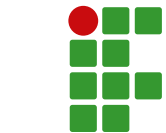
Percurso em grafos

Teoria dos Grafos

Prof. Daniel Saad Nogueira Nunes



INSTITUTO FEDERAL
Brasília

Campus
Taguatinga

Sumário

Introdução

- Existem diferentes formas de percorrer um grafo.
- Duas dessas formas são por profundidade (DFS) e por largura (BFS).
- Além de simplesmente percorrer o grafo, podemos usar essas abordagens para resolver várias tarefas como: encontrar caminhos mínimos entre os vértices; detectar ciclos, vértices de articulação ou pontes; encontrar componentes conexas; e realizar ordenação topológica em DAGs; entre outros.
- Exploraremos nessa aula essas abordagens e as suas aplicações.

Sumário

Percurso em grafos

- Estamos interessados em percorrer os grafos de maneira ótima, isto é, em tempo $\Theta(n + m)$.
- Diferentes abordagens permitem que resolvamos diferentes tipos de problemas.
- Iremos nos concentrar na busca em largura (Breath-first-search - BFS) e a busca em profundidade (Depth-first-search - DFS).

Sumário

BFS

- Em uma busca em largura os vértices são visitados em níveis.
- Primeiro, visitamos o vértice inicial (nível 0).
- Em seguida, os vizinhos imediatos do vértice inicial (nível 1).
- Em seguida, os vizinhos imediatos dos vértices do nível 1 (nível 2).
- Assim por diante.

BFS

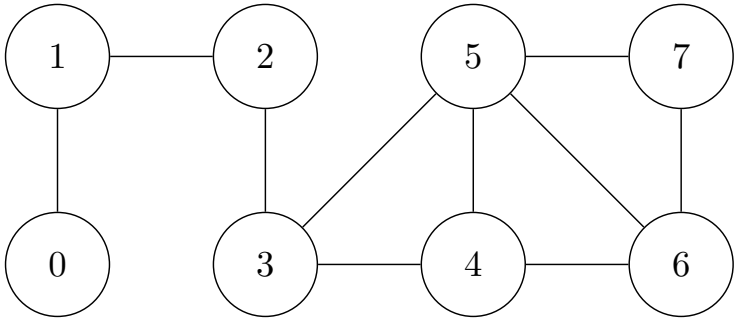
- Para conseguir esse comportamento, utilizamos uma fila.
- Inicialmente colocamos o vértice inicial na fila.
- Enquanto a fila não for vazia:
  - Processamos o nó que está na frente da fila;
  - Inserimos os nós adjacentes a ele, desde que não tenham iniciado o seu processamento ainda.

BFS

Adotaremos a seguinte conveção de cores de vértices para sinalizar o estágio de processamento:

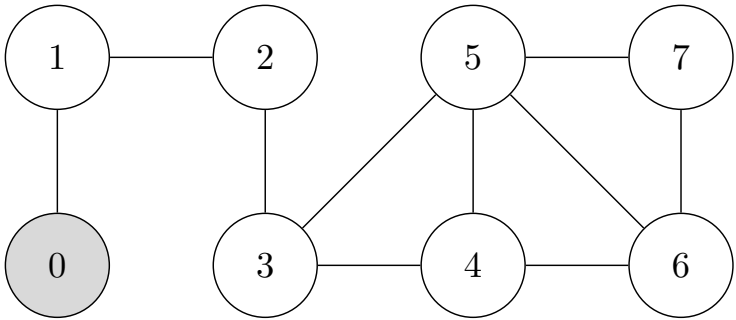
- Branco: não processado.
- Cinza: em processamento (na fila).
- Preto: processado.

BFS: exemplo



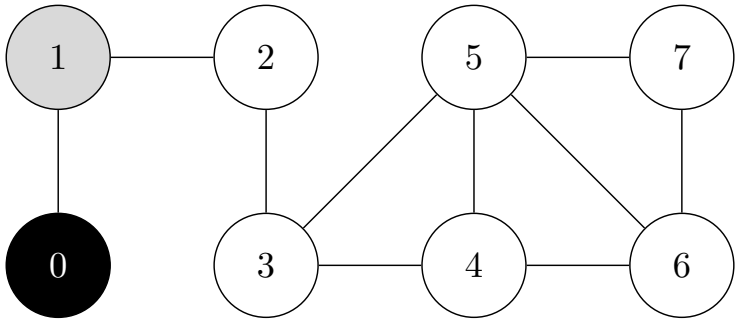
Q :

BFS: exemplo



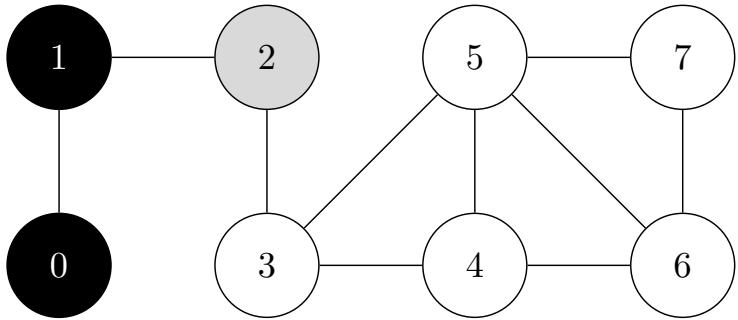
Q : 0

BFS: exemplo



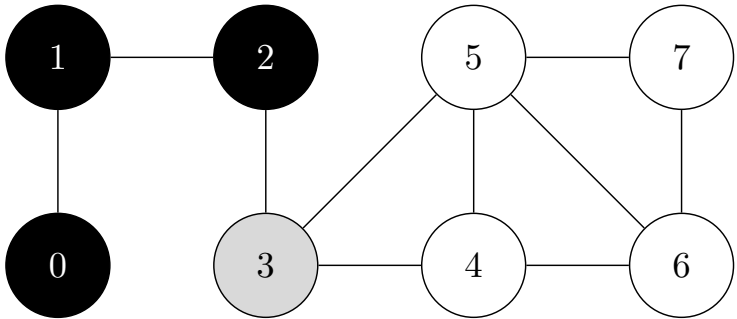
Q : 1

BFS: exemplo



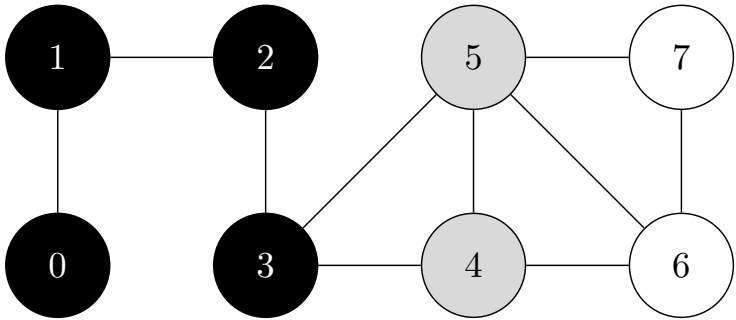
Q : 2

BFS: exemplo



Q : 3

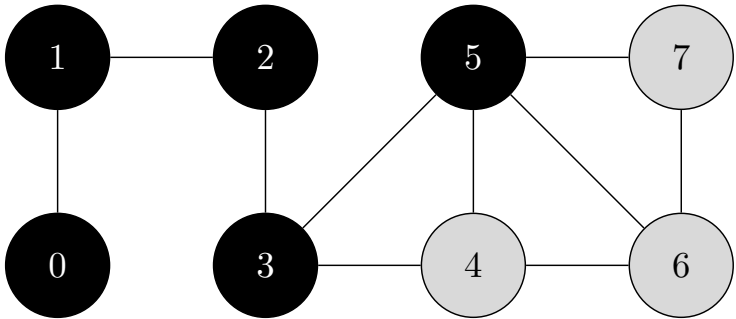
BFS: exemplo



Q :

5	4

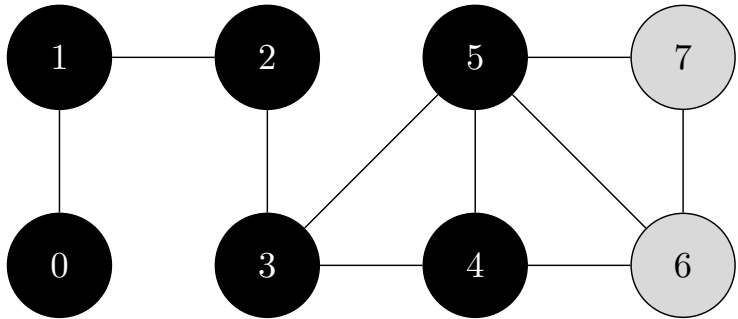
BFS: exemplo



Q :

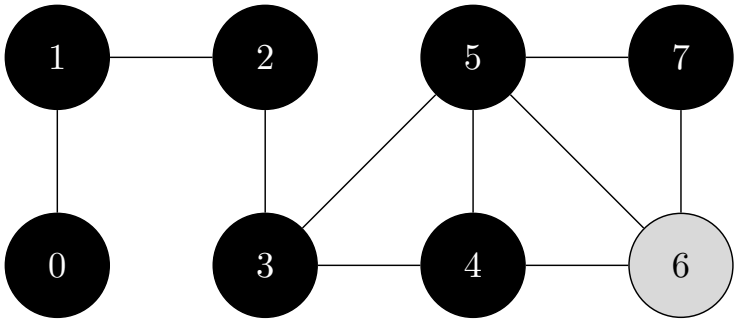
4	7	6

BFS: exemplo



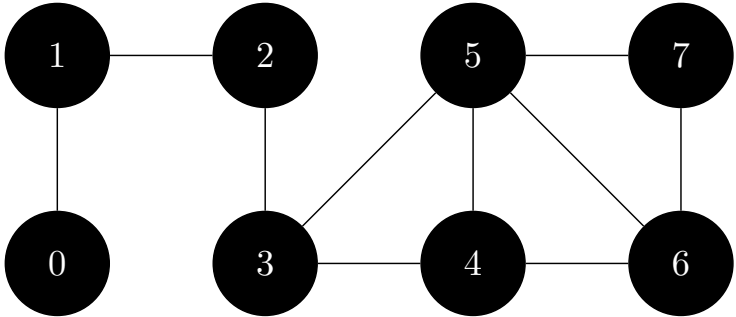
Q : | 7 | 6 |

BFS: exemplo



Q : 6

BFS: exemplo



Q :

BFS: algoritmo

Algorithm 1: $\text{BFS}(G, color, u)$

Input: G, o grafo, $color$, o vetor de cores, u o vértice inicial.

```
1  foreach v ∈ V do color[v] ← white
2  Q ← QUEUE()
3  Q.PUSH(u)
4  color[u] ← gray
5  while ¬Q.EMPTY() do
6      v ← Q.FRONT()
7      Q.POP()
8      foreach (v, w) ∈ E do
9          if color[w] = white then
10             color[w] ← gray
11             Q.PUSH(w)
12     color[v] ← black
```

BFS: análise

Análise de pior-caso

Cada vértice é colocado no máximo, uma vez na fila. Além disso, todas as arestas do grafo são inspecionadas.

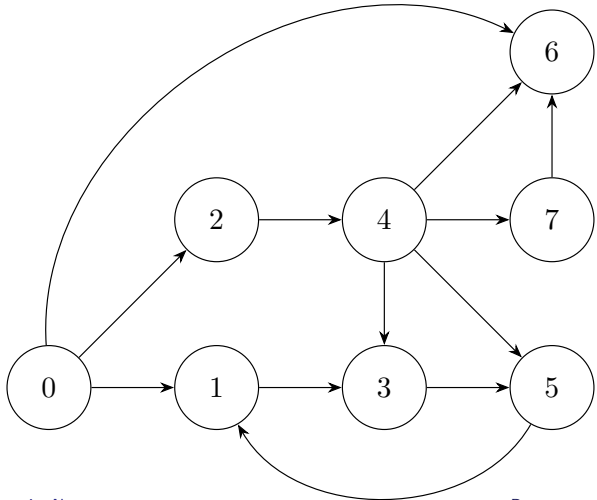
Custo: $\Theta(n + m)$ com listas de adjacências. $\Theta(n^2)$ com matriz de adjacências.

Sumário

DFS

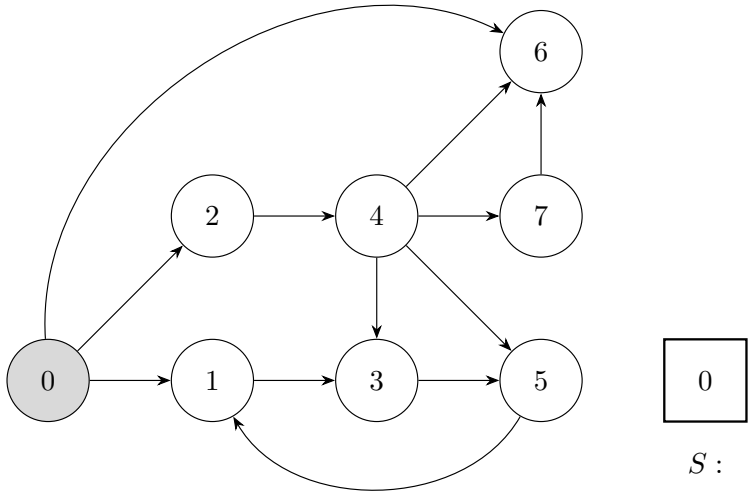
- A busca em profundidade (Depth-First-Search - DFS) inicia de um nó u e avança até o seu vizinho imediato, digamos, v.
- **Recursivamente** a busca continua de v. Isto é, usa-se a mesma estratégia.
- Quando um vértice não tem mais vizinhos a serem explorados, a busca **retrocede** ao vértice anterior.
- Durante as chamadas recursivas, uma pilha implícita de chamadas é mantida, o que possibilita o retrocesso e a exploração de novos caminhos.

DFS: exemplo

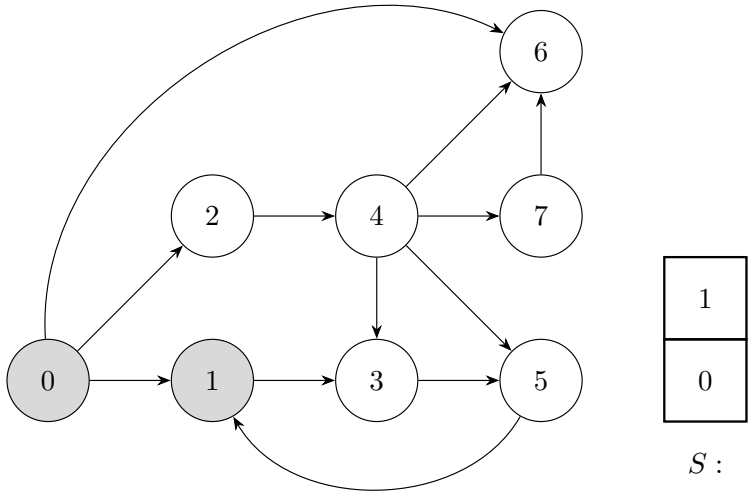


S :

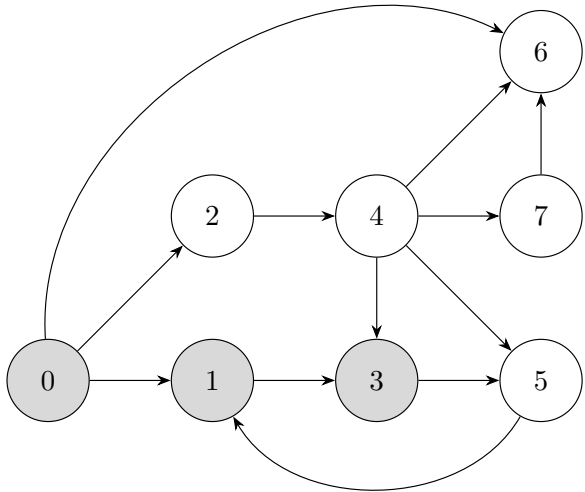
DFS: exemplo

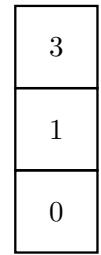


DFS: exemplo



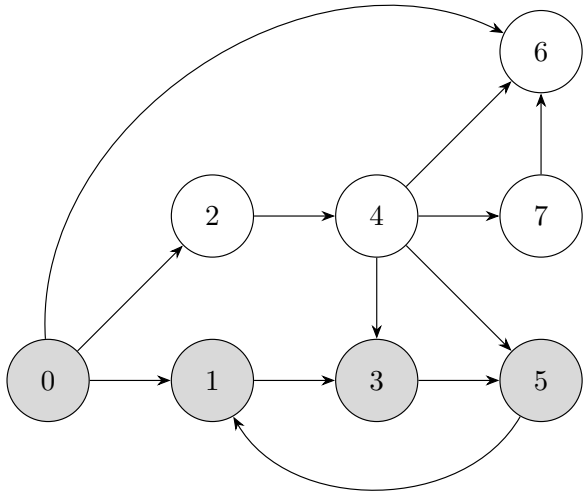
DFS: exemplo

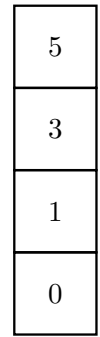




S :

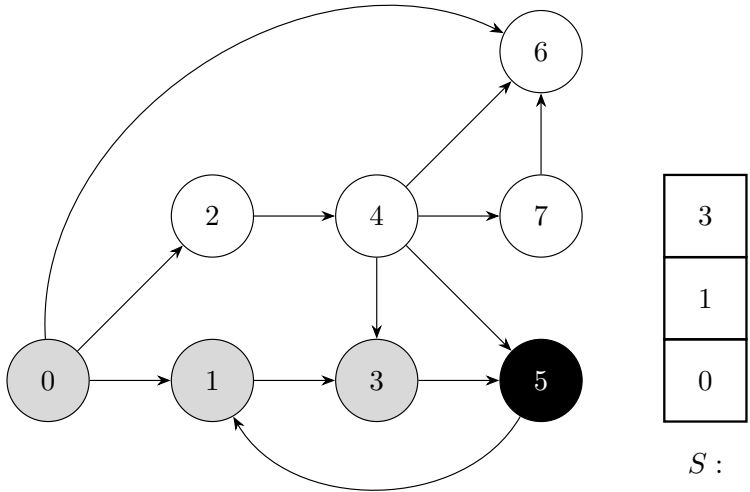
DFS: exemplo



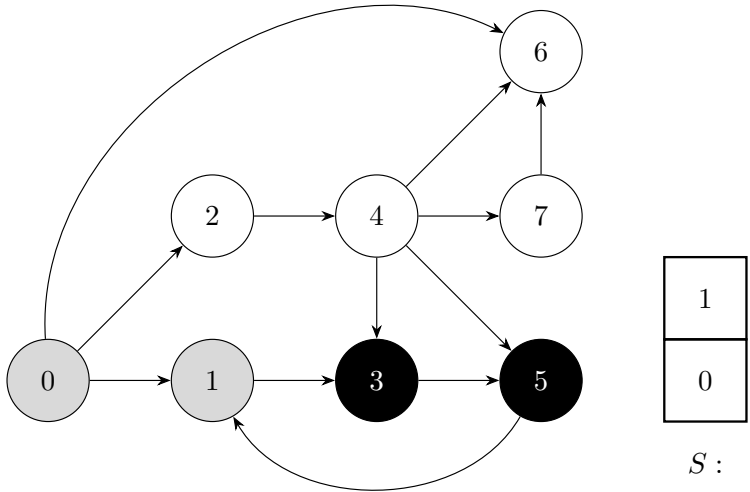


S :

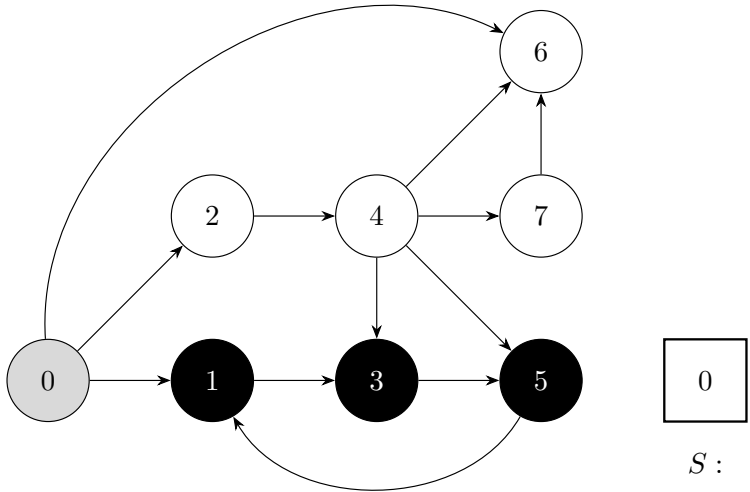
DFS: exemplo



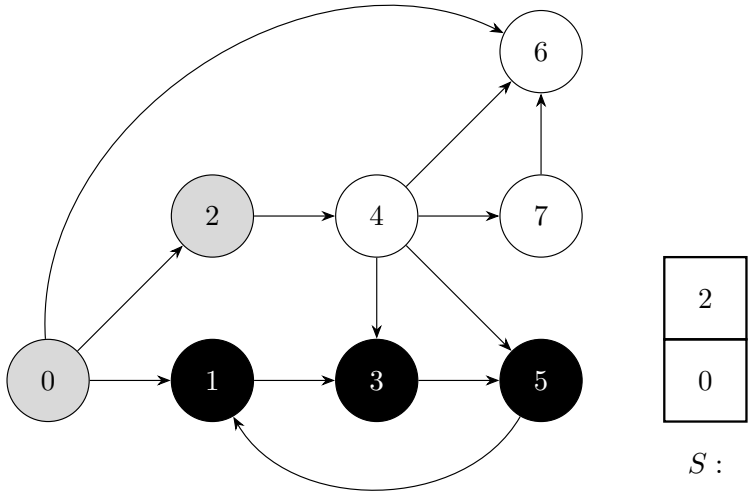
DFS: exemplo



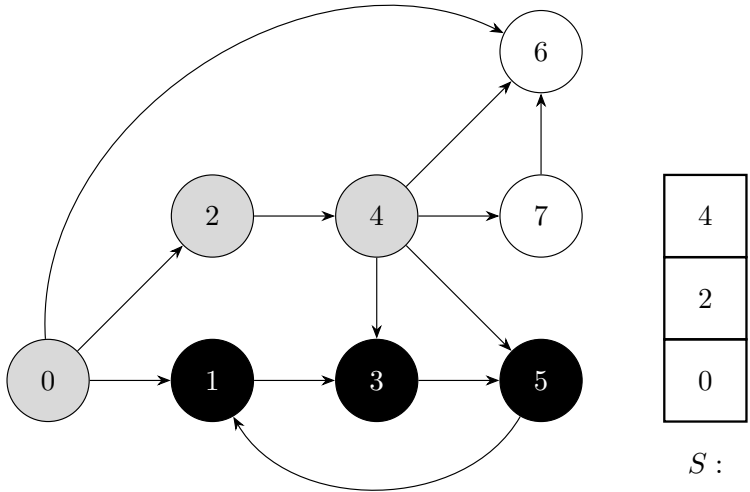
DFS: exemplo



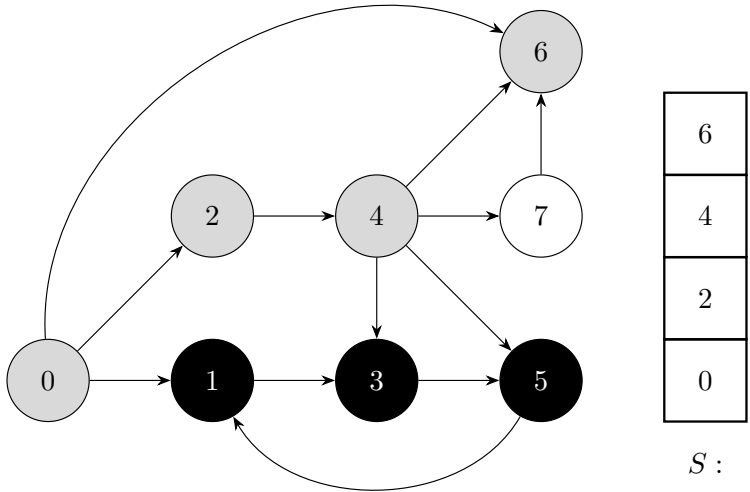
DFS: exemplo



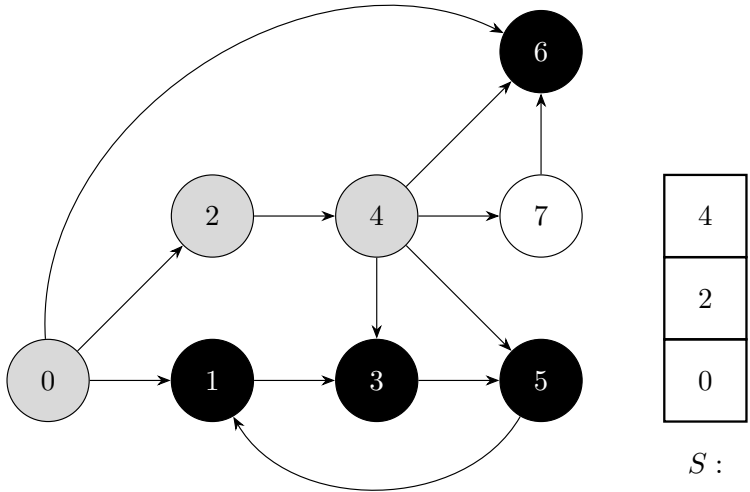
DFS: exemplo



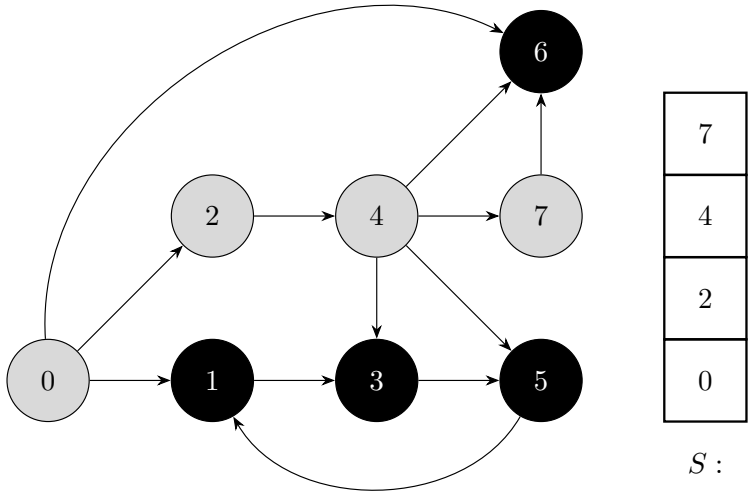
DFS: exemplo



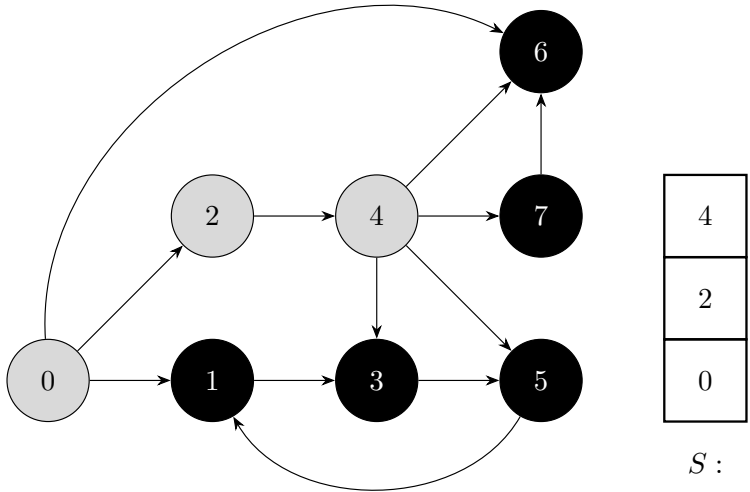
DFS: exemplo



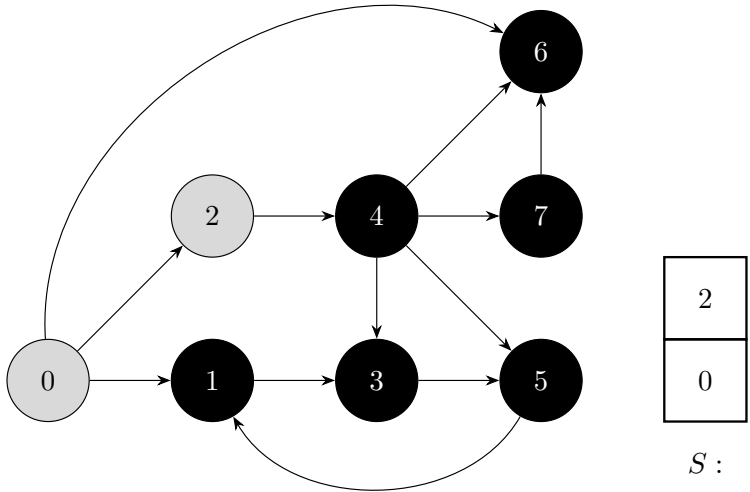
DFS: exemplo



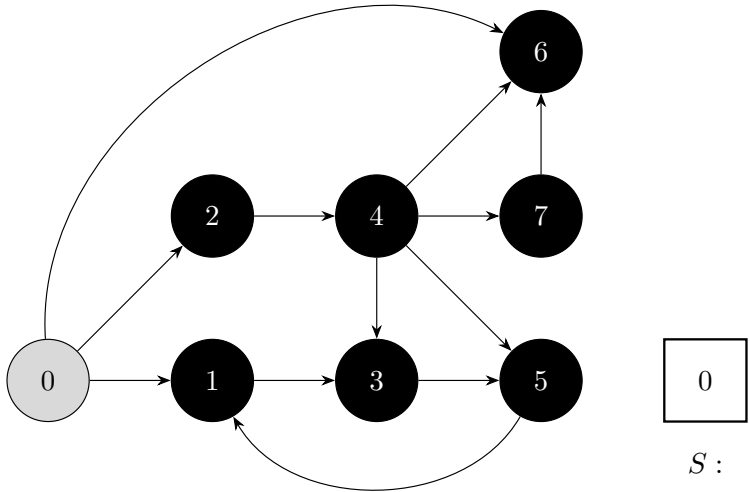
DFS: exemplo



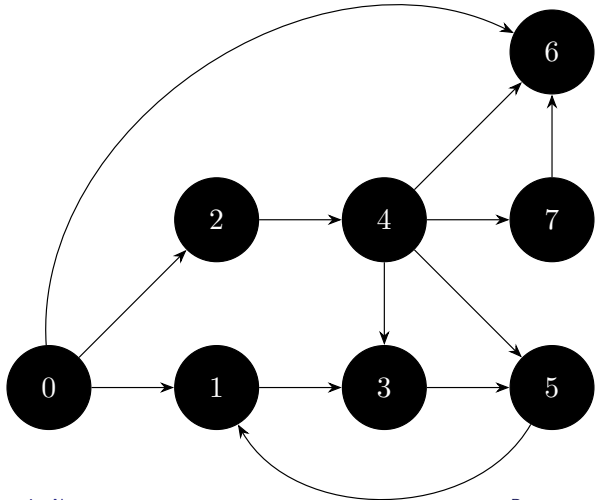
DFS: exemplo



DFS: exemplo



DFS: exemplo



S :

DFS

Manteremos a mesma convenção de cores:

- Vértices brancos: não processados.
- Vértices cinzas: em processamento.
- Vértices pretos: processados.

DFS: algoritmo

Algorithm 2: $\text{DFS}(G, color, u)$

Input: G, o grafo, $color$, o vetor de cores, e u, a raiz da DFS.

```
1 color[u] ← gray
2 foreach (u, v) ∈ E do
3     if color[v] = white then
4         DFS(G, color, v)
5 color[u] ← black
```

DFS: análise

Análise de pior-caso

Cada aresta é explorada uma única vez, então o tempo de execução da DFS é $\Theta(n + m)$ para listas de adjacências e $\Theta(n^2)$ para matrizes de adjacências.

DFS: tipos de arestas

Podemos classificar as arestas de uma DFS em quatro tipos:

- Tree edge.
- Cross edge.
- Back edge.
- Forward edge.

DFS: arestas

Tree edge

Uma tree edge é uma aresta (u, v) tal que v foi descoberto pela primeira vez a partir de u. São arestas que correspondem a caminhos em cada ramo da árvore de DFS.

DFS: arestas

Forward edge

Uma forward edge é uma aresta que liga um vértice a um descendente que já foi descoberto.

DFS: arestas

Cross edge

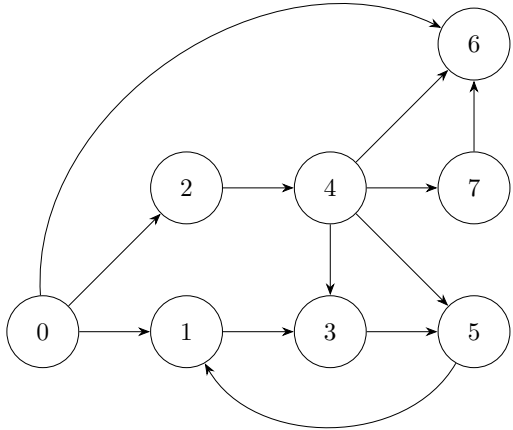
Uma cross edge é uma aresta que liga dois vértices que não compartilham nenhuma relação de ancestralidade. Isto é, são vértices que estão em dois ramos diferentes da árvore de DFS.

DFS: arestas

Back edge

Uma back edge é uma aresta que liga um vértice a um ancestral já descoberto.

DFS: exemplo



S :

DFS: exemplo

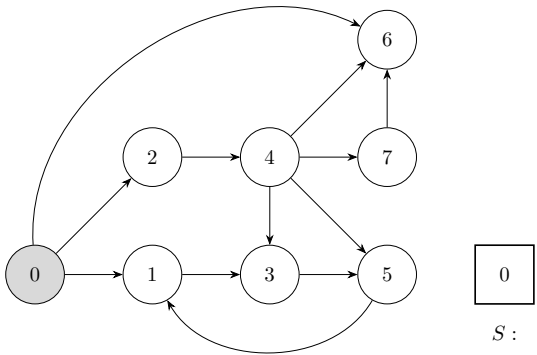


Figura: $(0, 1)$ é uma tree edge.

DFS: exemplo

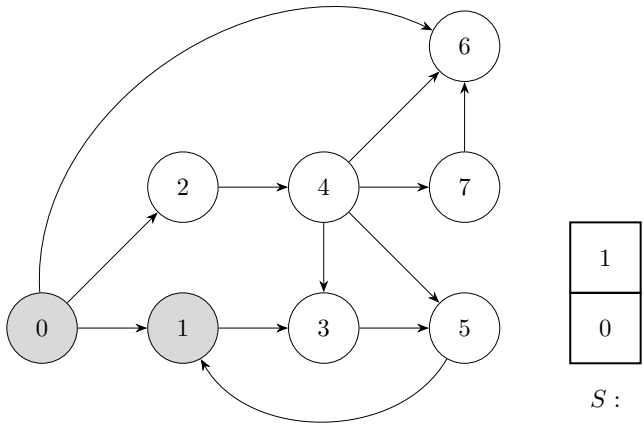


Figura: $(1, 3)$ é uma tree edge

DFS: exemplo

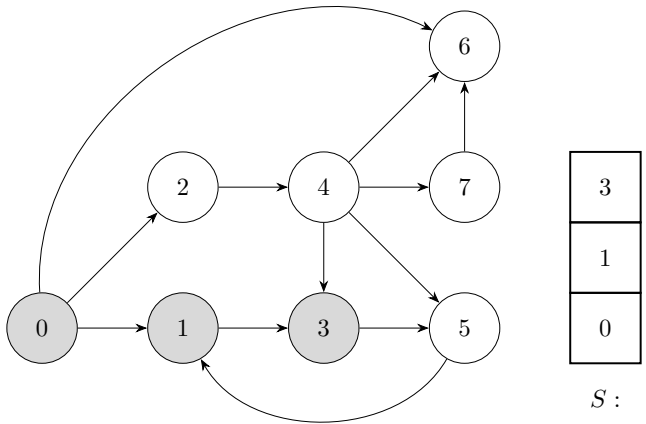


Figura: $(3, 5)$ é uma tree edge.

DFS: exemplo

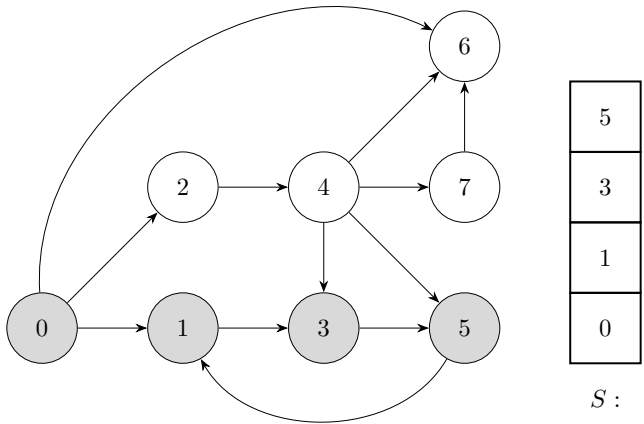
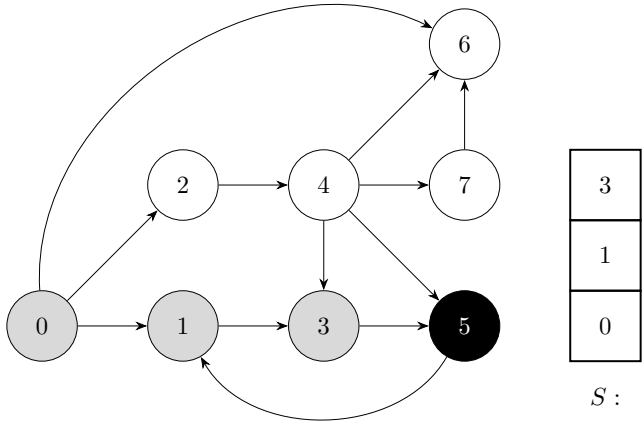
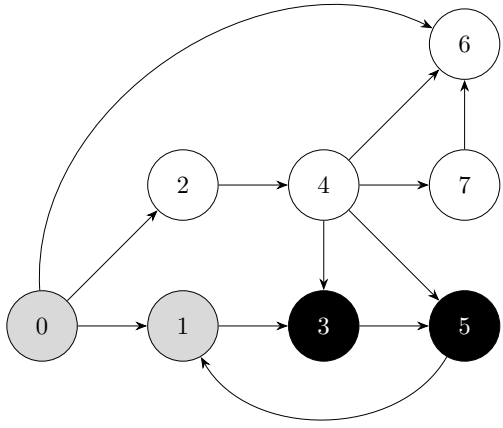


Figura: $(5, 1)$ é uma back edge.

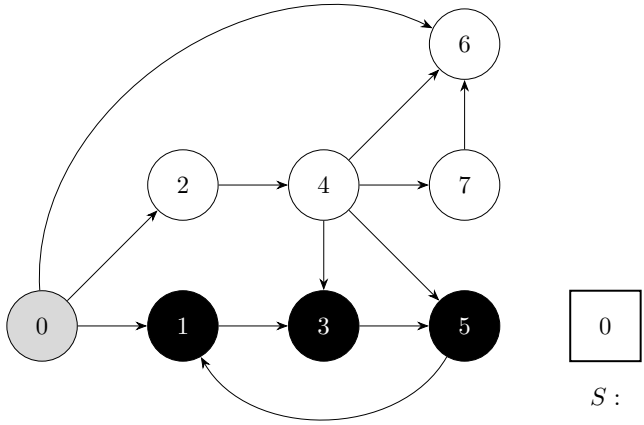
DFS: exemplo



DFS: exemplo



DFS: exemplo



DFS: exemplo

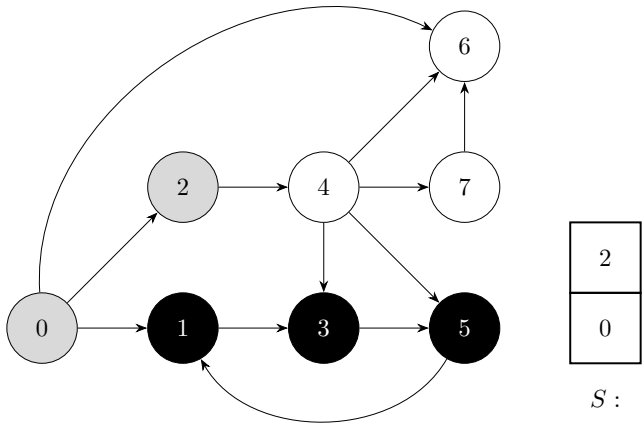


Figura: $(0, 2)$ é uma tree edge.

DFS: exemplo

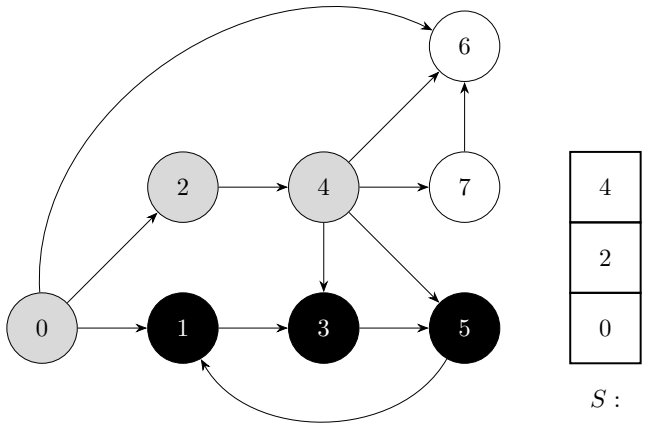


Figura: $(2, 4)$ é uma tree edge.

DFS: exemplo

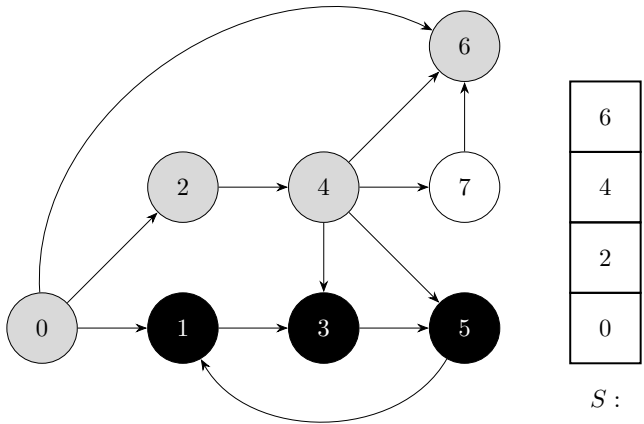
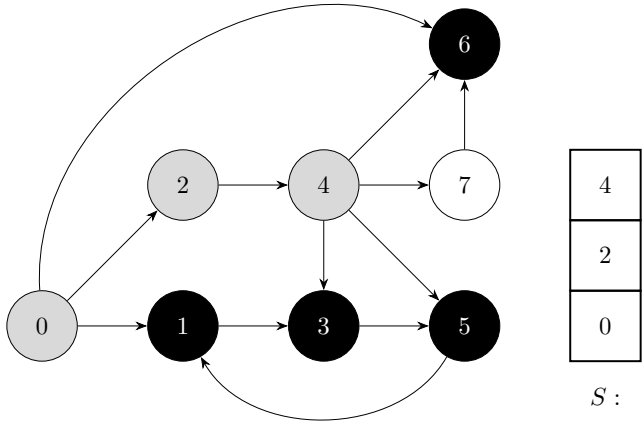


Figura: $(4, 6)$ é uma tree edge.

DFS: exemplo



DFS: exemplo

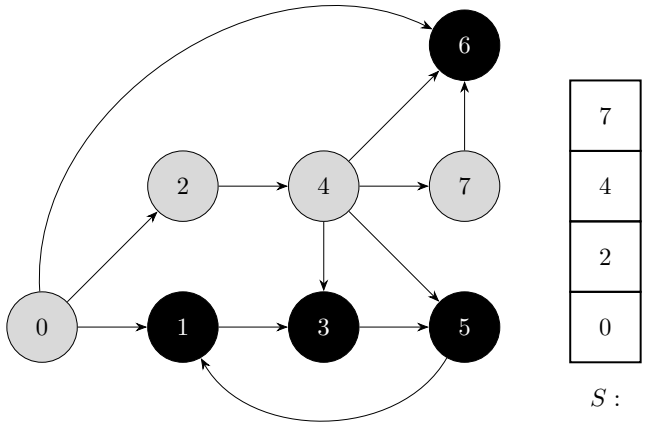


Figura: $(4, 7)$ é uma tree edge.

DFS: exemplo

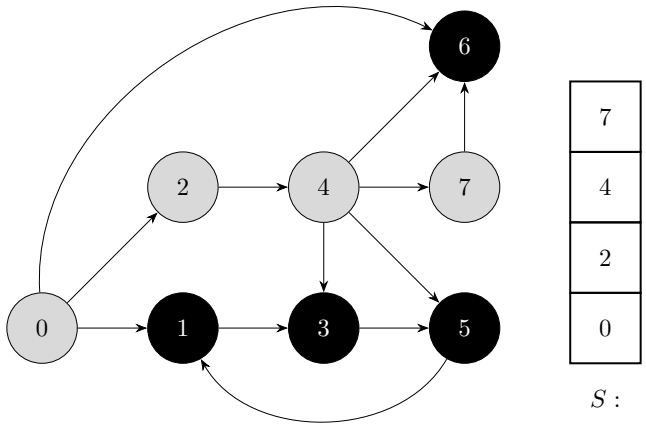
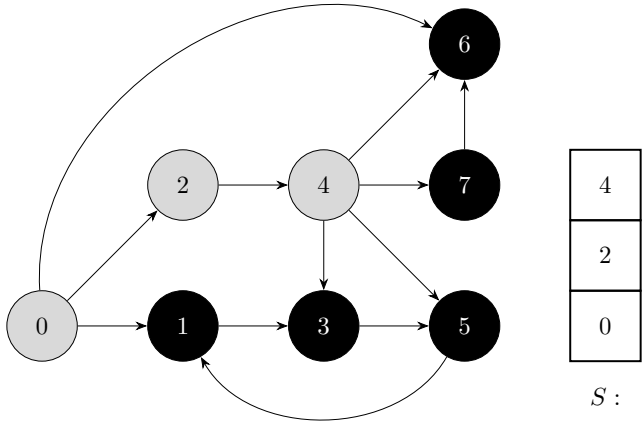


Figura: $(7, 6)$ é uma cross edge.

DFS: exemplo



DFS: exemplo

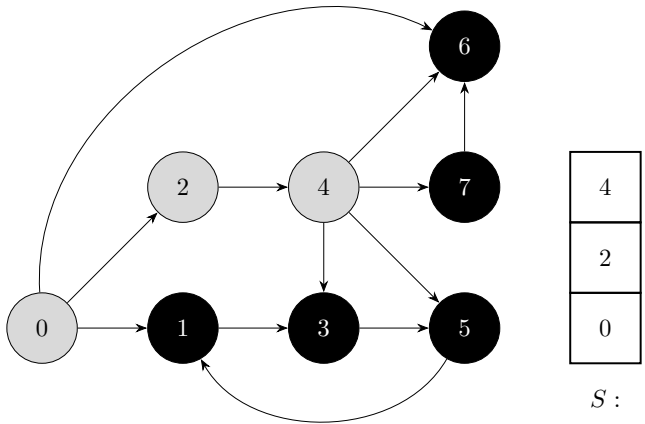


Figura: $(4, 3)$ é uma cross edge.

DFS: exemplo

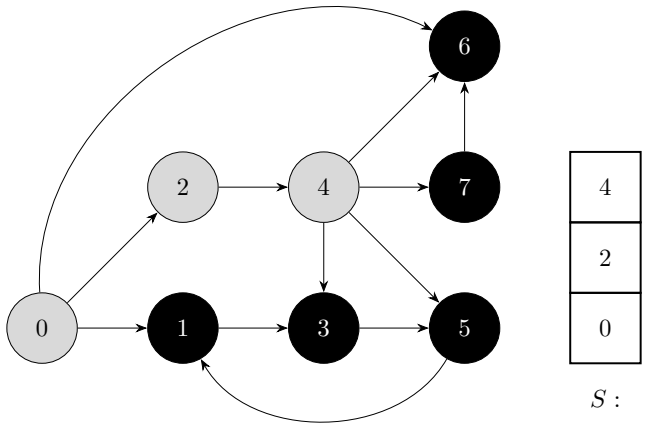
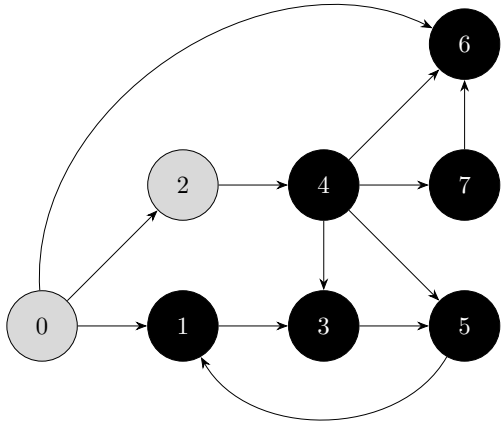


Figura: $(4, 5)$ é uma cross edge.

DFS: exemplo



DFS: exemplo

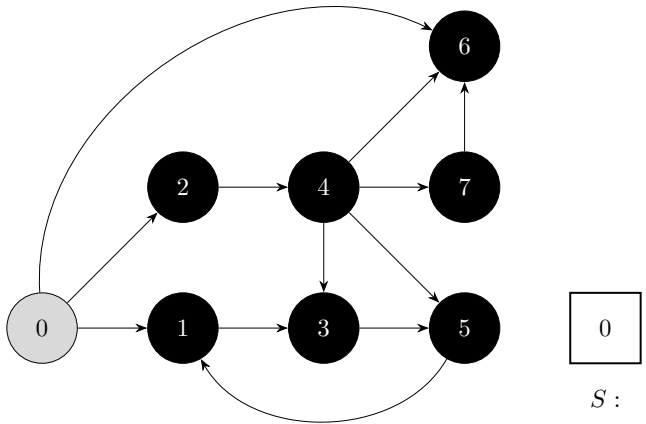
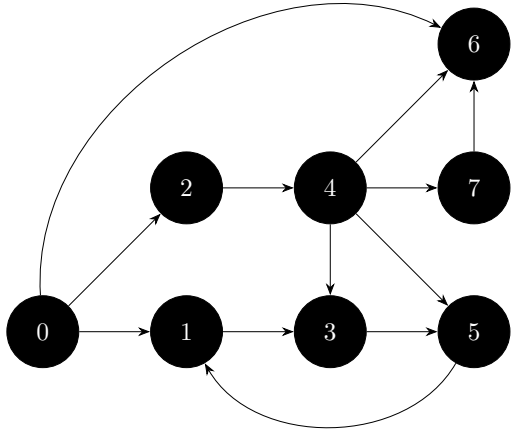


Figura: $(0, 6)$ é uma forward edge.

DFS: exemplo



S :

Sumário

Sumário

Saltos

- Dado um vértice u, o número de saltos é o menor número de arestas para chegar em qualquer vértice v.
- Como determinar, a partir de u, o número de saltos para qualquer $v \in V$?

Saltos: definição

Definição

Número de Saltos Dado um grafo $G = (V, E)$, o número de saltos é definido como:

$$\text{hop}(u, v) = \begin{cases} 0 & u = v \\ 1 + \min\{\text{hop}(w, v) \mid (u, w) \in E\} & u \neq v \\ \infty & \textit{se não existe caminho de } u \textit{ para } v \end{cases}$$

ou, em outras palavras:

$$\text{hop}(u, v) = \begin{cases} \min\{k \mid \langle u, e_1, \ldots, e_k, v\rangle \textit{ é um caminho de } u \textit{ para } v\} \\ \infty, \quad \textit{se não existe caminho de } u \textit{ para } v \end{cases}$$

Saltos

- Naturalmente, a BFS organiza os nós em níveis a partir de uma origem u.
- O nível de um vértice v corresponde a $\mathrm{hop}\,(u, v)$.
- Podemos fazer apenas uma pequena adaptação na BFS. Quando um nó w é colocado na fila, $\mathrm{hop}(u, w) = \mathrm{hop}(u, v) + 1$, em que v é o pai de w na BFS.

Saltos: algoritmo

Algorithm 3: BFS$(G, color, u)$

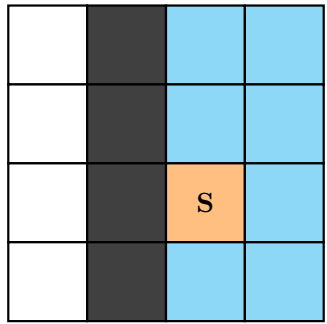
Input: G, o grafo, $color$, o vetor de cores, u o vértice inicial.
Output: $\text{hop}[v] = \text{hop}(u, v)$

```
1  foreach v ∈ V do
2      color[v] ← white
3      hop[v] ← ∞
4  Q ← QUEUE()
5  Q.PUSH(u)
6  hop[u] ← 0
7  color[u] ← gray
8  while ¬Q.EMPTY() do
9      v ← Q.FRONT()
10     Q.POP()
11     foreach (v, w) ∈ E do
12         if color[w] = white then
13             color[w] ← gray
14             Q.PUSH(w)
15             hop[w] ← hop[v] + 1
16     color[v] ← black
17 return hop
```

Sumário

Flood-fill

- O flood-fill, ou preenchimento por inundação, é uma técnica muito utilizada em softwares de manipulação de imagens.
- A ferramenta preenche todo o espaço contido dentro de bordas.



Flood-fill

- Podemos imaginar o grid bidimensional como um grafo.
- Cada célula (i, j) corresponde a um vértice.
- As adjacências dos vértices são os vizinhos daquela célula. Elas serão coloridas se forem um espaço em branco, e não uma borda.
- Política de vizinhança: vizinhança de 4 (vertical e horizontal) ou de 8 (também inclui as diagonais).
- Grafo **incorporado**.

Flood-fill

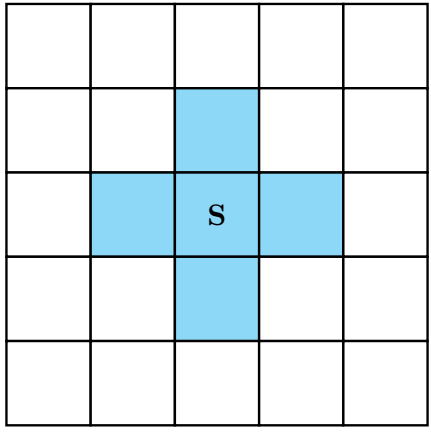


Figura: Vizinhança de 4.

Flood-fill

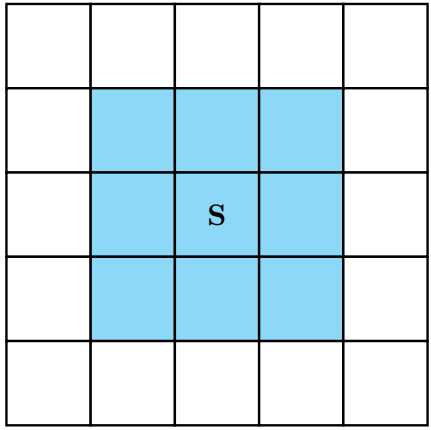


Figura: Vizinhança de 8.

Flood-fill

- Pode-se fazer o preenchimento por largura (BFS) ou por profundidade (DFS).
- A BFS preenche o espaço em "ondas".
- A DFS preenche o espaço em "profundidade".

Flood-fill

- Para o algoritmo, vamos assumir que o grid é representado por uma matriz G de dimensões $n \times m$.
- Cada célula $G[i][j]$ é um vértice do grafo.
- As células que são bordas são marcadas com 1 e as células que são espaços em branco são marcadas com 0.
- O objetivo é preencher todos os vértices que são em branco e que são alcançáveis de um vértice inicial (i, j) sem extrapolar as bordas.

Flood-fill: algoritmo

Algorithm 4: FLOOD-FILL-BFS$(G, color, i, j)$

Input: G, o grid, $color$, a matriz de cores, (i, j) o vértice inicial.

```
1  Q ← QUEUE()
2  Q.PUSH((i, j))
3  color[i][j] ← gray
4  while ¬Q.EMPTY() do
5  |   (a, b) ← Q.FRONT()
6  |   Q.POP()
7  |   N ← NEIGHBORS(G, color, (a, b))
8  |   foreach (c, d) ∈ N do
9  |   |   color[c][d] ← gray
10 |   |   Q.PUSH((c, d))
11 color[i][j] ← black
```

Flood-fill: algoritmo

Algorithm 5: NEIGHBORS(G, i, j)

Input: G, o grid, $color$, a matriz de cores, (i, j) o vértice.

1 $N \leftarrow [\,]$
2 **if** $i > 0$ **and** $G[i-1][j] = 0$ **and** $color[i-1][j] =$ **white then** N.APPEND$((i-1, j))$
3 **if** $i < G$.HEIGHT$() - 1$ **and** $G[i+1][j] = 0$ **and** $color[i+1][j] =$ **white then** N.APPEND$((i+1, j))$
4 **if** $j > 0$ **and** $G[i][j-1] = 0$ **and** $color[i][j-1] =$ **white then** N.APPEND$((i, j-1))$
5 **if** $j < G$.WIDTH$() - 1$ **and** $G[i][j+1] = 0$ **and** $color[i][j+1] =$ **white then** N.APPEND$((i, j+1))$
6 **return** N

Flood-fill: algoritmo

Algorithm 6: FLOOD-FILL-DFS$(G, color, i, j)$

Input: G, o grid, $color$, a matriz de cores, (i, j) o vértice inicial.

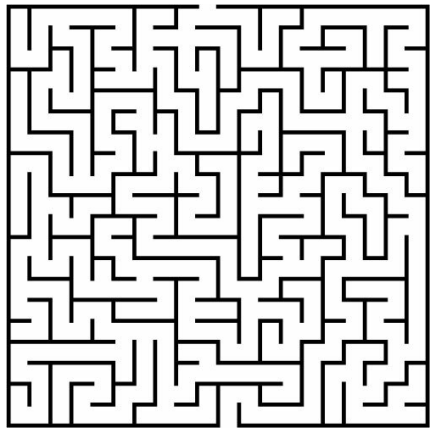
```
1 N ← NEIGHBORS(G, color, (i, j))
2 foreach (a, b) ∈ N do
3     FLOOD-FILL-DFS(G, color, a, b)
4 color[i][j] ← black
```

Flood-fill

- A função $\text{NEIGHBORS}(G, i, j)$ retorna os vizinhos de (i, j) que ainda não foram processados.
- Pode ser implementada de acordo com a política de vizinhança desejada (4 ou 8).

Sumário

Labirintos



Labirintos

- Novamente, podemos ver o labirinto como um grid 2D, em que cada célula é um vértice.
- As paredes são marcadas com 1 enquanto os caminhos são marcados com 0.
- Não se pode atravesar as paredes.
- O objetivo é sair do início e chegar ao fim.

Labirintos

- Se houver apenas uma forma de se chegar ao fim, a BFS e a DFS encontrarão o mesmo caminho.
- Se houver mais de uma forma de se chegar ao fim, a BFS encontrará o caminho com o menor número de saltos.

Labirintos: algoritmo

Algorithm 7: MAZE-SOLVER-BFS$(G, color, a, b, c, d)$

Input: G, o grid, $color$ a matriz de cores, (a, b) o vértice inicial, (c, d) o vértice final.

```
1  Q ← QUEUE()
2  Q.PUSH((a, b))
3  color[a][b] ← gray
4  while ¬Q.EMPTY() do
5      (i, j) ← Q.FRONT()
6      Q.POP()
7      if (i, j) = (c, d) then REPORT-SOLUTION()
8      N ← NEIGHBORS(G, color, (i, j))
9      foreach (i', j') ∈ N do
10         color[i'][j'] ← gray
11         Q.PUSH((i', j'))
12     color[i][j] ← black
```

Labirintos: algoritmo

Algorithm 8: NEIGHBORS$(G, color, (i, j))$

Input: G, o grid, $color$, a matriz de cores, (i, j) o vértice.

```
1  N ← [ ]
2  if i > 0 and G[i − 1][j] = 0 and color[i − 1][j] = white then
3      N.APPEND((i − 1, j))
4  if i < G.HEIGHT() − 1 and G[i + 1][j] = 0 and color[i + 1][j] = white then
5      N.APPEND((i + 1, j))
6  if j > 0 and G[i][j − 1] = 0 and color[i][j − 1] = white then
7      N.APPEND((i, j − 1))
8  if j < G.WIDTH() − 1 and G[i][j + 1] = 0 and color[i][j + 1] = white then
9      N.APPEND((i, j + 1))
10 return N
```

Labirintos: algoritmo

Algorithm 9: MAZE-SOLVER-DFS$(G, color, a, b, c, d)$

Input: G, o grid, $color$ a matriz de cores, (a, b) o vértice inicial, (c, d) o vértice final.

```
1 color[a][b] ← gray
2 if (a, b) = (c, d) then REPORT-SOLUTION()
3 N ← NEIGHBORS(G, color, (a, b))
4 foreach (i, j) ∈ N do
5     MAZE-SOLVER-DFS(G, color, i, j, c, d)
6 color[a][b] ← black
```

Sumário

Ciclos

- Ciclos podem ser detectados através de uma DFS com a presença de back edges.

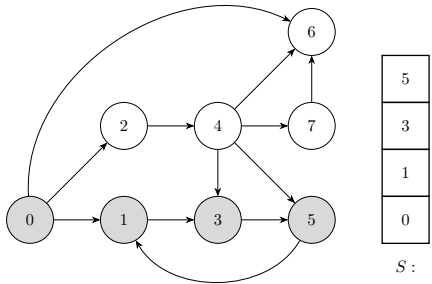


Figura: $(5, 1)$ é uma back edge.

Ciclos: algoritmo

Algorithm 10: DFS-CYCLE$(G, color, u)$

Input: G, o grafo, $color$, o vetor de cores, e u, a raiz da DFS.
Output: **true** se houver um ciclo, **false** caso contrário.

```
1 color[u] ← gray
2 foreach (u, v) ∈ E do
3     if color[v] = white and DFS-CYCLE(G, color, v) then
4         return true
5     else if color[v] = gray then
6         return true
7 color[u] ← black
8 return false
```

Sumário

Grafos bipartidos

- Um grafo é bipartido se seus vértices podem ser particionados em dois conjuntos disjuntos, de modo que todas as arestas conectem vértices de conjuntos diferentes.
- Podemos determinar se um grafo é bipartido através de uma BFS ou DFS.
- A ideia é colorir os vértices com duas cores diferentes, de modo que vértices adjacentes tenham cores diferentes.
- Se conseguirmos colorir o grafo dessa forma, ele é bipartido. Caso contrário, não é.

Grafos bipartidos: algoritmo

Algorithm 11: BIPARTITE-CHECK-BFS$(G, color, u, c)$

Input: G, o grafo, $color$, o vetor de cores,u, o vértice inicial, c a cor de u.
Output: **true** se o grafo for bipartido, **false** caso contrário.

```
1  color[u] ← c
2  Q ← QUEUE()
3  Q.PUSH(u)
4  while ¬Q.EMPTY() do
5      if c = blue then c' ← red
6      else c' ← blue
7      v ← Q.FRONT()
8      Q.POP()
9      foreach (v, w) ∈ E do
10         if color[w] = white then
11             color[w] ← c'
12             Q.PUSH(w)
13         else if color[w] = c then
14             return false
15 return true
```

Grafos bipartidos: algoritmo

Algorithm 12: BIPARTITE-CHECK-DFS$(G, color, u, c)$

Input: G, o grafo, $color$, o vetor de cores, u, a raiz da DFS, c a cor de u.
Output: **true** se o grafo for bipartido, **false** caso contrário.

```
1 color[u] ← c
2 if c = blue then c' ← red
3 else c' ← blue
4 foreach (u, v) ∈ E do
5     if color[v] = white then
6         return BIPARTITE-CHECK-DFS(G, color, v, c')
7     else if color[v] = red then
8         return false
9 return true
```

Sumário

Ordenação topológica

- Em um DAG (grafo acíclico direcionado), uma ordenação topológica é uma ordenação dos vértices, de modo que se existir uma aresta (u, v), u precisa vir antes de v na ordenação.
- Softwares como o *make* utilizam ordenação topológica para determinar a ordem de compilação de arquivos para que todas as dependências sejam respeitadas.
- Gerenciadores de pacotes, como o *apt* e o *dnf*, utilizam ordenação topológica para determinar a ordem de instalação de pacotes de modo a satisfazer todas as dependências.
- A ordenação topológica **não** é única.

Ordenação topológica

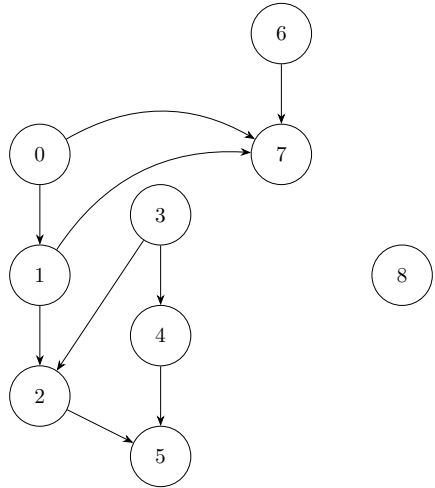
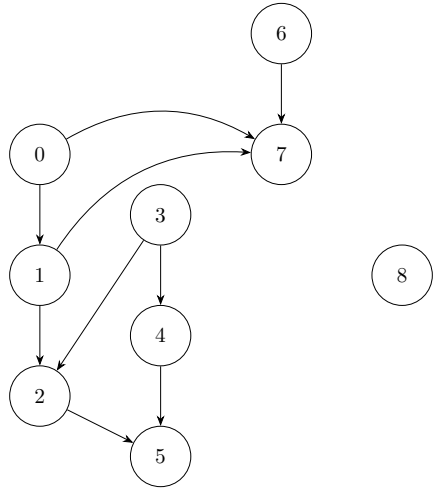


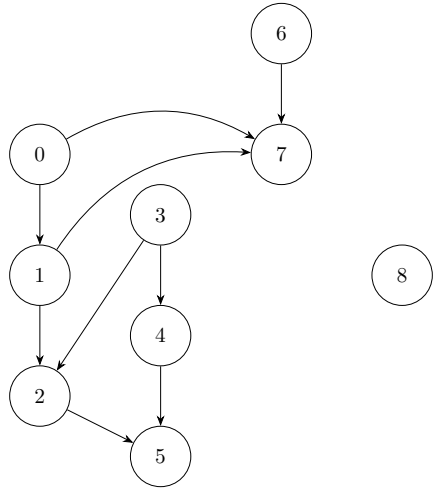
Figura: Exemplo de DAG.

Ordenação topológica



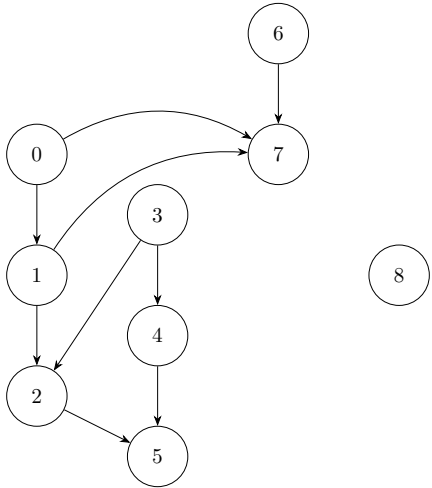
Uma ordenação topológica possível: $\langle 0, 1, 3, 2, 4, 5, 6, 7, 8 \rangle$.

Ordenação topológica



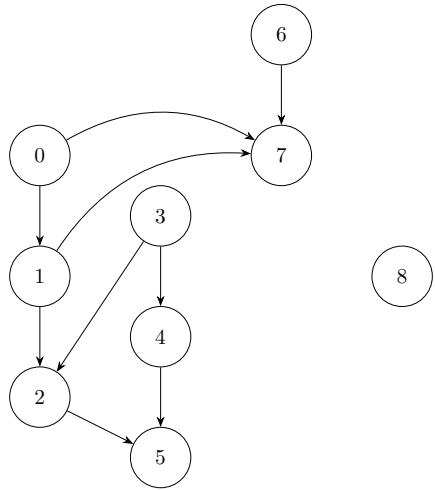
Uma ordenação topológica possível: $\langle 8, 3, 4, 0, 1, 2, 5, 6, 7 \rangle$.

Ordenação topológica



Uma ordenação topológica possível: $\langle 0, 3, 1, 2, 4, 5, 8, 6, 7 \rangle$.

Ordenação topológica



Uma ordenação topológica possível: $\langle 6, 0, 1, 7, 3, 2, 4, 5, 8 \rangle$.

Ordenação topológica

Definição (Fontes em DAGs)

Em um DAG, uma fonte é um vértice v com $\deg^-(v) = 0$.

Ordenação topológica

Definição (Sorvedouro em DAGs)

Em um DAG, um sorvedouro é um vértice v com $\deg^+(v) = 0$.

Ordenação topológica

Teorema

Todo DAG possui pelo menos uma fonte.

Ordenação topológica

Demonstração

Suponha que um DAG G não possua fontes. Então, todo vértice $v \in V$ possui $\deg^-(v) > 0$.

Escolha um vértice $v_0 \in V$. Como $\deg^-(v_0) > 0$, existe um vértice v_1 tal que $(v_1, v_0) \in E$. Como $\deg^-(v_1) > 0$, existe um vértice v_2 tal que $(v_2, v_1) \in E$. Continuando esse processo, obtemos uma sequência de vértices $v_0, v_1, v_2, \ldots$ tal que $(v_{i+1}, v_i) \in E$ para todo $i \geq 0$. Como o grafo é finito, essa sequência deve eventualmente repetir algum vértice, formando um ciclo. Isso contradiz a suposição de que G é um DAG. Portanto, todo DAG possui pelo menos uma fonte.

□

Ordenação topológica

Teorema

Todo DAG possui pelo menos um sorvedouro.

Ordenação topológica

Demonstração

Suponha que um DAG G não possua sorvedouros. Então, todo vértice $v \in V$ possui $\deg^+(v) > 0$.

Escolha um vértice $v_0 \in V$. Como $\deg^+(v_0) > 0$, *existe um vértice v_1 tal que* $(v_0, v_1) \in E$. *Como* $\deg^+(v_1) > 0$, *existe um vértice v_2 tal que* $(v_1, v_2) \in E$. *Continuando esse processo, obtemos uma sequência de vértices $v_0, v_1, v_2, \ldots$ tal que* $(v_i, v_{i+1}) \in E$ *para todo* $i \geq 0$. *Como o grafo é finito, essa sequência deve eventualmente repetir algum vértice, formando um ciclo. Isso contradiz a suposição de que G é um DAG. Portanto, todo DAG possui pelo menos um sorvedouro.*

□

Ordenação topológica

- Dada a presença de fontes e sorvedouros, podemos construir uma ordenação topológica de um DAG a partir do algoritmo de Kahn.
- A ideia dele é simples: enquanto houver fontes, remova-as do grafo e adicione-as à ordenação topológica.
- A cada remoção de uma fonte, novos vértices podem se tornar fontes.
- O algoritmo termina quando não houver mais fontes. Se o grafo ainda tiver vértices, então ele possui ciclos e não é um DAG. Caso contrário, a ordenação topológica foi construída com sucesso.

Ordenação topológica: algoritmo de Kahn

Algorithm 13: TOP-SORT-KAHN(G, deg)

Input: G, o DAG, deg o vetor de graus de entrada.
Output: L, uma ordenação topológica de G

```
1  L ← [ ]
2  Q ← QUEUE()
3  foreach u ∈ V do
4      if deg[u] = 0 then
5          Q.PUSH(u)
6  while not Q.EMPTY() do
7      u ← Q.FRONT()
8      Q.POP()
9      L.APPEND(u)
10     foreach (u, v) ∈ E do
11         deg[v] ← deg[v] − 1
12         if deg[v] = 0 then
13             Q.PUSH(v)
14 return L
```

Algoritmo de Kahn: análise

Análise de pior-caso

Cada vértice só pode entrar na fila uma vez, e cada aresta só pode ser processada uma vez. Portanto, o tempo de execução é $\Theta(n + m)$.

Ordenação topológica: DFS

- Também é possível construir uma ordenação topológica a partir de uma DFS.
- A ideia é simples: a cada vez que um vértice é finalizado, adicione-o ao final do vetor ordenação topológica.
- A cada finalização de um vértice, todos os seus descendentes já foram finalizados, garantindo que a ordenação topológica seja respeitada.
- Ao final do processo, inverta o vetor ordenação topológica para obter a ordenação correta.
- Também é feito em tempo $\Theta(n + m)$ com listas de adjacências.

Ordenação topológica: DFS

Algorithm 14: TOP-SORT-DFS$(G, color, u, L)$

Input: G, o grafo, $color$, o vetor de cores, u, o vértice atual, L, o vetor de ordenação topológica.

```
1 color[u] ← gray
2 foreach (u, v) ∈ E do
3     if color[v] = white then
4         TOP-SORT-DFS(G, color, v, L)
5 color[u] ← black
6 L.APPEND(u)
```

Ordenação topológica: DFS

Algorithm 15: TOP-SORT-DFS-MAIN(G)

Input: G, o grafo.
Output: L, uma ordenação topológica de G

```
1 L ← [ ]
2 foreach u ∈ V do
3     color[u] ← white
4 foreach u ∈ V do
5     if color[u] = white then
6         TOP-SORT-DFS(G, color, u, L)
7 REVERSE(L)
8 return L
```

Sumário

Componentes conexas

- Tome o problema de determinar as componentes conexas de um grafo não direcionado.
- Isto é, devemos retornar conjuntos de vértices $C_1, C_2, \ldots, C_k$ tais que:

$$\bigcup_{i=1}^{k} C_i = V \quad \text{e} \quad C_i \cap C_j = \emptyset, \forall i \neq j$$

- Podemos resolver este problema utilizando uma DFS ou BFS. A ideia é simples: enquanto houver vértices não visitados, execute uma DFS ou BFS a partir de um vértice não visitado, incluindo todos os vértices alcançados na mesma componente conexa.

Componentes conexas: algoritmo

Algorithm 16: CONNECTED-COMPONENTS-DFS(G)

Input: G, o grafo.
Output: C, um vetor de componentes conexas.

```
1 C ← [ ]
2 foreach u ∈ V do
3     color[u] ← white
4 foreach u ∈ V do
5     if color[u] = white then
6         C_i ← [ ]
7         DFS-CC(G, color, u, C_i)
8         C.APPEND(C_i)
9 return C
```

Componentes conexas: algoritmo

Algorithm 17: BFS-CC$(G, color, u, C_i)$

Input: G, o grafo, $color$, o vetor de cores, u, a raiz da DFS, C_i, a componente conexa atual.

```
1 color[u] ← gray
2 C_i.APPEND(u)
3 foreach (u, v) ∈ E do
4     if color[v] = white then
5         DFS-CC(G, color, v, C_i)
6 color[u] ← black
```

Componentes conexas: algoritmo

Algorithm 18: CONNECTED-COMPONENTS-DFS(G)

Input: G, o grafo.
Output: C, um vetor de componentes conexas.

```
1 C ← [ ]
2 foreach u ∈ V do
3     color[u] ← white
4 foreach u ∈ V do
5     if color[u] = white then
6         C_i ← [ ]
7         BFS-CC(G, color, u, C_i)
8         C.APPEND(C_i)
9 return C
```

Componentes conexas: algoritmo

Algorithm 19: BFS-CC$(G, color, u, C_i)$

Input: G, o grafo, $color$, o vetor de cores, u, o vértice inicial, C_i, a componente conexa atual.

```
1  Q ← QUEUE()
2  Q.PUSH(u)
3  color[u] ← gray
4  while ¬Q.EMPTY() do
5      v ← Q.FRONT()
6      Q.POP()
7      C_i.APPEND(v)
8      foreach (v, w) ∈ E do
9          if color[w] = white then
10             color[w] ← gray
11             Q.PUSH(w)
12     color[v] ← black
```

Sumário

Componentes fortemente conexas

- Uma componente fortemente conexa em um grafo direcionado permite que qualquer vértice da componente seja alcançado a partir de qualquer outro vértice da mesma componente.
- Em outras palavras, para qualquer componente com mais de um vértice, eles devem estar envolvidos em um ciclo.
- Chave para detecção de componentes fortemente conexas é a utilização de DFS e a análise das back edges e tree edges.

Componentes fortemente conexas

Ideia do algoritmo

Manteremos duas informações para cada vértice u:

- $index[u]$: a ordem em que o vértice u foi visitado na DFS.
- $low[u]$: o menor índice de qualquer vértice alcançável a partir de u por meio de uma back edge.

Os vértices v que possuem $low[v] \leq index[v]$ pertencem à mesma componente fortemente conexa de $low[v]$.

Componentes fortemente conexas: algoritmo

Algorithm 20: STRONGLY-CONNECTED-COMPONENTS(G)

Input: G, o grafo.
Output: C, um vetor de componentes fortemente conexas.

```
1  C ← [ ]
2  foreach u ∈ V do
3      color[u] ← white
4      index[u] ← −1
5      low[u] ← −1
6  foreach u ∈ V do
7      idx ← 0
8      if color[u] = white then
9          S ← STACK()
10         C_i ← [ ]
11         DFS-SCC(G, color, u, index, low, idx, C_i, S)
12         C.APPEND(C_i)
13 return C
```

Componentes fortemente conexas: algoritmo

Algorithm 21: DFS-SCC($G, color, u, index, low, \&idx, C_i$)

Input: G, o grafo, $color$, o vetor de cores, u, o vértice atual, $index$, o vetor de índices, low, o vetor de low, idx, o índice atual, C_i, a componente fortemente conexa atual, S, a pilha de vértices.

```
1  index[u] ← idx
2  low[u] ← idx
3  idx ← idx + 1
4  color[u] ← gray
5  S.PUSH(u)
6  foreach (u, v) ∈ E do
7      if color[v] = white then
8          DFS-SCC(G, color, v, index, idx, C_i, S)
9          low[u] ← min(low[u], low[v])
10     else if color[v] = gray then
11         low[u] ← min(low[u], index[v])
12 if low[u] = index[u] then
13     C_i ← CREATE-COMPONENT(S, color, u)
```

Componentes fortemente conexas: algoritmo

Algorithm 22: CREATE-COMPONENT$(S, color, u)$

Input: S, a pilha de vértices, $color$, o vetor de cores, u, o vértice atual.
Output: C_i, a componente fortemente conexa atual.

```
1 C_i ← [ ]
2 w ← ⊥
3 repeat
4 |   w ← S.TOP()
5 |   S.POP()
6 |   C_i.APPEND(w)
7 |   color[w] ← black
8 until w ≠ u
9 return C_i
```

Sumário

Vértices de articulação

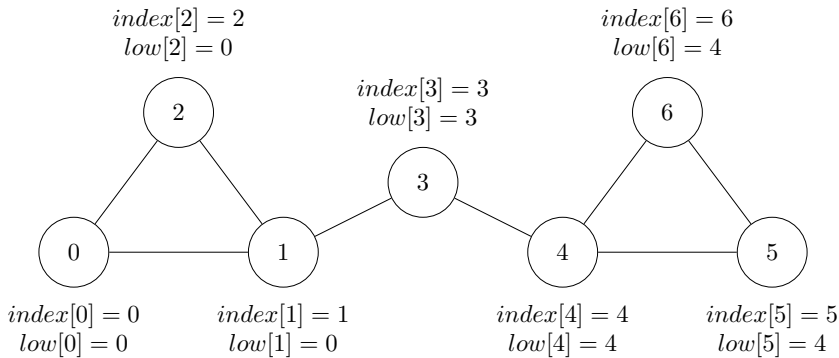
- Um vértice de articulação é aquele que, se removido, aumenta o número de componentes conexas de um grafo.
- Podemos identificar vértices de articulação em tempo linear usando DFS.
- A ideia é similar ao algoritmo de detecção de componentes fortemente conexas conexas.
- Seja u o predecessor de um nó v na DFS. Se $low[v] \geq index[u]$, então quer dizer que não é possível alcançar um ancestral de u por meio de uma back edge. Logo u é um vértice de articulação.

Vértices de articulação

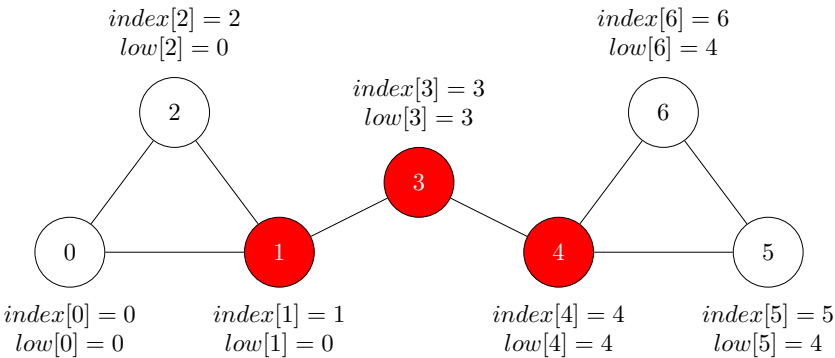
Ideia do Algoritmo

- Computar $index[v]$ e $low[v]$.
- Se u é predecessor de v, isto é $(u, v) \in E$ é uma tree edge na DFS, e $low[v] \geq index[u]$, então u é um vértice de articulação.
- **Exceção:** a raiz original da DFS é um vértice de articulação se possui dois ou mais filhos na árvore de busca em profundidade (tree edges).

Vértices de articulação



Vértices de articulação



Vértices de articulação: algoritmo

Algorithm 23: ARTICULATION-POINTS(G)

Input: G, o grafo.
Output: A, um vetor de vértices de articulação.

```
1  A ← [ ]
2  foreach u ∈ V do
3      color[u] ← white
4      index[u] ← −1
5      low[u] ← −1
6      parent[u] ← −1
7  idx ← 0
8  foreach u ∈ V do
9      if color[u] = white then
10         DFS-ARTICULATION(G, color, u, index, low, parent, idx, A)
11 return A
```

Vértices de articulação: algoritmo

Algorithm 24: DFS-ARTICULATION$(G, color, u, index, low, parent, \&idx, A)$

Input: G, o grafo, $color$, o vetor de cores, u, o vértice atual, $index$, o vetor de índices, low, o vetor de low, $parent$, o vetor de predecessores, idx, o índice atual, A, o vetor de vértices de articulação.

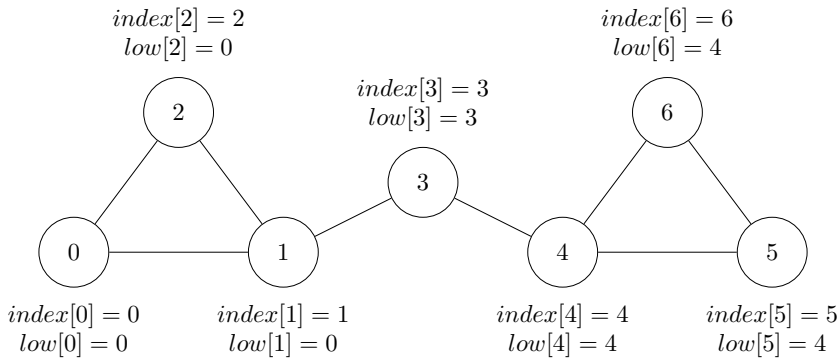
```
1  index[u] ← idx
2  low[u] ← idx
3  idx ← idx + 1
4  color[u] ← gray
5  children ← 0
6  foreach (u, v) ∈ E do
7      if color[v] = white then
8          parent[v] ← u
9          children ← children + 1
10         DFS-ARTICULATION(G, color, v, index, low, parent, idx, A)
11         low[u] ← min(low[u], low[v])
12         if parent[u] ≠ −1 and low[v] ≥ index[u] then
13             A.APPEND(u)
14     else if v ≠ parent[u] then
15         low[u] ← min(low[u], index[v])
16 if parent[u] = −1 and children > 1 then
17     A.APPEND(u)
18 color[u] ← black
```

Sumário

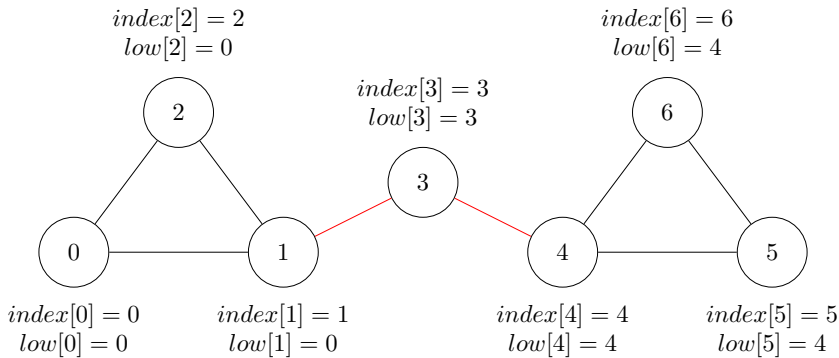
Pontes

- Pontes são arestas que, se removidas, aumentam o número de componentes conexas de um grafo.
- Para identificar pontes, podemos utilizar uma abordagem similar à de vértices de articulação.
- Se u é predecessor de v na árvore de DFS e $low[v] > index[u]$, então (u, v) é uma ponte.

Vértices de articulação



Vértices de articulação



Pontes: algoritmo

Algorithm 25: BRIDGES(G)

Input: G, o grafo.
Output: B, um vetor de pontes.

```
1  B ← [ ]
2  foreach u ∈ V do
3      color[u] ← white
4      index[u] ← −1
5      low[u] ← −1
6      parent[u] ← −1
7  idx ← 0
8  foreach u ∈ V do
9      if color[u] = white then
10         DFS-BRIDGES(G, color, u, index, low, parent, idx, B)
11 return B
```

Pontes: algoritmo

Algorithm 26: DFS-BRIDGES$(G, color, u, index, low, parent, \&idx, B)$

Input: G, o grafo, $color$, o vetor de cores, u, o vértice atual, $index$, o vetor de índices, low, o vetor de low, $parent$, o vetor de predecessores, idx, o índice atual, B, o vetor de pontes.

```
1  index[u] ← idx
2  low[u] ← idx
3  idx ← idx + 1
4  color[u] ← gray
5  foreach (u, v) ∈ E do
6      if color[v] = white then
7          parent[v] ← u
8          DFS-BRIDGES(G, color, v, index, low, parent, idx, B)
9          low[u] ← min(low[u], low[v])
10         if low[v] > index[u] then
11             B.APPEND((u, v))
12     else if v ≠ parent[u] then
13         low[u] ← min(low[u], index[v])
14 color[u] ← black
```

Sumário

Busca exaustiva

- O processo de busca exaustiva envolve explorar todas as possibilidades de um problema para encontrar uma solução.
- Em grafos, isso geralmente significa explorar todos os caminhos possíveis a partir de um vértice inicial até encontrar um vértice alvo ou até que todas as possibilidades tenham sido exploradas.
- Algoritmos de busca exaustiva podem ser implementados usando técnicas como backtracking, DFS ou BFS, dependendo da natureza do problema e das restrições impostas.