complexidade

a. Matriz de Adjacência

- O algoritmo percorre todos os pares de vértices (u,v) e, para cada par, verifica todos os n vértices intermediários w.
- **Complexidade**: O(n^3), onde nnn é o número de vértices.

b. Lista de Adjacência

- Para cada vértice u, percorremos seus vizinhos w (complexidade O(deg(u)) e, para cada w, percorremos seus vizinhos v (complexidade O(deg(w)).
- No pior caso, para grafos densos, a complexidade também pode atingir O(n^3), mas para grafos esparsos, a complexidade é aproximadamente proporcional ao número total de arestas e graus dos vértices:

- **Complexidade**: O(n+m · Δ(G)), onde m é o número de arestas e Δ(G) é o grau máximo.

Resumo

- **Matriz de Adjacência**: O(n^3).
- **Lista de Adjacência**: O(n+m · Δ(G)), com pior caso O(n^3).

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).*

Matriz de adjacência: é mais fácil de se colocar os pesos. Ao invés de possuir o valor 1 quando o vértice i se interliga ao vértice j, e zero quando não se interliga, o valor 1 é substituído pelo peso da aresta.
Lista de adjacência: ao invés de apenas relacionar um único valor atrelado ao vértice v que se interliga ao vértice u, colocar um par de valores, o vértice v e o peso w interligado à u.

Exercício 8. *Mostre que uma aresta uv é*

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

ii. uma aresta de retorno se e somente se $i[v] \leq i[u] < f[u] \leq f[v]$; *e*

iii. uma aresta de passagem se e somente se $i[v] < f[v] < i[u] < f[u]$.

Teorema dos parêntesis - tempo inicial e tempo final

- consideremos i de um vértice como o tempo inicial, ou o tempo em que o vértice é encontrado pelo algoritmo de busca, e f o tempo final, ou o tempo em que se "finalizou" este vértice, já que se explorou todas as arestas adjacentes. Se o tempo em que encontramos u é menor que o tempo em que encontramos v, significa que u aparece primeiro que v.

Item (i): Aresta de árvore ou de avanço

Uma aresta uv é de árvore ou avanço se, e somente se:

i[u]<i[v]<f[v]<f[u]

Prova:

- i[u]<i[v]: O vértice v é descoberto após u, pois u descobriu v como parte do DFS.
- i[v]<f[v]: Após a descoberta de v, todas as suas arestas adjacentes são exploradas antes que ele seja finalizado.
- f[v]<f[u]: O vértice v está dentro do tempo de processamento de u, o que é esperado, pois u é o ancestral no DFS.

Portanto, as condições para i[u]<i[v]<f[v]<f[u] descrevem corretamente uma aresta de **árvore ou avanço**.

Item (ii): Aresta de retorno

Uma aresta uv é de retorno se, e somente se:

i[v]≤i[u]<f[u]≤f[v].**Prova:**

- i[v]≤i[u]: O vértice v foi descoberto antes ou no mesmo momento em que u.
- i[u]<f[u]: O vértice u ainda está sendo processado quando uv é explorada.
- f[u]≤f[v]: O vértice u termina antes de v, pois v é um ancestral de u (e o DFS finaliza vértices em ordem inversa à descoberta no ciclo).

Essas condições descrevem uma aresta que retorna a um vértice ancestral, confirmando que ela é uma **aresta de retorno**.

Item (iii): Aresta de passagem

Uma aresta uv é de passagem se, e somente se:

i[v]<f[v]<i[u]<f[u].Prova:

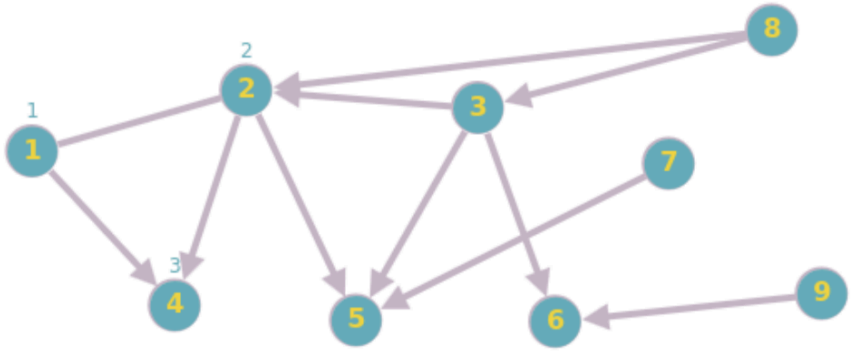
- i[v]<f[v]: O vértice v foi descoberto e finalizado antes da descoberta de u.
- f[v]<i[u]: O vértice v já foi completamente processado antes que u fosse descoberto.
- i[u]<f[u]: u segue o processo normal de descoberta e finalização após v.

Essas condições descrevem uma conexão entre componentes diferentes (em grafos direcionados) ou uma conexão que aponta para vértices já finalizados.

Questão 9)

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?*

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

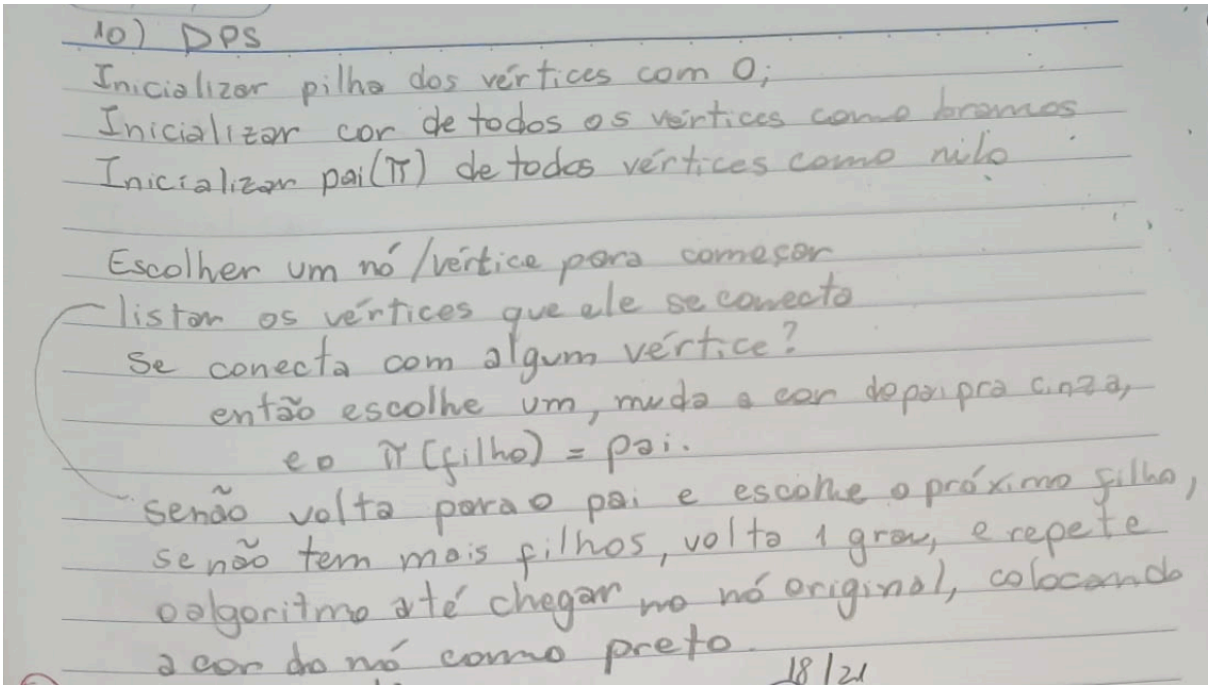


Matriz de adjacência

i/j	1	2	3	4	5	6	7	8	9
1	0	1	0	1	0	0	0	0	0
2	1	0	0	1	1	0	0	0	0
3	0	1	0	0	1	1	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0	0	0
8	0	1	1	0	0	0	0	0	0
9	0	0	0	0	0	1	0	0	0

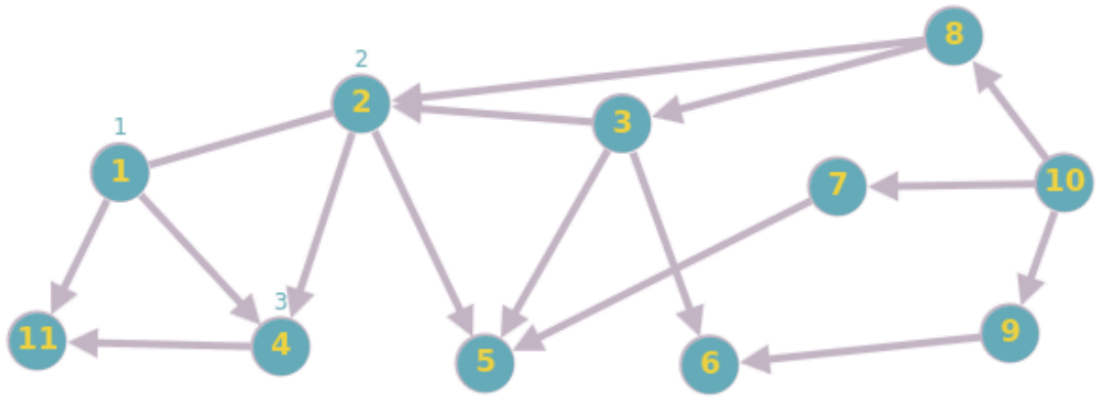
Lista de adjacência:
adj[1] = {2,4}
adj[2] = {1,4,5}
adj[3] = {2,5,6}
adj[4] = {}
adj[5] = {}
adj[6] = {}
adj[7] = {5}
adj[8] = {2,3}
adj[9] = {6}

1. Começa em um vértice inicial.
2. Visita um dos seus vértices adjacentes não visitados.
3. Continua visitando os vértices adjacentes até que não haja mais vértices adjacentes não visitados.
4. Retrocede (volta um passo) e explora outros vértices adjacentes que ainda não foram visitados.
5. O processo se repete até que todos os vértices conectados ao inicial sejam visitados.



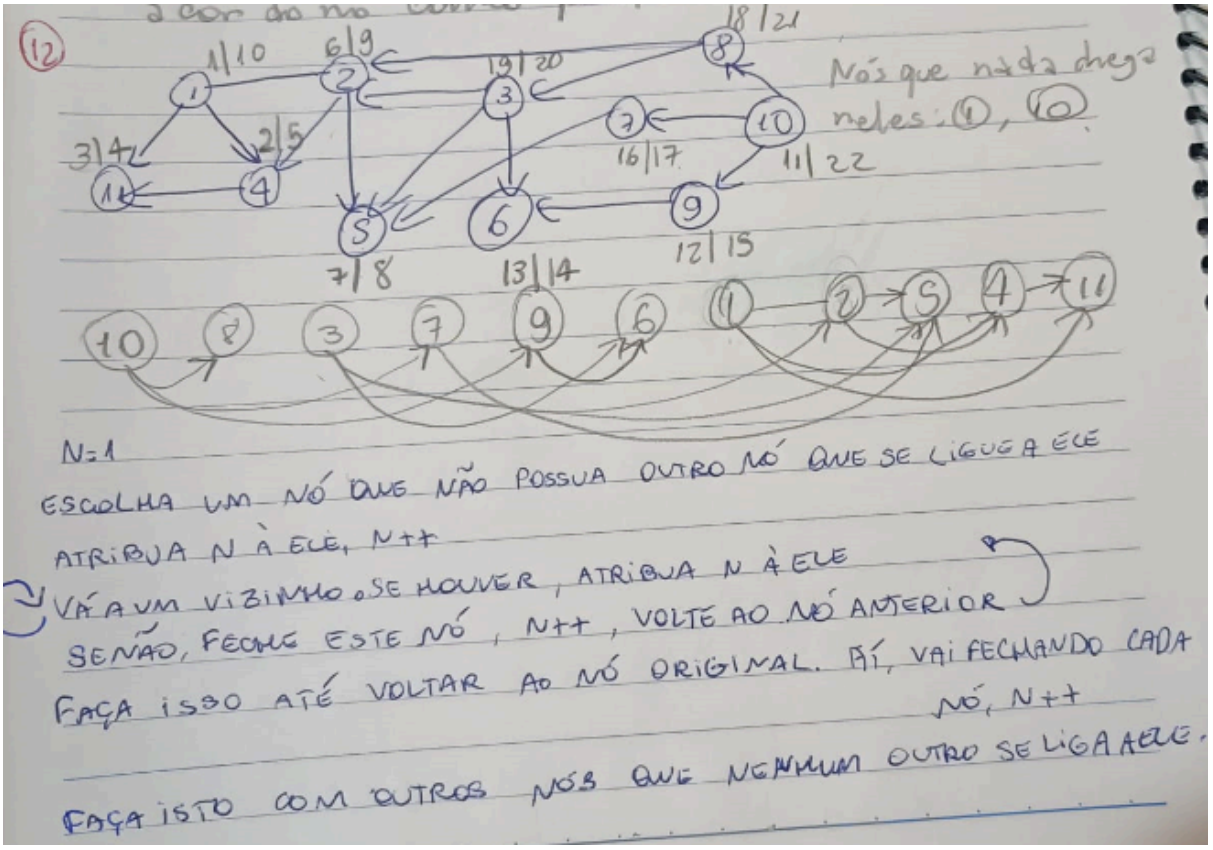
Exercício 11. *Apresente uma versão não recursiva para o algoritmo de busca em profundidade que possua a mesma complexidade da versão recursiva.*

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



adj[1] = {2,4,11}

adj[2] = {1,4,5}
adj[3] = {2,5,6}
adj[4] = {11}
adj[5] = {}
adj[6] = {}
adj[7] = {5}
adj[8] = {2,3}
adj[9] = {6}
adj[10] = {7,8,9}
adj[11] = {}



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

Ordenação topológica só é definida para *grafos dirigidos acíclicos* (DAGs - *Directed Acyclic Graphs*). Isso ocorre porque a ordenação topológica depende da ausência de ciclos para que seja possível definir uma sequência linear onde, para toda aresta u→v, o vértice u preceda v na ordem.

Se o grafo possuir ciclos dirigidos:

1. O algoritmo de ordenação topológica detectará o ciclo (diretamente ou indiretamente) ao tentar processar os vértices.
2. Ele não conseguirá produzir uma ordenação válida, porque os ciclos introduzem dependências circulares. Por exemplo, se A→B→C→A, não há uma maneira de linearizar A,B,C que satisfaça as dependências.

Conclusão:
Quando aplicado a um grafo com ciclos, o algoritmo de ordenação topológica falha ou retorna um resultado inválido, e a ordenação não faz sentido. Detectar ciclos é um passo crucial em muitos algoritmos para assegurar que a ordenação topológica seja aplicável.

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

Um algoritmo eficiente para detectar ciclos em um grafo não direcionado é baseado em *Busca em Profundidade* (DFS - *Depth-First Search*). Aqui estão os passos:

Algoritmo:

1. **Inicialização:** Marque todos os vértices como não visitados.
2. **DFS:**
   - Inicie a DFS a partir de um vértice arbitrário.
   - Para cada vértice v visitado, marque-o como visitado e explore seus vizinhos.
   - Se um vizinho u já tiver sido visitado e não for o "pai" do vértice v na DFS, isso indica a presença de um ciclo.
3. **Repetição:** Caso o grafo não seja conexo, repita o processo para todos os componentes desconexos.

Pseudocódigo (python)

```
def hasCycle(graph):
    visited = set()

    def dfs(v, parent):
        visited.add(v)
        for neighbor in graph[v]:
            if neighbor not in visited:
                if dfs(neighbor, v):  # Recurse to neighbor
                    return True
            elif neighbor != parent:  # A back edge
                return True
        return False
```

```
    for vertex in graph:
        if vertex not in visited:
            if dfs(vertex, None):  # Check each connected component
                return True

    return False
```

Complexidade:

- **Tempo:** O(V+E), onde V é o número de vértices e E o número de arestas. Isso ocorre porque cada aresta e cada vértice é processado no máximo uma vez.
- **Espaço:** O(V) para armazenar os vértices visitados e a recursão.

Conclusão:
Esse algoritmo é eficiente para grafos esparsos e densos e detecta ciclos em grafos não direcionados de maneira confiável.

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

Não é possível garantir que o algoritmo da questão anterior, que detecta ciclos em um grafo não direcionado, execute em O(∣V∣)O(|V|)O(∣V∣) **para toda entrada**, porque a análise do grafo depende tanto do número de vértices VVV quanto do número de arestas EEE. Assim, o limite inferior para qualquer algoritmo que percorre um grafo será O(V+E)O(V + E)O(V+E), já que você precisa visitar todas as arestas para processar completamente o grafo.
JUSTIFICATIVA:

1. Para um grafo não direcionado:
    - O número de arestas E pode variar de 0 a O(V^2), dependendo de quão denso é o grafo.
    - Mesmo em um grafo esparso, E pode ser próximo de V, o que ainda contribui para a complexidade O(V+E).
2. Se o grafo tiver muitas arestas (denso), qualquer algoritmo baseado em busca (como DFS ou BFS) terá de explorar todas essas arestas para garantir a detecção correta de ciclos.

Conclusão:

A complexidade O(V) só seria alcançada em cenários extremamente restritivos, como em grafos onde E = O(1) (um número constante de arestas). Porém, para **toda entrada**, O(V) não é possível devido à dependência intrínseca da quantidade de arestas EEE.

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

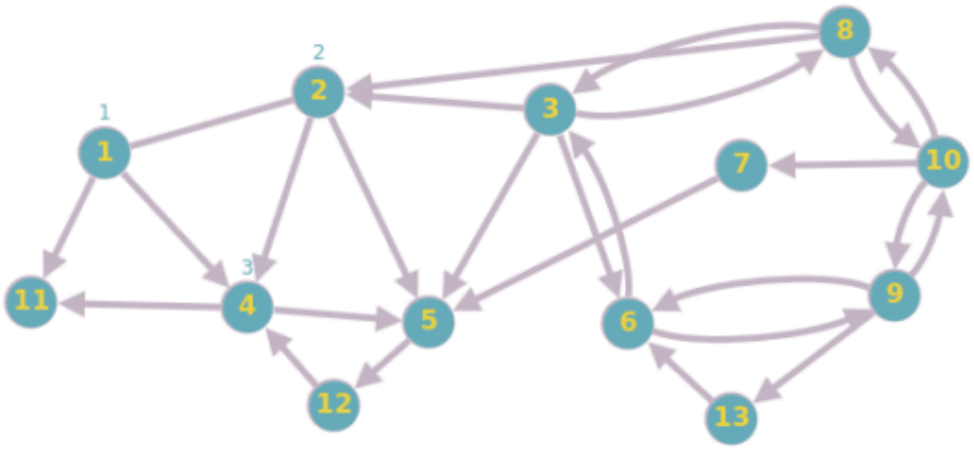
Se adicionarmos um arco entre vértices que fazem parte de componentes fortemente conexas diferentes, então haverá uma conexão entre os componentes onde não havia, e o número de componentes fortemente conexas é diminuído por 1 unidade.

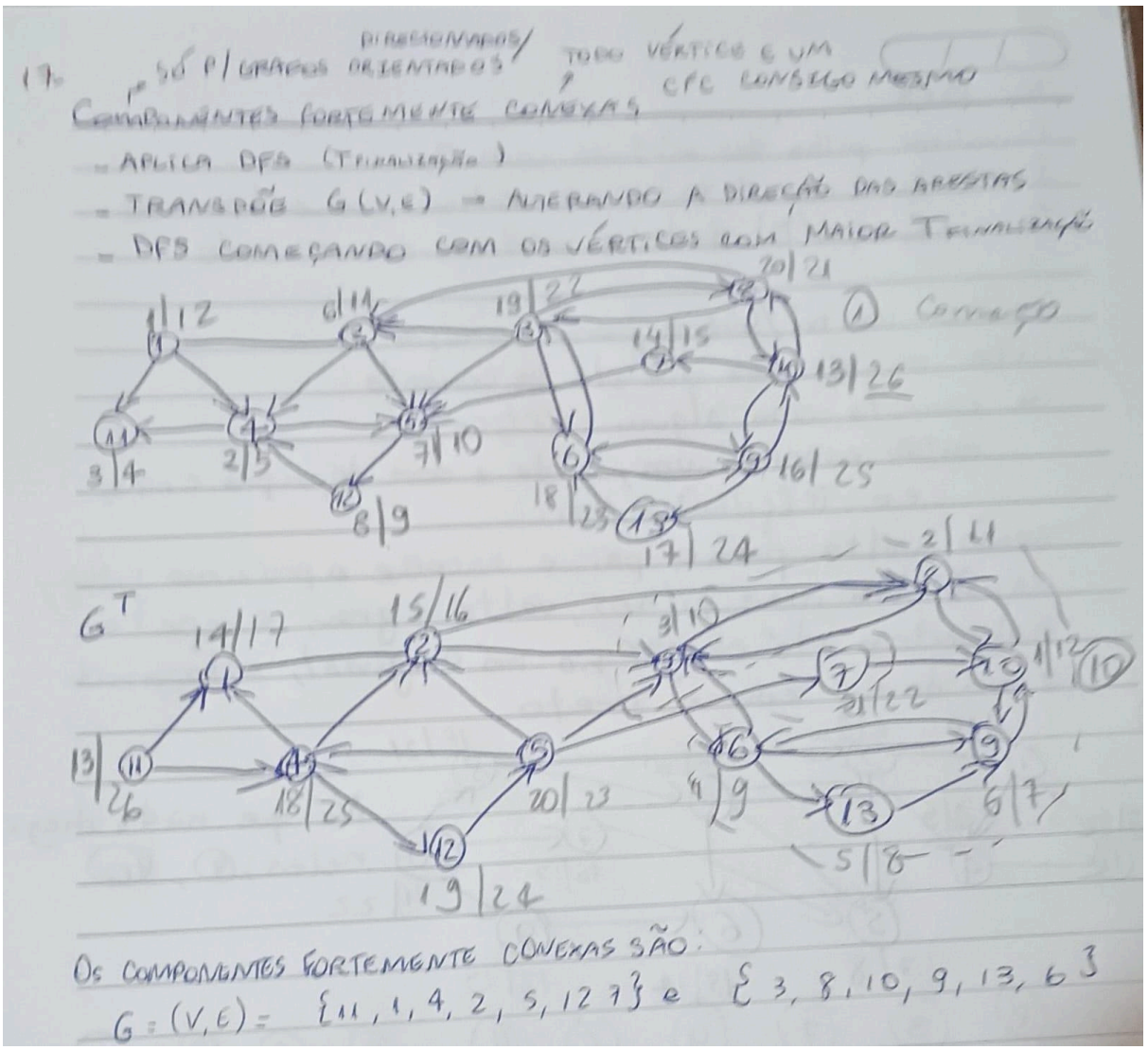
Definição de Componente Fortemente Conexo:

In the context of **directed graphs**, a **Strongly Connected Component (SCC)** is a **maximal subgraph** in which any two vertices are mutually reachable. This means that for every pair of vertices u and v in the SCC:

- There exists a directed path from u to v.
- There exists a directed path from v to u.

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






```
void DFS(node, visited) {
    visited[node] = true;
    FOR EACH n IN neighbours {
        IF NOT visited[n] {
            DFS(n, visited);
        }
    }
}
```