Exercício 1. *Prove ou refute: em uma busca em largura o conjunto formado pelos nós PRETOS sempre induz um grafo conexo.*

Em uma Busca em Largura (BFS), os nós pretos são aqueles que já foram visitados e não têm mais nenhum vizinho disponível (que ainda não visitou). A definição de grafo conexo é: "Um grafo é conexo se existir pelo menos um caminho entre cada par de vértices do grafo." Se o vértice está preto, significa que ele está conectado a pelo menos 1 outro vértice. Se ao final do algoritmo algum vértice ainda está branco, é porque ele não foi visitado (não há arestas em comum com algum subgrafo de *G*), logo este grafo não é conexo. Logo, por absurdo, foi provado que em uma busca em largura o conjunto formado pelos nós PRETOS sempre induz um grafo conexo.
Não necessariamente. Se o grafo *G* for desconexo, o algoritmo irá encontrar o conjunto conexo do subgrafo que pertence o nó original.

Exercício 2. *Prove ou refute: em uma busca em largura o conjunto formado pelos nós CINZAS sempre induz um grafo conexo.*

Refutável: ao final de um BFS, apenas é possível que os grafos sejam representados pelas cores branca (quando o vértice ainda não foi visitado, em caso de um vértice sozinho em um subgrafo) ou preto (quando ele tem conexão com outros grafos)

Na busca em largura (BFS), o conjunto de nós **cinzas** representa os nós que:

1. Foram descobertos e inseridos na fila para serem explorados.
2. Ainda não tiveram todos os seus vizinhos visitados (ou seja, ainda estão sendo processados).

Durante a execução da BFS:

- A BFS visita camadas do grafo. Um nó cinza pode ser conectado a outros nós cinzas ou a nós que estão na próxima camada da BFS.
- Enquanto a fila da BFS está sendo processada, os nós cinzas estão todos na mesma componente conectada (o processo da BFS garante isso).

Portanto, os **nós cinzas**, em qualquer momento específico durante a BFS, pertencem à mesma componente e são conectados.

Quando um nó se torna cinza, ele foi colocado na fila pela exploração de um nó vizinho. Isso significa que existe um caminho entre esse nó cinza e pelo menos outro nó cinza que o descobriu.

Assim, enquanto a BFS está em andamento, o conjunto de nós cinzas formará sempre um subgrafo conexo.

Exercício 3. *Prove ou refute: em uma busca em largura o conjunto formado pelos nós BRANCOS sempre induz um grafo conexo.*

Refutado: Os nós brancos são os não visitados ainda pelo algoritmo. Se o grafo for disconexo, os nós brancos podem ser parte de diferentes subgrafos, o que torna o grafo disconexo.

Exercício 4. *O diâmetro de uma grafo é seu maior menor caminho caminho de uma folha até a raiz. Dado uma árvore* $T = (V, E)$ *escreva um algoritmo que calcula o diâmetro dessa árvore. Qual a complexidade do seu algoritmo? Ele é eficiente?*

Para encontrar o diâmetro do grafo, temos que realizar o DFS, e a cada vez que ele encontrar um nó que não tenha mais vizinhos sem serem visitados, ele chegou no valor máximo daquela “ramificação”. Devemos armazenar este número (diâmetro), e continuar rodando o algoritmo, até encontrar um outro nó que tenha um maior número de arestas, e atualizar o valor do diâmetro. No final do algoritmo, ele executará em O(V). O algoritmo é eficiente.

Exercício 5. *Dado um grafo* $G = (V, E)$ *qualquer escreva um algoritmo que calcula o diâmetro do grafo. Qual a complexidade do seu algoritmo?*

A definição de diâmetro nos diz que ela é a distância entre dois vértices de um grafo. Logo, para detectar o maior diâmetro em um grafo, devemos rodar um algoritmo para verificar o diâmetro do grafo V vezes (uma vez para cada vértice). Além disso, se o grafo possuir componentes desconexos, o diâmetro entre vértices de componentes diferentes é infinito. Primeiramente, seria necessário rodar o algoritmo DPS ou BPS para descobrir se o grafo é conexo ou não. Se não for, o diâmetro é infinito. Senão, devemos guardar o valor da distância e comparar com a aplicação do mesmo algoritmo, só que começando por outro vértice. Deve-se fazer isso V-1 vezes

Para calcular o **diâmetro de um grafo qualquer G=(V,E)** (não necessariamente uma árvore), a tarefa é mais complexa, porque:

1. O diâmetro é definido como a maior distância entre dois vértices no grafo, o que requer analisar **todos os pares de vértices conectados**.

2. É necessário lidar com **componentes desconexas** (o diâmetro é infinito para um grafo desconexo).

Algoritmos possíveis

1. Algoritmo baseado no cálculo de todas as menores distâncias:

O método mais direto envolve encontrar as distâncias mínimas entre todos os pares de vértices utilizando o **algoritmo de Floyd-Warshall** ou o **algoritmo de Dijkstra** para cada vértice.

Algoritmo:

1. Use o **algoritmo de Floyd-Warshall** para calcular todas as distâncias mínimas entre todos os pares de vértices.
2. Para cada par (u,v), registre a distância d(u,v).
3. O diâmetro será o maior valor finito de d(u,v) encontrado.

Pseudocódigo (Floyd-Warshall):

```
def graph_diameter(graph, n):
    # Inicializar a matriz de distâncias com infinito (ou muito grande)
    INF = float('inf')
    dist = [[INF] * n for _ in range(n)]

    # Configurar a distância inicial (0 para si mesmo, peso para arestas existentes)
    for u in range(n):
        dist[u][u] = 0  # Distância de um vértice a ele mesmo é 0
        for v, weight in graph[u]:  # graph[u] é uma lista de (vizinho, peso)
            dist[u][v] = weight

    # Aplicar Floyd-Warshall
    for k in range(n):  # Iterar sobre vértices intermediários
        for i in range(n):
            for j in range(n):
                if dist[i][k] < INF and dist[k][j] < INF:
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    # Encontrar o maior valor finito na matriz de distâncias
```

```
    diameter = 0
    for i in range(n):
        for j in range(n):
            if dist[i][j] < INF:
                diameter = max(diameter, dist[i][j])

    return diameter
```

Complexidade:

- O **Floyd-Warshall** tem complexidade O(V^3), onde v é o número de vértices.
- Em grafos pequenos ou densos, isso pode ser aceitável, mas para grafos muito grandes, é ineficiente.

---

2. Algoritmo baseado em BFS para grafos não ponderados:

Se o grafo **não tem pesos nas arestas** (ou todos os pesos são iguais), o diâmetro pode ser calculado de forma eficiente com BFS:

1. Execute uma **BFS a partir de cada vértice v** para calcular a distância máxima a partir dele para todos os outros vértices.
2. O maior valor entre as distâncias máximas encontradas é o diâmetro.

Algoritmo:

1. Para cada vértice v, execute uma **BFS**.
2. Registre a distância máxima alcançada na BFS.
3. Retorne o maior valor entre todas as distâncias máximas.

Complexidade:

- Executar v BFSs: O(V ⋅ (V+E)).
- Em grafos esparsos (E=O(V), isso é O(V^2), mas em grafos densos (E=O(V2, é O(V^3).

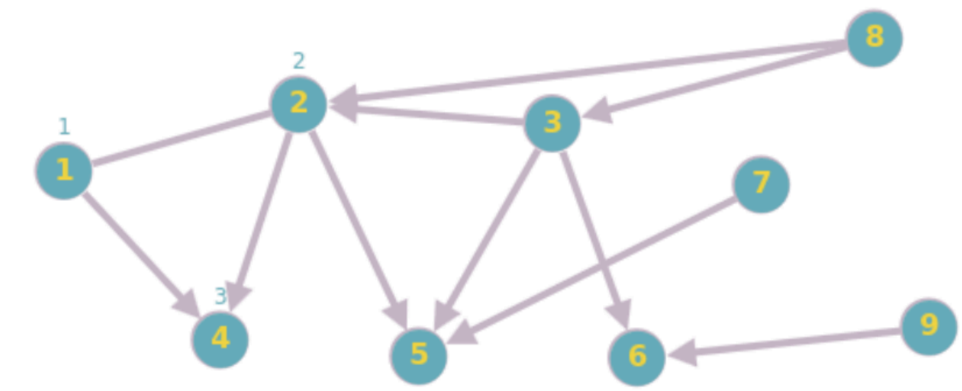
Eficiência:

O método escolhido depende do tipo de grafo:

- Para **grafos esparsos ou não ponderados**, usar BFS é mais eficiente $O(V^2)$.
- Para **grafos densos ou ponderados**, Floyd-Warshall $O(V^3)$ ou Dijkstra repetido ($O(V \cdot (E+V\log V))$) são alternativas.

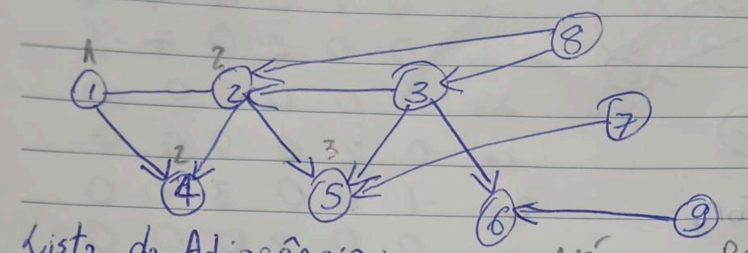
Se o grafo for muito grande, esses métodos podem ser inviáveis, e técnicas aproximadas, como heurísticas, podem ser mais adequadas.

Exercício 6. *Aplique o algoritmo de busca em largura para o grafo abaixo:*



4 - Busca em largura - Achar caminhos mínimos

6 -



BFS - start = 1

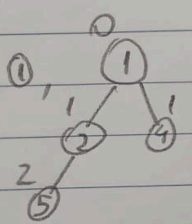
Lista de Adjacência:

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}

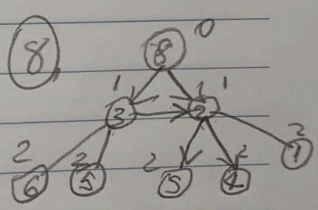
Nó	PAI	d
1	1	0
2	1	1
3		
4	1	1
5	2	2
6		
7		
8		
9		

// FINISH

Se começar do 1,



Se começar do 8



Exercício 7. *A aplicação da busca em largura em um grafo ponderado nas arestas não produz os caminhos de custo mínimo (considerando como o tamanho do caminho sendo a soma das arestas). Dê um exemplo para ilustrar esse fato.*

O BFS pode ser utilizado para encontrar o caminho mínimo no sentido de número de arestas, mas não quando o grafo é ponderado.

Exercício 8. *É possível modificar o algoritmo de busca em largura para calcular o menor caminho mesmo em um grafo ponderado? Como? Qual a nova complexidade desse algoritmo? Sua abordagem funciona se o grafo tiver pesos negativos?*

Exercício 9. *Em um DAG, é possível executar o laço interior do algoritmo de Belman-Ford apenas uma vez, se os vértices forem ordenados de forma conveniente antes. Qual seria essa ordenação? Qual a complexidade do algoritmo obtido dessa forma?*

Exercício 10. *Considere o conjunto de inequações abaixo:*

$$
\begin{aligned}
x_1 - x_2 &\leq 1 \\
x_1 - x_4 &\leq -4 \\
x_2 - x_3 &\leq 2 \\
x_2 - x_5 &\leq 7 \\
x_2 - x_6 &\leq 5 \\
x_3 - x_6 &\leq 10 \\
x_4 - x_2 &\leq 2 \\
x_5 - x_1 &\leq -1 \\
x_5 - x_4 &\leq 3 \\
x_6 - x_3 &\leq -8
\end{aligned}
$$

Determine uma solução viável (que respeite todas as restrições) para esse conjunto de inequações.

Exercício 11. *Considere um conjunto de m inequações sobre n variáveis na forma $x_i - x_j \leq b_k$. Apresente um algoritmo para determinar se esse conjunto de inequações possui uma solução viável ou não. Qual a complexidade do seu algoritmo?*

Exercício 12. *Usando as propriedades de caminho mínimo em um grafo não direcionado é possível conceber um algoritmo recursivo para calcular todos os menores caminhos a um vértice fixo* $s \in V(G)$. *Apresente um algoritmo recursivo (ou uma relação de recorrência) que faça exatamente isso.*