

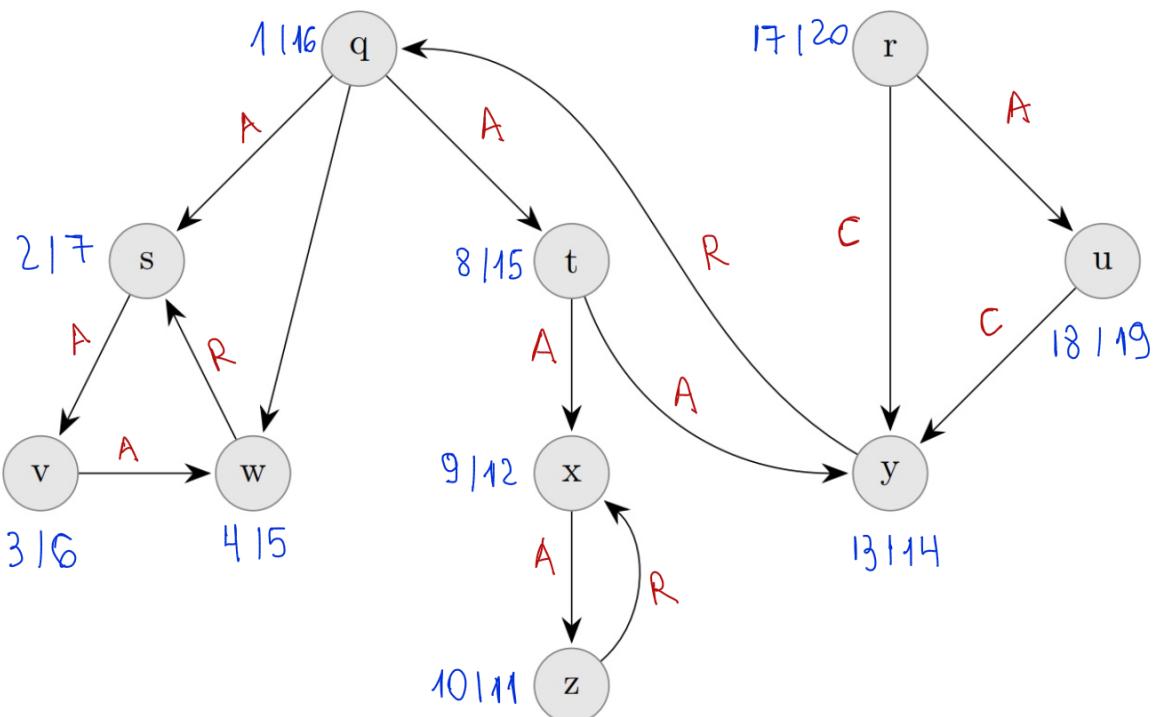
①

a) 10 passos.

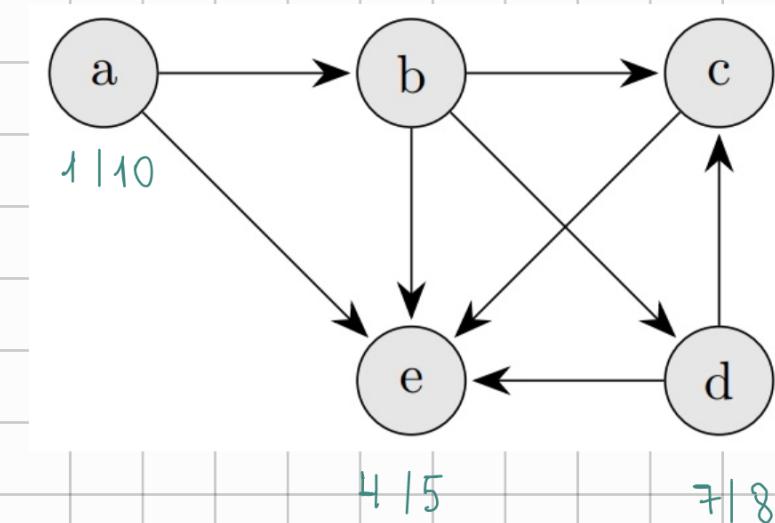
b) Cada célula livre seria um nó cujos vizinhos são as células livres que estejam acima, abaixo, à esquerda ou à direita.

c) Busca em largura. Como esse algoritmo nos entrega o caminho mínimo de um vértice source para cada outro vértice alcançável por ele do grafo, basta rodá-lo com o vértice R como source e verificar a quantidade de vértices no caminho mínimo entre R e C.

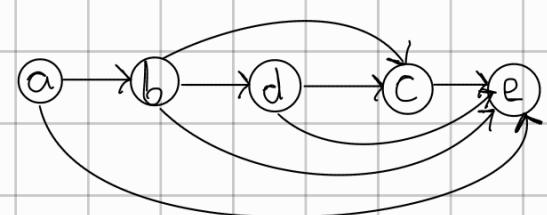
⑧



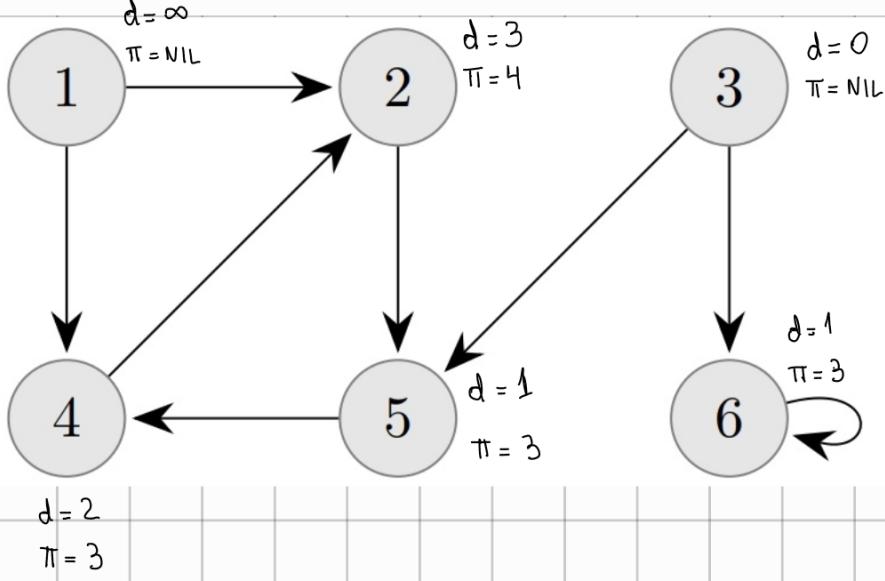
⑫



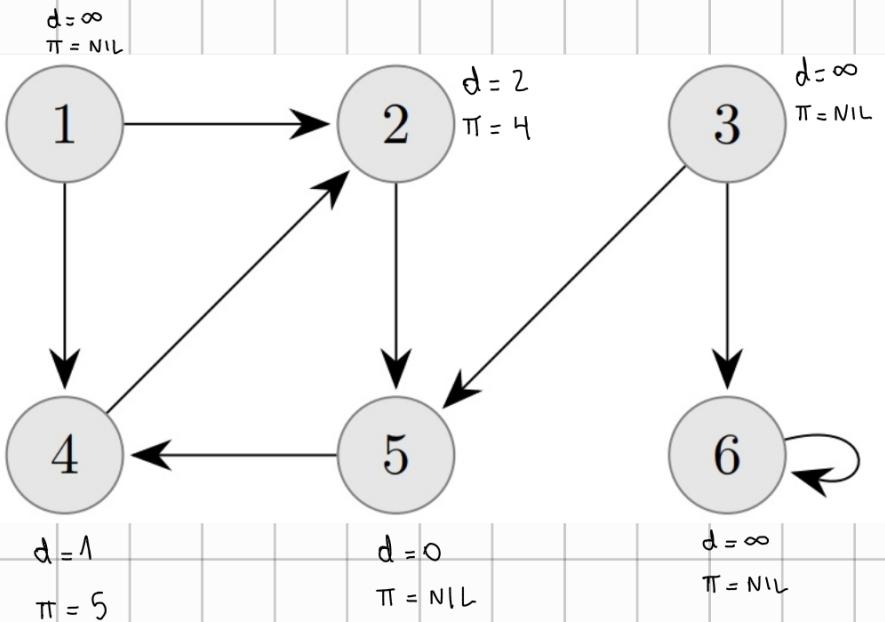
1. DFS p/ determinar d e f.
2. Ordenar os vértices pelo f decrescente.



⑥



b)



Na primeira busca, calculamos d e π apenas dos vértices 3, 5, 6, 2 e 4 e na segunda 5, 2 e 4. Isso ocorre porque o algoritmo retorna os caminhos mínimos entre o vértice source e seus vértices alcançáveis e, como se trata de um grafo direcionado, cada vértice não alcança necessariamente todos os outros.

②

a) O grafo de Ktar será um grafo direcionado e não ponderado.

Para sua representação, usaremos listas de adjacências pois teremos um algoritmo $O(\deg(v))$ para calcular os vizinhos de um vértice e $O(V+E)$ para encontrar as CFCs.

b) numero_seguidos (G, v) :

contador = 0

for v in G.Adj[v] :

 contador += 1;

return contador

c) numero_seguidores (G, v) :

contador = 0

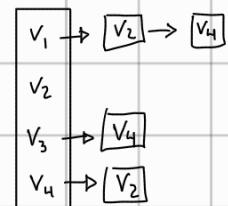
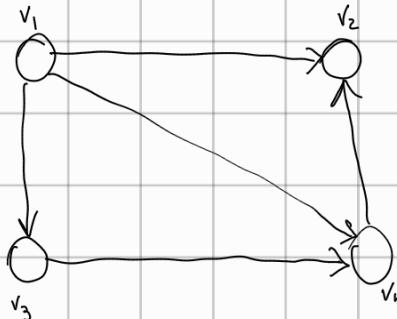
for v in G :

 for z in Adj[v] :

 if (z == v) :

 Contador += 1

return contador



d) get_cfc (G, v) :

pilha = \emptyset

for v in G :

 if v.color == WHITE :

 DFS1(G, v)

reinicializar_grafo(G)

G^T = transpor_grafo(G)

while pilha :

 v = pop(pilha)

 if v.color == WHITE :

 componente = \emptyset

 DFS2(G, v, componente)

 if v \in componente :

 return componente

DFS1(G, v, pilha)

v.color = GREY

for v in Adj[v] :

 if v.color == WHITE :

 DFS1(G, v, pilha)

push(pilha, v)

DFS2(G, v, componente)

v.color = GREY

componente.adiciona(v)

for v in Adj[v] :

 if v.color == WHITE :

 DFS2(G, v, componente)

③ transpor - grafo (G) : LISTA ADJACÊNCIAS

$$T = \emptyset$$

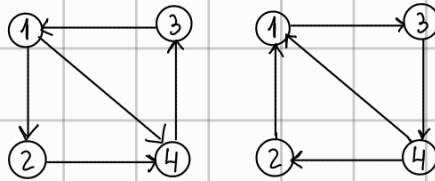
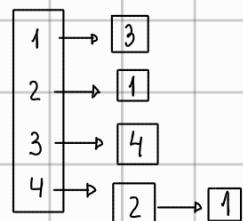
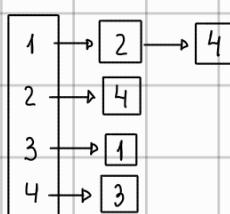
$$T.V = G.V$$

for $v \in G.V$:

for $v \in \text{Adj}[v]$:

$$T.E \cup (v, u)$$

return T



Como no pior caso passamos por todos os vértices e arestas, temos que a complexidade do algoritmo no caso de listas de adjacência é $O(V+E)$.

transpor - grafo (G) : MATRIZ ADJACÊNCIAS

n = número de vértices de G

T = matriz $n \times n$ inicializada com 0

for i in $(1, n+1)$:

for j in $(1, n+1)$:

if $G[i][j] = 1$:

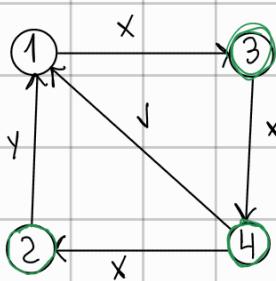
$$T[j][i] = 1$$

return T

	1	2	3	4
1	0	1	0	1
2	0	0	0	1
3	1	0	0	0
4	0	0	1	0

Como obrigatoriamente passamos por todos os pares (i, j) - existindo aresta ou não - temos que a complexidade do algoritmo é dada por $O(V^2)$.

④



1, 2, 3, 4 usuários

x, y, z, w, v notícias

$$1 \rightarrow 3$$

$$2 \rightarrow 1$$

$$3 \rightarrow 4$$

$$4 \rightarrow 1 \rightarrow 2$$

a) Esse grafo será modelado da seguinte forma:

- Cada usuário será um vértice com atributo de identificação numérico;

- Uma aresta (u, v) com atributo de identificação x significa que o usuário v recebeu a notícia x do usuário u .
- Usaremos listas de adjacência para representá-lo pois buscas costumam performar melhor nesse tipo de representação.

b) alcance_noticia (G, x) :

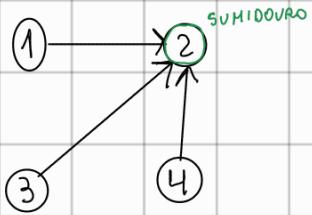
```

cont = 0
for v ∈ G.V:
    for v ∈ Adjs[v]:
        if v.idx == x:
            cont += 1
return cont

```

c e d) Não entendi.

5



	1	2	3	4
1	0	1	0	0
2	0	0	0	0
3	0	1	0	0
4	0	1	0	0

 $O(V)$

```

detectar_universal_sink(A, n):
    candidato = 1
    for i de 2 até n:
        if A[candidato][i] == 1:
            // candidato tem aresta de saída → não pode ser sumidouro
            candidato = i
        // senão, i não pode ser sumidouro → ignorar

    // Verificação final: conferir se candidato é de fato universal sink
    for j de 1 até n:
        if j != candidato:
            if A[candidato][j] == 1 ou A[j][candidato] == 0:
                return -1 // Não existe universal sink
    return candidato // Encontrado!

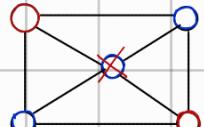
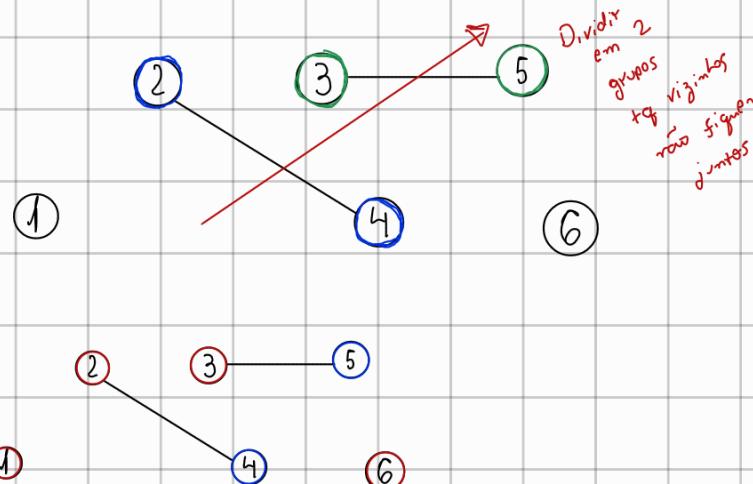
```

Porque detectar esse banco permite saber quem tem o controle central da liquidez do sistema e monitorar o risco de sua falência. No mais, a matriz de adjacência é melhor nesse caso pois verificar a existência de uma aresta é $O(1)$. Assim, se usássemos lista de adjacência, certamente teríamos $O(V+E)$.

Gerando o algoritmo $O(V)$

7. n comparsas em 2 corpos.

- \checkmark pares de comparsas não podem estar no mesmo corpo.
- exemplos $v = 2 \text{ e } 4 / 3 \text{ e } 5$



planejamento (G):

for $v \in G.V$:

$v.visitado = False$

$v.color = None$

for $s \in G.V$:

cor_oposta(s):

if $s.color == RED$:

return BLUE

return RED

if not $s.\text{visitado}$:

$Q = \emptyset$

$s.\text{color} = \text{RED}$

ENQUEUE(Q, s)

while Q :

$v = \text{DEQUEUE}(Q)$

for $u \in \text{Adj}[v]$:

if not $u.\text{visitado}$:

$u.\text{color} = \text{cor_oposta}(v)$

ENQUEUE(Q, u)

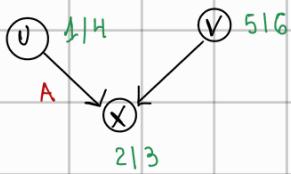
if $u.\text{color} == v.\text{color}$:

return -1

$v.\text{visitado} = \text{True}$

return grafo particionado

⑨



⑩

pontos_articulacao(G):

lista_pontos = []

for $v \in G.V$:

$T.V = G.V - \{v\}$

$T.E = G.E - \{(x,y) \text{ t.q. } x == v \text{ ou } y == v\}$

w = random($T.V$)

BFS(T, w)

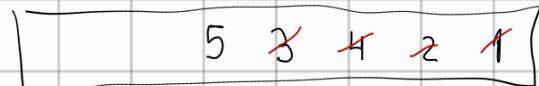
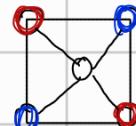
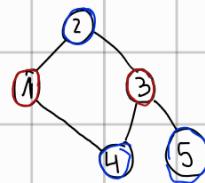
for $u \in T.V$:

if $u.d == +\infty$

lista_pontos.append(u)

break

return lista_pontos



Esses usuários são fundamentais na difusão de informações pela rede, considerando correntes distintas e sua eventual saída contribui para formação de bolhas. Ademais, estudar esses atores é fundamental p/ compreender melhor as interações entre comunidades, preservar difusão de mensagens e protegê-las.

11) a)