ACH2024

Aula 7 – Grafos:
Busca em Profundidade (parte 1)

Profa. Ariane Machado Lima

Aulas anteriores...

- Conceitos básicos de grafos
- Duas formas de implementar uma estrutura de dados (e rotinas BÁSICAS de manipulação:
  - Matriz de adjacência
  - Lista de adjacência

Daqui em diante

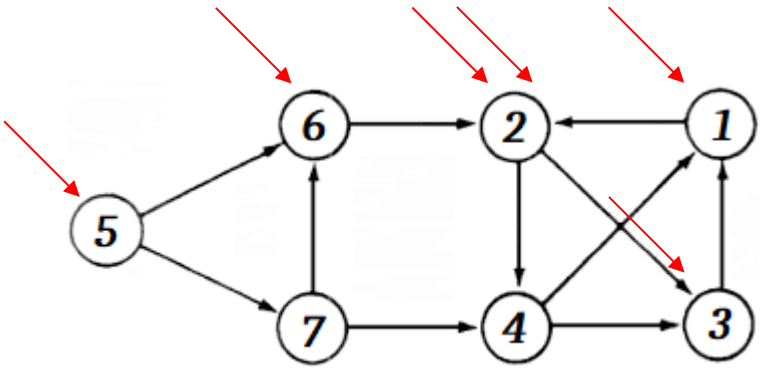
- Algoritmos gerais para grafos (independente de qual implementação de estrutura de dados) e aplicações
  - Formas de percorrer o grafo
    - Busca em profundidade
    - Busca em largura
  - Árvore geradora mínima
  - Caminhos mínimos

Daqui em diante

- Algoritmos gerais para grafos (independente de qual implementação de estrutura de dados) e aplicações
  - Formas de percorrer o grafo
    - Busca em profundidade
    - Busca em largura
  - Árvore geradora mínima
  - Caminhos mínimos

Aula passada

- Idéia da busca em profundidade



Busca em Profundidade

- A busca em profundidade, do inglês *depth-first search*), é um algoritmo para caminhar no grafo.
- A estratégia é buscar o mais profundo no grafo sempre que possível.
- As arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda possui arestas não exploradas saindo dele.
- Quando todas as arestas adjacentes a v tiverem sido exploradas a busca anda para trás para explorar vértices que saem do vértice do qual v foi descoberto. (antecessor de v)
- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes fortemente conectados.

Busca em Profundidade

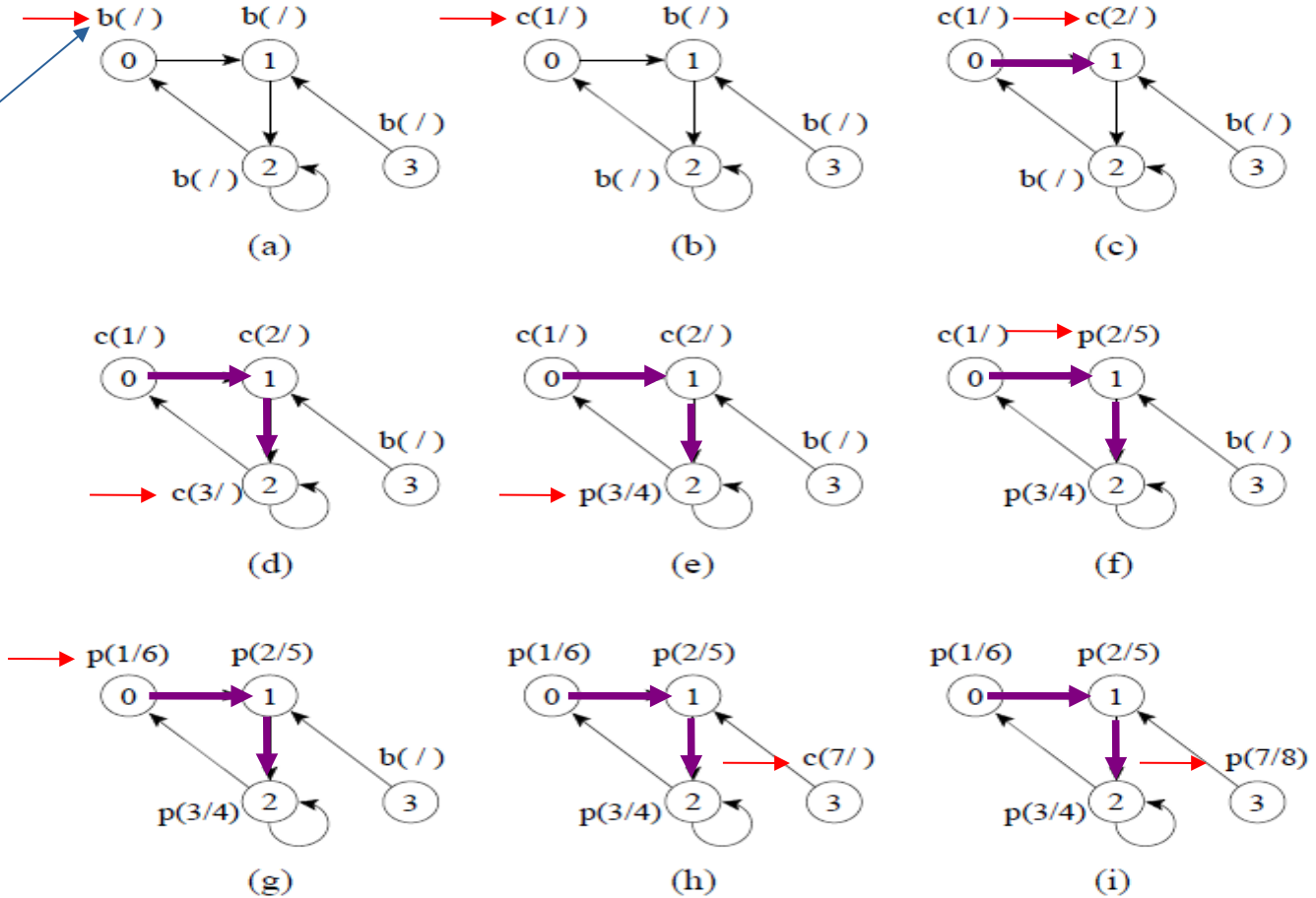
- Para acompanhar o progresso do algoritmo cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é *descoberto* pela primeira vez ele torna-se cinza, e é tornado preto quando sua lista de adjacentes tenha sido completamente examinada.
- $d[v]$: tempo de descoberta
- $t[v]$: tempo de término do exame da lista de adjacentes de v.

Medidores de tempo: úteis para
- acompanhar a evolução da busca
- utilizados em vários algoritmos de grafos

- Estes registros são inteiros entre 1 e $2|V|$ pois existe um evento de descoberta e um evento de término para cada um dos $|V|$ vértices.

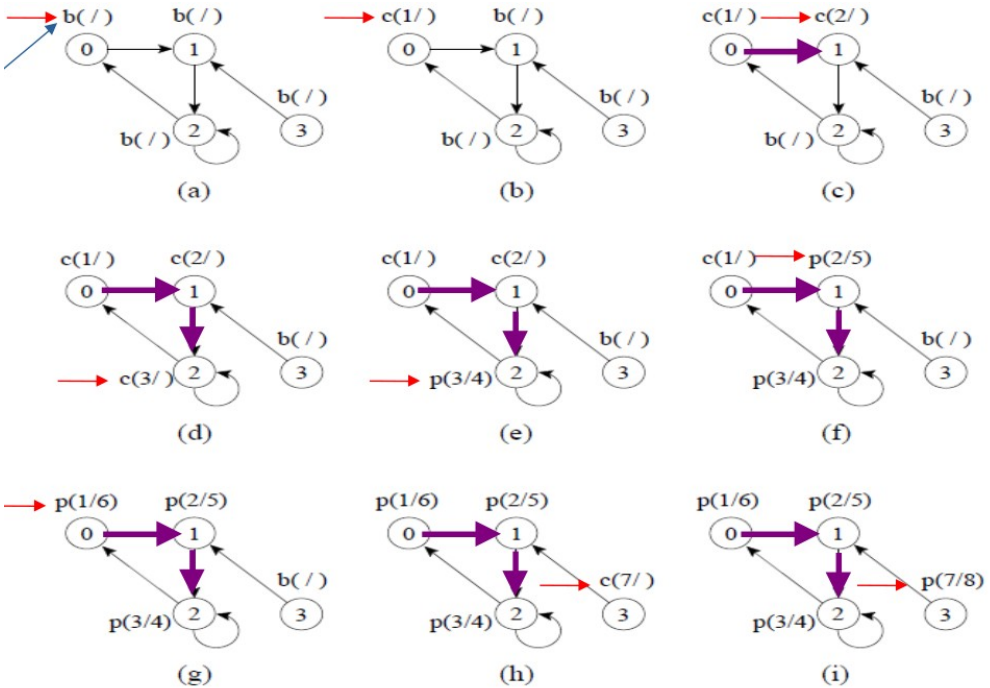
Busca em Profundidade: Exemplo

Cada vértice tem:
cor(d[v],t[v]) e
antecessor (→)



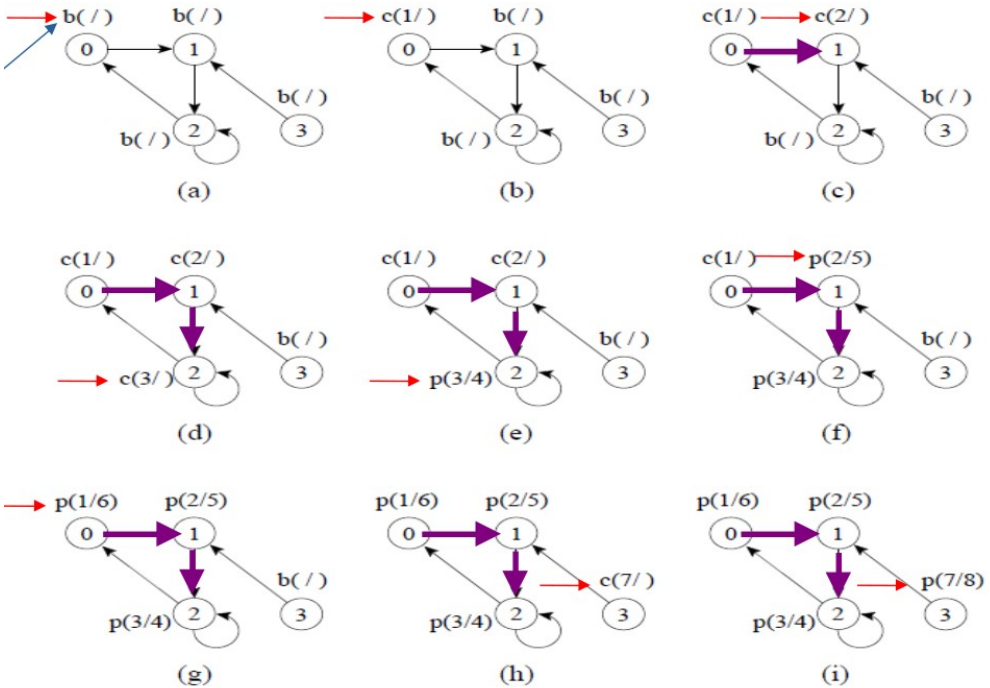
Busca em Profundidade: Implementação

Pseudocódigo baseado na interface (ou seja, sem se basear na implementação por matriz ou listas de adjacências)



Busca em Profundidade: Implementação

```
buscaProfundidade(grafo){
    Aloca vetores cor, tdesc, tterm, antecessor com tamanho grafo->nrVertices
     tempo ← 0;
     Para cada vertice v
          cor[v] ← branco;  tdesc[v] = tterm[v] = 0;  antecessor[v] ← -1;
```



Busca em Profundidade: Implementação

```
buscaProfundidade(grafo){
      Aloca vetores cor, tdesc, tterm, antecessor com tamanho grafo->nrVertices
       tempo ← 0;
       Para cada vertice v
              cor[v] ← branco;  tdesc[v] = tterm[v] = 0;  antecessor[v] ← -1;
       Para cada vertice v
              Se cor[v] = branco visitaBP(v, grafo, &tempo, cor, tdesc, tterm, antecessor);
}


visitaBP(v, grafo, tempo, cor, tdesc, tterm, antecessor){
       cor[v] ← cinza; tdesc[v] ← ++(*tempo);
       Para cada vertice u da lista de adjacência de v
              Se u é branco
                     antecessor[u] ← v;
                     visitaBP(u, grafo, &tempo, cor, tdesc, tterm, antecessor);
       tterm[v] ← ++(*tempo);
       cor[v] ← preto;
}
```

Busca em Profundidade: Implementação

```
buscaProfundidade(grafo){
      Aloca vetores cor, tdesc, tterm, antecessor com tamanho grafo->nrVertices
       tempo ← 0;
       Para cada vertice v
              cor[v] ← branco;  tdesc[v] = tterm[v] = 0;  antecessor[v] ← -1;
       Para cada vertice v
              Se cor[v] = branco visitaBP(v, grafo, &tempo, cor, tdesc, tterm, antecessor);
}


visitaBP(v, grafo, tempo, cor, tdesc, tterm, antecessor){
       cor[v] ← cinza; tdesc[v] ← ++(*tempo);
       Para cada vertice u da lista de adjacência de v
              Se u é branco
                     antecessor[u] ← v;
                     visitaBP(u, grafo, &tempo, cor, tdesc, tterm, antecessor);
       tterm[v] ← ++(*tempo);
       cor[v] ← preto;
}
```

Complexidade ?

Busca em Profundidade: Análise

- Os dois anéis da *BuscaEmProfundidade* têm custo $O(|V|)$ cada um, a menos da chamada do procedimento *VisitaDfs*(u) no segundo anel.
- O procedimento *VisitaDfs* é chamado exatamente uma vez para cada vértice $u \in V$, desde que *VisitaDfs* é chamado apenas para vértices brancos e a primeira ação é pintar o vértice de cinza.
- Durante a execução de *VisitaDfs*(u) o anel principal é executado $|Adj[u]|$ vezes.
- Desde que $\sum_{u \in V} |Adj[u]| = O(|A|)$, o tempo total de execução de *VisitaDfs* é $O(|A|)$.
- Logo, a complexidade total da *BuscaEmProfundidade* é $O(|V| + |A|)$.

Linear no tamanho total do grafo

Busca em Profundidade: Implementação

```
buscaProfundidade(grafo){
      Aloca vetores cor, tdesc, tterm, antecessor com tamanho grafo->nrVertices
       tempo ← 0;
       Para cada vertice v
              cor[v] ← branco;  tdesc[v] = tterm[v] = 0;  antecessor[v] ← -1;
       Para cada vertice v
              Se cor[v] = branco visitaBP(v, grafo, &tempo, cor, tdesc, tterm, antecessor);
}


visitaBP(v, grafo, tempo, cor, tdesc, tterm, antecessor){
      cor[v] ← cinza; tdesc[v] ← ++(*tempo);
      Para cada vertice u da lista de adjacência de v
              Se u é branco
                     antecessor[u] ← v;
                     visitaBP(u, grafo, &tempo, cor, tdesc, tterm, antecessor);
      tterm[v] ← ++(*tempo);
      cor[v] ← preto;
}
```

Pseudocódigo baseado na interface (ou seja, sem se basear na implementação por matriz ou listas de adjacências)

Busca em Profundidade: Análise

Essa complexidade vale para as duas implementações de grafos?
Se não:

- qual a complexidade para matrizes e listas de adjacência?

Busca em Profundidade: Análise

Essa complexidade vale para as duas implementações de grafos?
Se não:

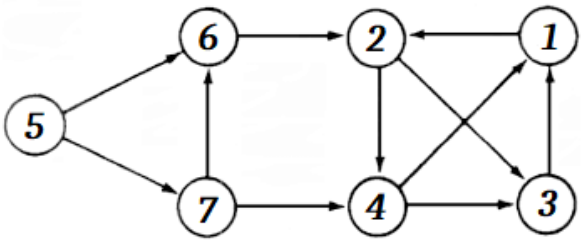
- qual a complexidade para matrizes e listas de adjacência?
  - $O(V^2)$ para matrizes, pois para cada vértice percorrer sua lista de adjacência custa O(V)
- então deve-se sempre usar lista de adjacência? Se não, por quê?

Busca em Profundidade: Análise

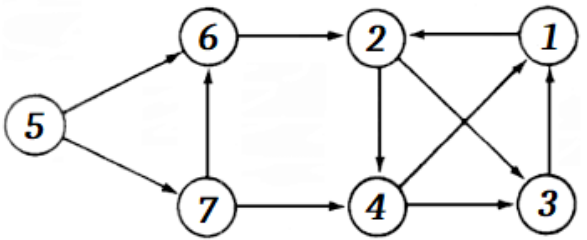
Essa complexidade vale para as duas implementações de grafos?
Se não:

- qual a complexidade para matrizes e listas de adjacência?
  - $O(V^2)$ para matrizes, pois para cada vértice percorrer sua lista de adjacência custa O(V)
- então deve-se sempre usar lista de adjacência? Se não, por quê?
  - 1) se está usando matriz, provavelmente o grafo é denso, e portanto A = $O(V^2)$ mesmo
  - 2) depende de quais operações quer mais velocidade

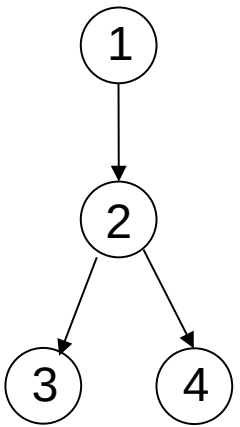
Qual a ordem em que os nós são descobertos durante a busca em profundidade?

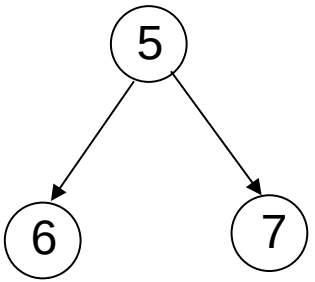


Qual a ordem em que os nós são descobertos durante a busca em profundidade?



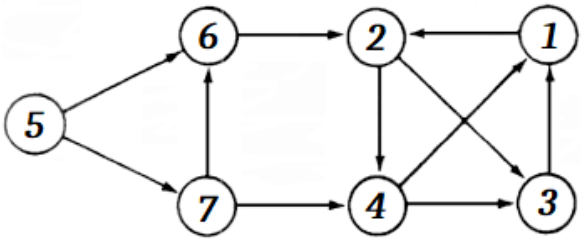
Ordem de descoberta



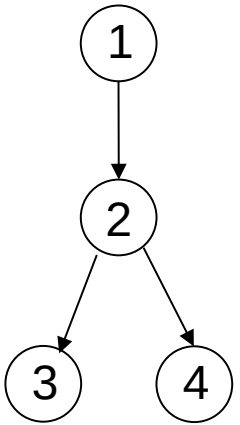


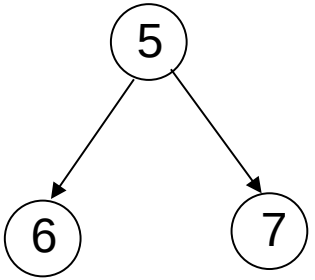
Exemplos no caso de usar matriz de adjacência ou se os vértices estiverem ordenados nas listas de adjacências (por quê?)

Qual a ordem em que os nós são descobertos durante a busca em profundidade?



Ordem de descoberta





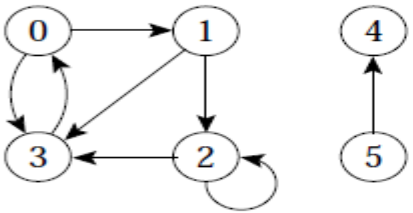
Árvores de busca em profundidade do grafo

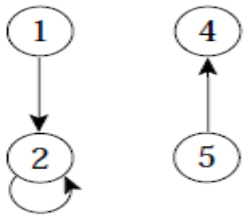
Investigações nessas árvores nos dão várias informações…
Para isso vamos ver alguns conceitos

Abre parêntesis para mais alguns conceitos básicos de grafos...

Subgrafos

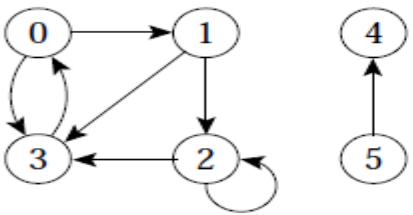
- Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.

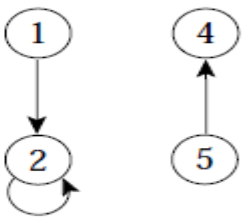




Subgrafos

- Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.

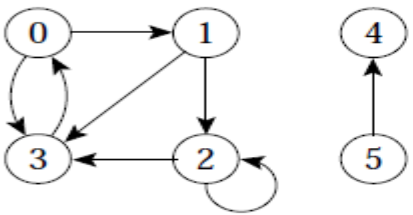


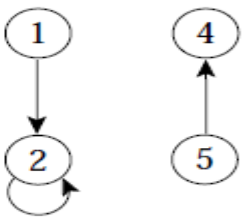


Um subgrafo pode ter uma aresta sem uma das pontas?

Subgrafos

- Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.

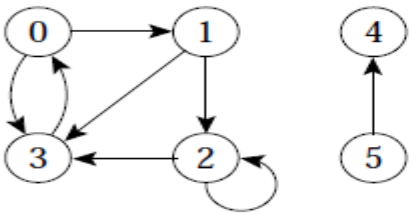


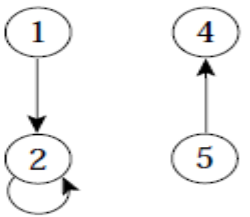


Um subgrafo pode ter uma aresta sem uma das pontas?
NÃO! Se não não seria um grafo! (lembre da definição de aresta e de grafo)

Subgrafos

- Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.

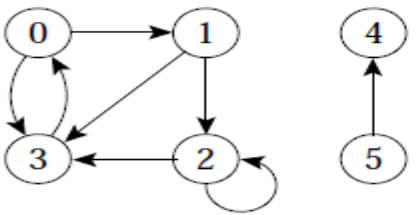


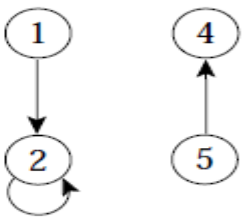


Um subgrafo é próprio quando

Subgrafos

- Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.



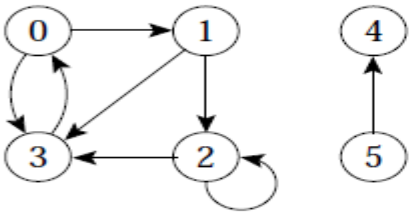


Um subgrafo é próprio quando V' ≠ V OU A' ≠ A

Subgrafos

- Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.
- Dado um conjunto $V' \subseteq V$, o subgrafo induzido por V' é o grafo $G' = (V', A')$, onde $A' = \{(u, v) \in A | u, v \in V'\}$.

Ex.: Subgrafo induzido pelo conjunto de vértices $\{1, 2, 4, 5\}$.

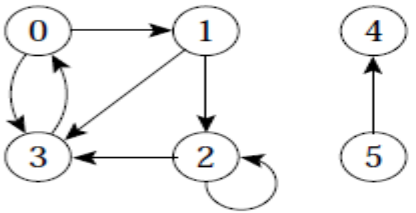


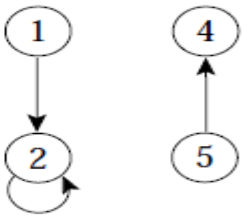
?

Subgrafos

- Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.
- Dado um conjunto $V' \subseteq V$, o subgrafo induzido por V' é o grafo $G' = (V', A')$, onde $A' = \{(u, v) \in A | u, v \in V'\}$.

Ex.: Subgrafo induzido pelo conjunto de vértices $\{1, 2, 4, 5\}$.

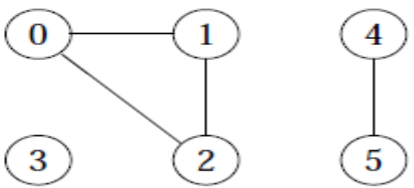




Grafo conectado (ou conexo)

- Um grafo não direcionado é conectado se cada par de vértices está conectado por um caminho.

Ex: esse grafo é conectado?

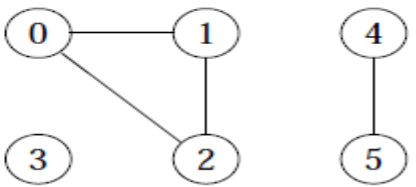


Grafo conectado (ou conexo)

- Um grafo não direcionado é conectado se cada par de vértices está conectado por um caminho.

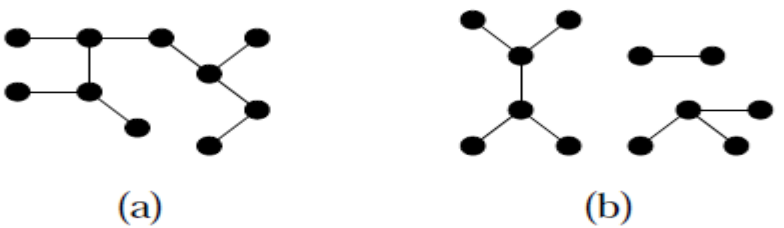
Ex: esse grafo é conectado?

Não!



Árvores

- **Árvore livre**: grafo não direcionado acíclico e conectado. É comum dizer apenas que o grafo é uma árvore omitindo o "livre".
- **Floresta**: grafo não direcionado acíclico, podendo ou não ser conectado.

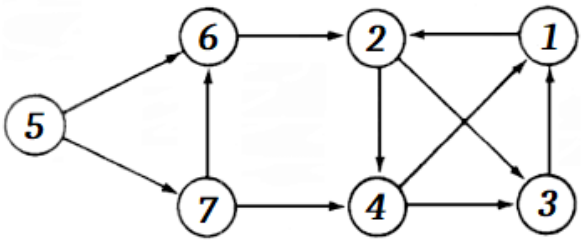


(a) árvore

(b) floresta

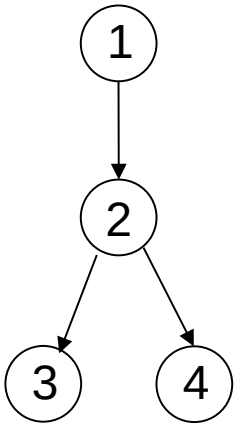
...fecha parêntesis para mais alguns conceitos básicos de grafos.

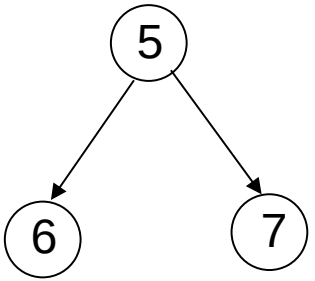
Qual a ordem em que os nós são descobertos durante a busca em profundidade?



Subgrafos de G que são árvores que descrevem a ordem em que os nós foram descobertos (tornados cinza)

Ordem de descoberta



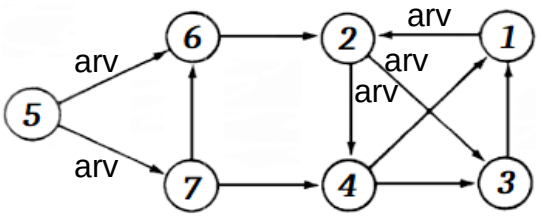


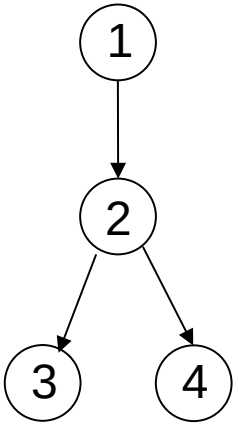
Árvores de busca em profundidade do grafo

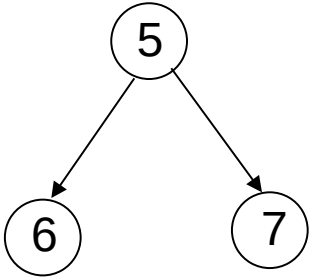
Investigações nessas árvores nos dão várias informações…
Para isso vamos ver alguns conceitos

Classificação de Arestas

1. **Arestas de árvore**: são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v).

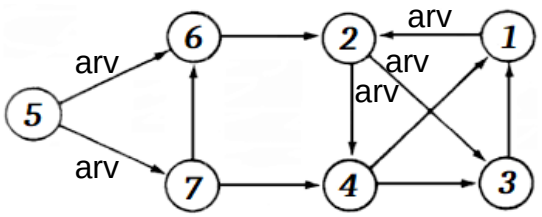


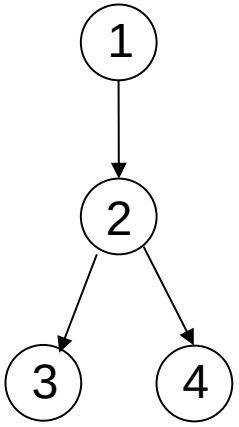


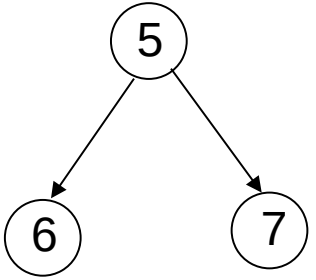


Classificação de Arestas

1. **Arestas de árvore**: são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v).
2. **Arestas de retorno**: conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*).

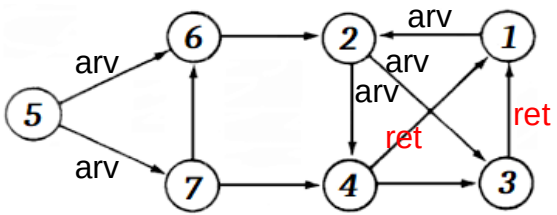


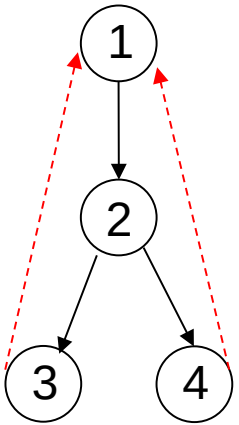


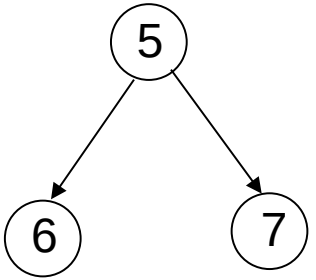


Classificação de Arestas

1. **Arestas de árvore**: são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v).
2. **Arestas de retorno**: conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*).

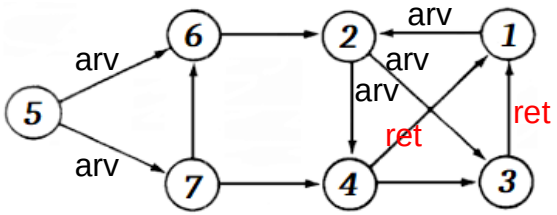


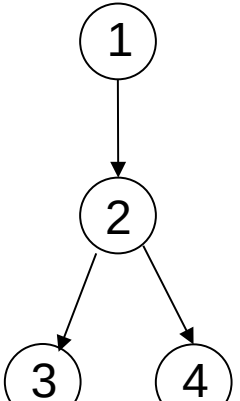


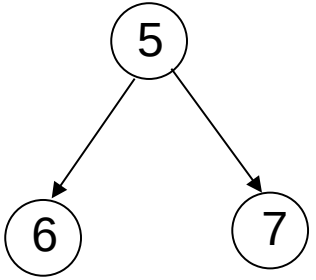


Classificação de Arestas

1. **Arestas de árvore**: são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v).
2. **Arestas de retorno**: conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*).
3. **Arestas de avanço**: não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade.

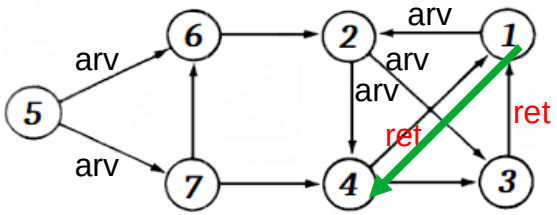


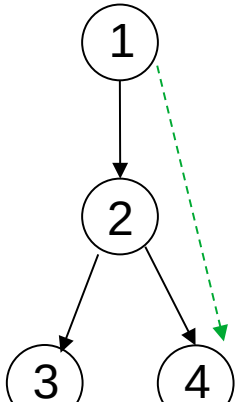


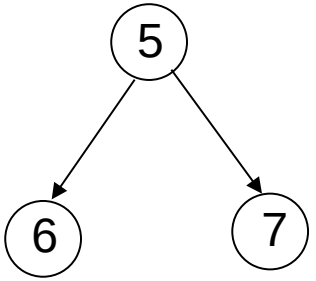


Classificação de Arestas

1. **Arestas de árvore**: são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v).
2. **Arestas de retorno**: conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*).
3. **Arestas de avanço**: não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade.

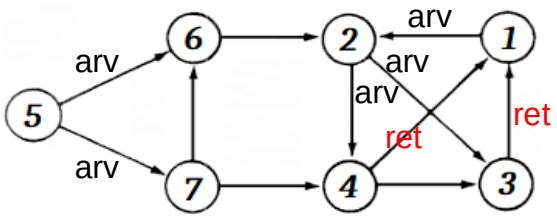


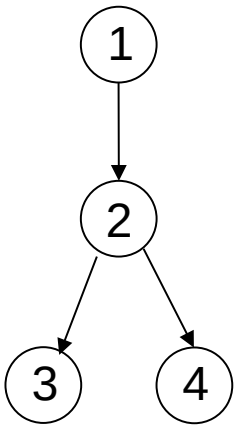


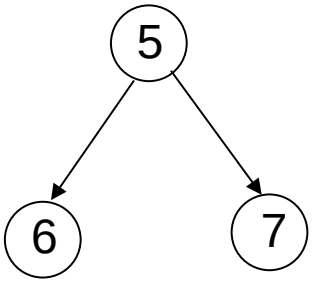


Neste grafo não há, mas se (1,4) existisse no grafo, ela seria de avanço

Classificação de Arestas



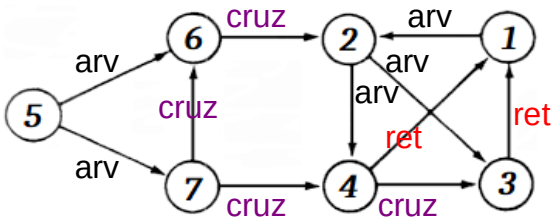


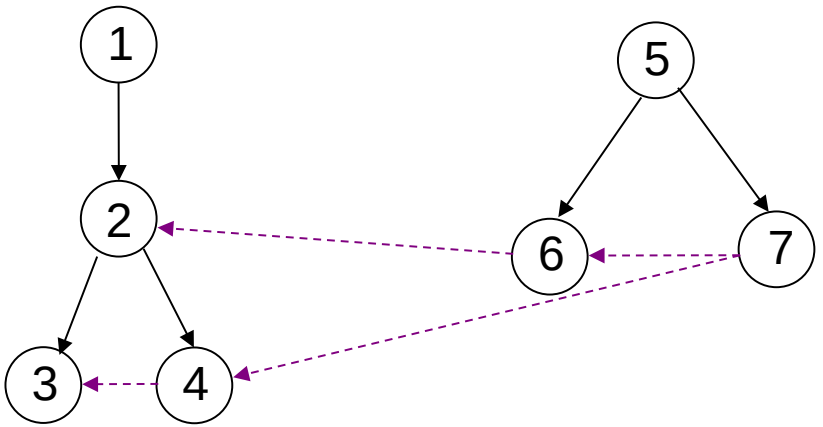


4. **Arestas de cruzamento**: podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes.

Neste caso (vértices da mesma árvore de busca), cruza ramos desta árvore, já que conecta um vértice a um outro que **não é seu antecessor nem descendente**

Classificação de Arestas





4. **Arestas de cruzamento**: podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes.

Neste caso (vértices da mesma árvore de busca), cruza ramos desta árvore, já que conecta um vértice a um outro que **não é seu antecessor nem descendente**

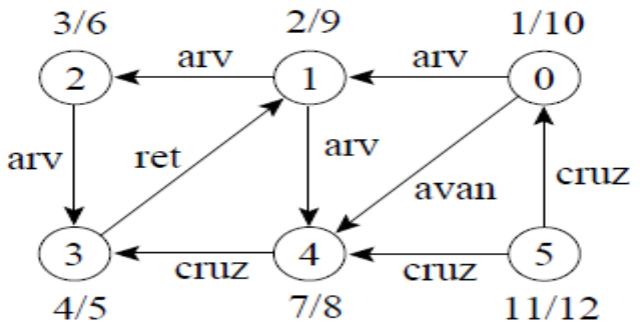
Classificação de Arestas

1. **Arestas de árvore**: são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v).
2. **Arestas de retorno**: conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*).
3. **Arestas de avanço**: não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade.
4. **Arestas de cruzamento**: podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes.

Neste caso (vértices da mesma árvore de busca), cruza ramos desta árvore, já que conecta um vértice a um outro que **não é seu antecessor nem descendente**

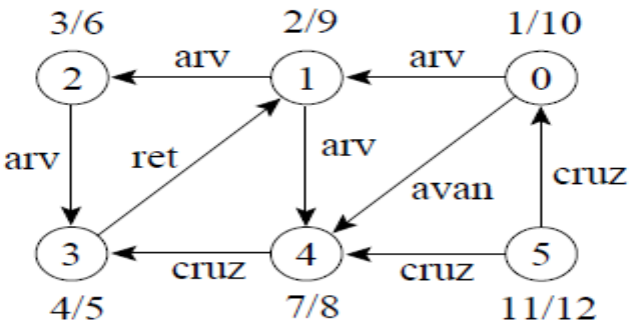
Classificação de Arestas

- Classificação de arestas pode ser útil para derivar outros algoritmos.
- Na busca em profundidade cada aresta (u,v) pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
  - Branco indica uma aresta de
  - Cinza indica uma aresta de
  - Preto indica



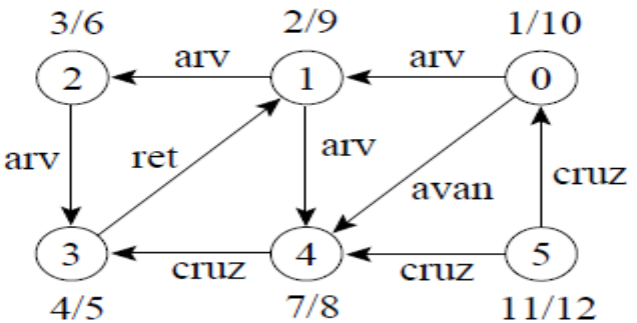
Classificação de Arestas

- Classificação de arestas pode ser útil para derivar outros algoritmos.
- Na busca em profundidade cada aresta (u,v) pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
  - Branco indica uma aresta de árvore.
  - Cinza indica uma aresta de
  - Preto indica



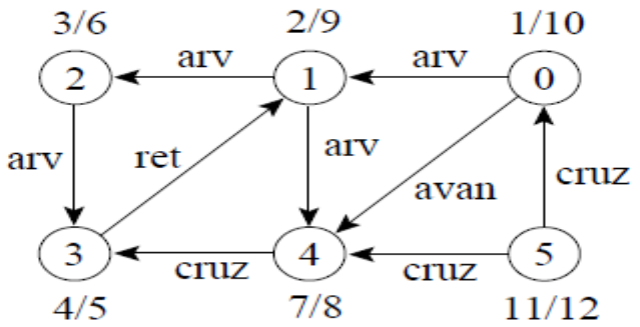
Classificação de Arestas

- Classificação de arestas pode ser útil para derivar outros algoritmos.
- Na busca em profundidade cada aresta (u,v) pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
  - Branco indica uma aresta de árvore.
  - Cinza indica uma aresta de retorno.
  - Preto indica



Classificação de Arestas

- Classificação de arestas pode ser útil para derivar outros algoritmos.
- Na busca em profundidade cada aresta (u,v) pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
  - Branco indica uma aresta de árvore.
  - Cinza indica uma aresta de retorno.
  - Preto indica uma aresta de avanço quando u é descoberto antes de v ou uma aresta de cruzamento caso contrário.

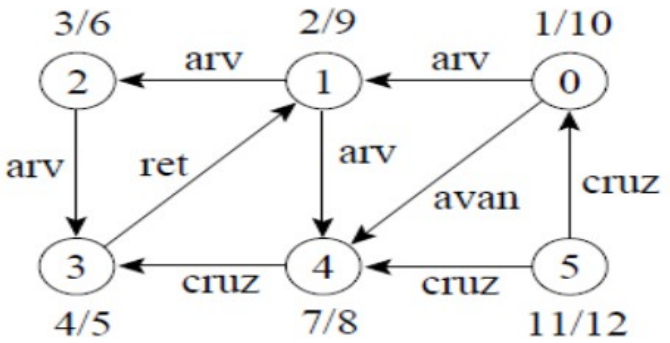


(u,v) é
avanço se tdesc[u] < tdesc[v]
cruzamento caso contrário

Observação

Se o grafo é não direcionado, a busca em profundidade produzirá apenas arestas de árvore e arestas de retorno

Por quê?



Exercício

Implementar Busca em profundidade utilizando as operações de interface dos grafos

Algumas aplicações de busca em profundidade

Identificar se um grafo é acíclico ou não

Com base nessa classificação de arestas, como identificar se um grafo é acíclico (não possui NENHUM ciclo) ou não?

Teste para Verificar se Grafo é Acíclico Usando Busca em Profundidade

- A busca em profundidade pode ser usada para verificar se um grafo é acíclico ou contém um ou mais ciclos.
- Se uma aresta de retorno é encontrada durante a busca em profundidade em G, então o grafo tem ciclo.
- Um grafo direcionado G é acíclico se e somente se a busca em profundidade em G não apresentar arestas de retorno.
- O algoritmo BuscaEmProfundidade pode ser alterado para descobrir arestas de retorno. Para isso, basta verificar se um vértice v adjacente a um vértice u apresenta a cor cinza na primeira vez que a aresta (u, v) é percorrida.
- O algoritmo tem custo linear no número de vértices e de arestas de um grafo $G = (V, A)$ que pode ser utilizado para verificar se G é **a**cíclico.

Exercício

Implemente tal algoritmo.

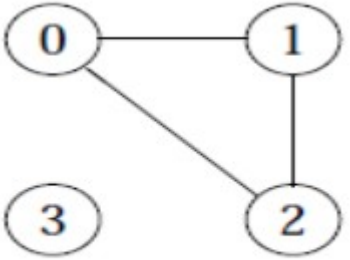
E se quiser que imprima tal ciclo?

Detecção de ciclos em grafos não-direcionados

- Segundo seu algoritmo, (u,v,u) seria detectado como um ciclo (ex: grafo G2)? Mas ciclos devem ter comprimento no mínimo 3 em um grafo não-direcionado… (grafo G3)
- O que isso muda a nossa ideia de busca em profundidade em grafos não-direcionados? Algo precisa ser adaptado nesses algoritmos?

G2:

G3:

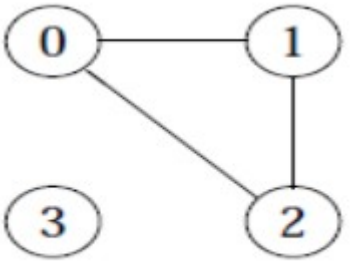


Detecção de ciclos em grafos não-direcionados

- Segundo seu algoritmo, (u,v,u) seria detectado como um ciclo (ex: grafo G2)? Mas ciclos devem ter comprimento no mínimo 3 em um grafo não-direcionado… (grafo G3)
- O que isso muda a nossa ideia de busca em profundidade em grafos não-direcionados? Algo precisa ser adaptado nesses algoritmos?

G2:

G3:



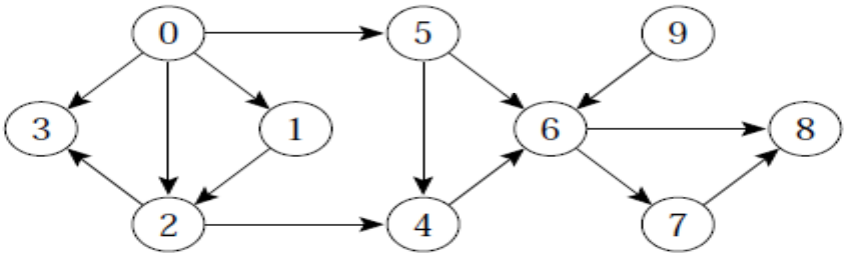
Cada aresta só pode ser percorrida 1 vez (u,v) = (v,u)

Ordenação Topológica

- Os grafos direcionados acíclicos são usados para indicar precedências entre eventos.

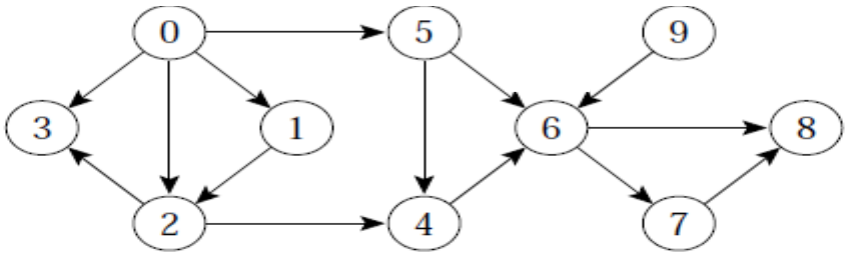
DAG

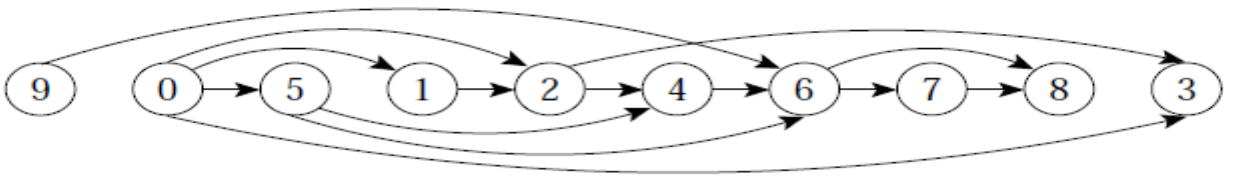
- Uma aresta direcionada (u, v) indica que a atividade u tem que ser realizada antes da atividade v.



Ordenação Topológica

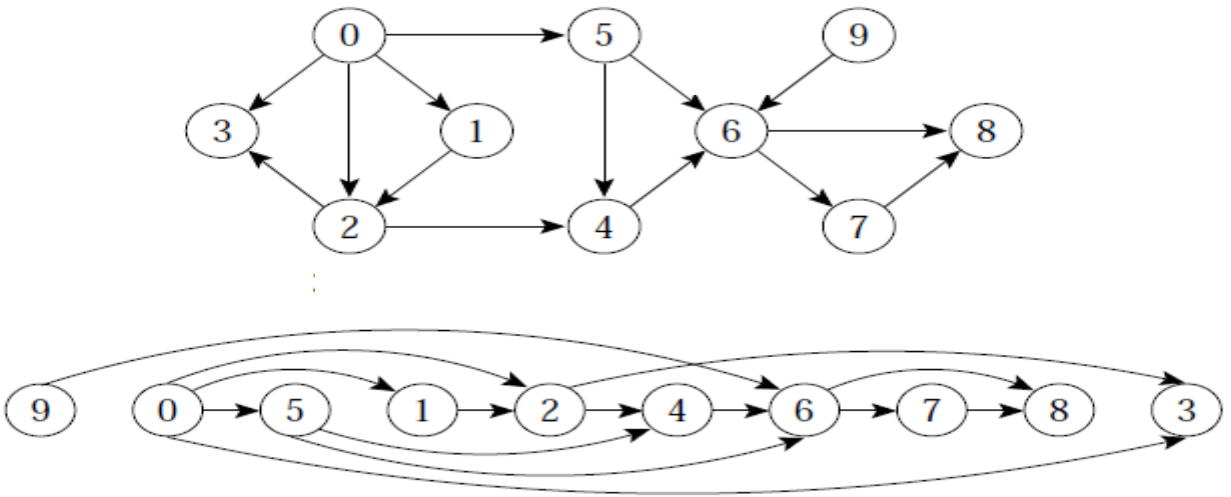
- Ordenação linear de todos os vértices, tal que se G contém uma aresta (u, v) então u aparece antes de v.
- Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita.





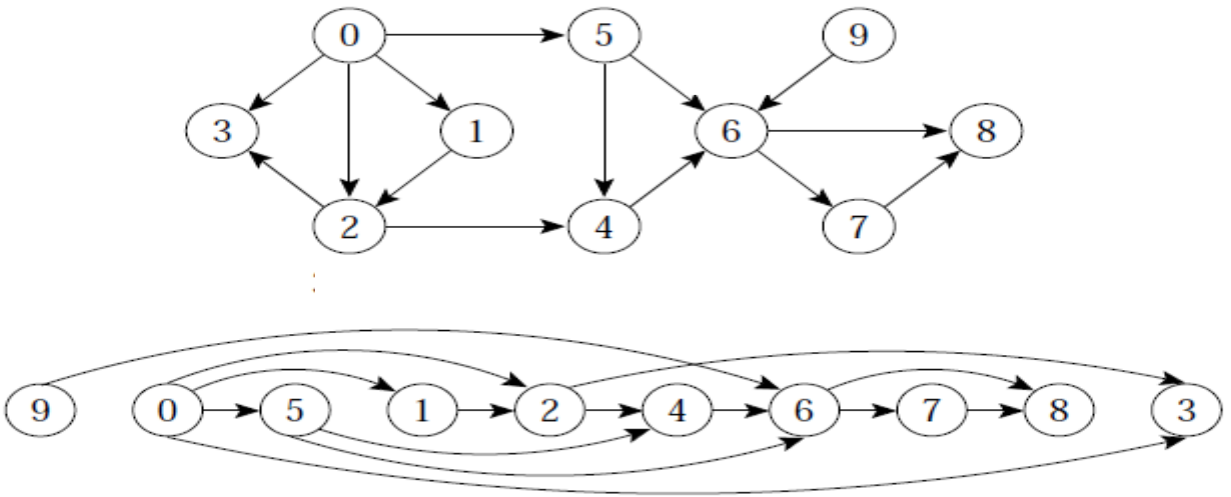
Ordenação Topológica

Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”?



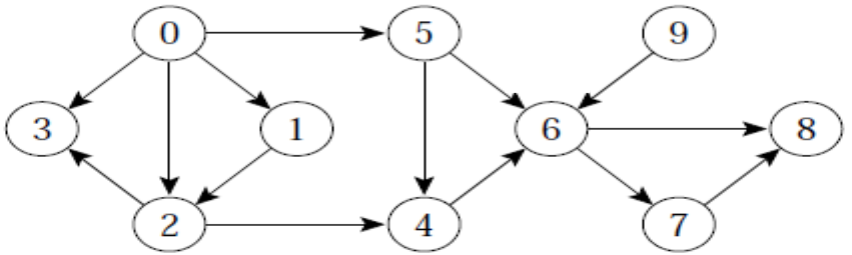
Ordenação Topológica

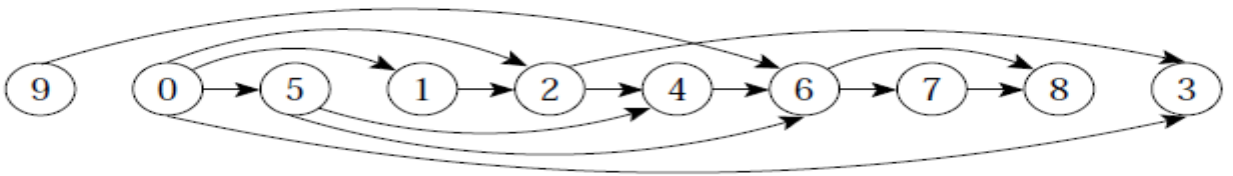
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.



Ordenação Topológica

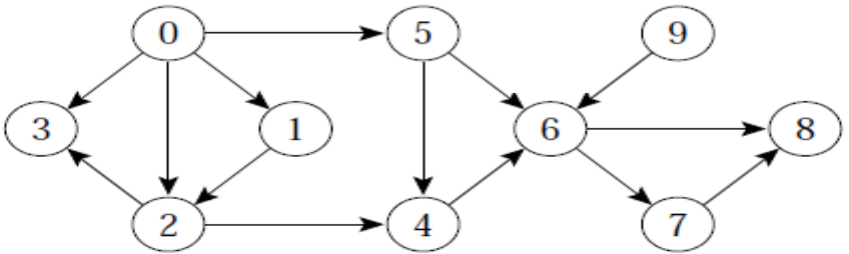
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)

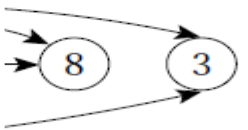




Ordenação Topológica

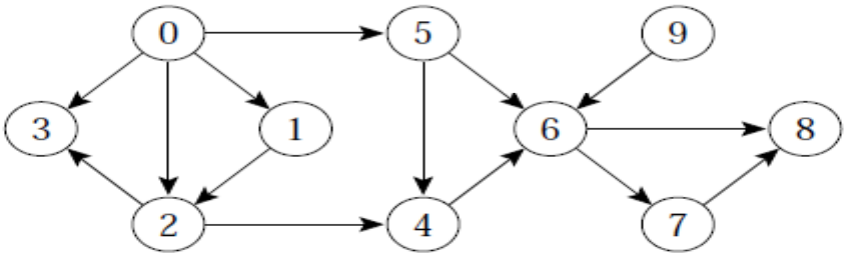
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)

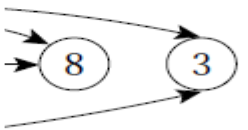




Ordenação Topológica

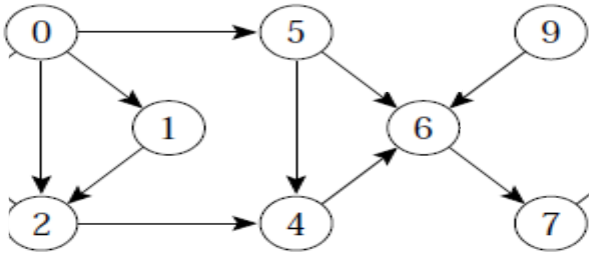
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?

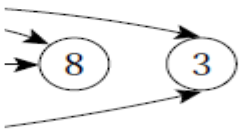




Ordenação Topológica

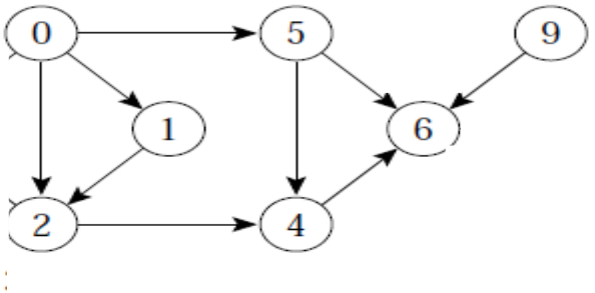
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça o processo.

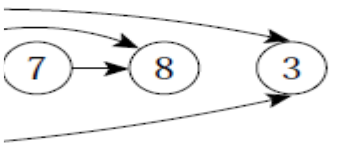




Ordenação Topológica

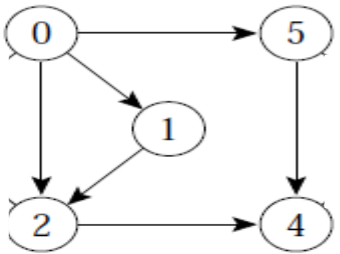
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça o processo.

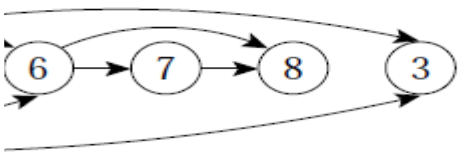




Ordenação Topológica

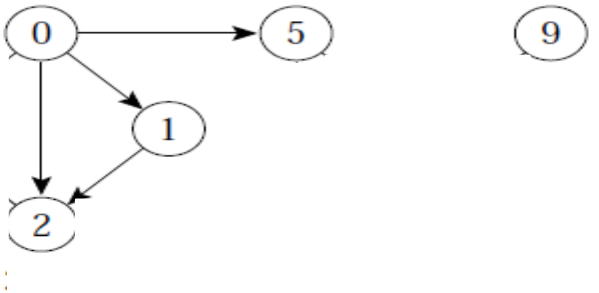
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça o processo.

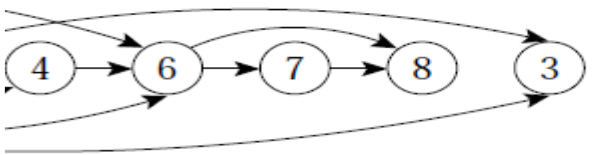




Ordenação Topológica

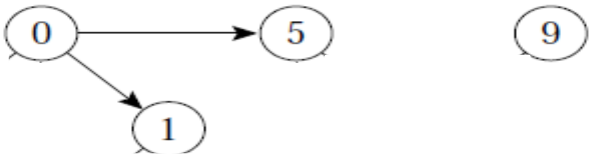
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça
o processo.

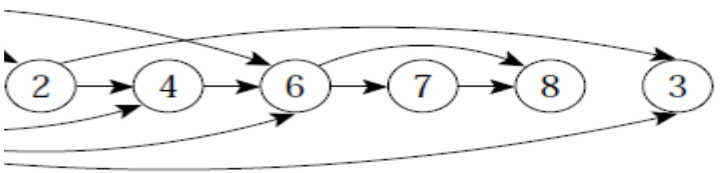




Ordenação Topológica

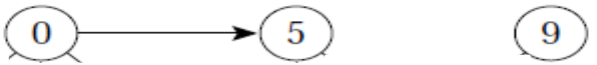
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça
o processo.

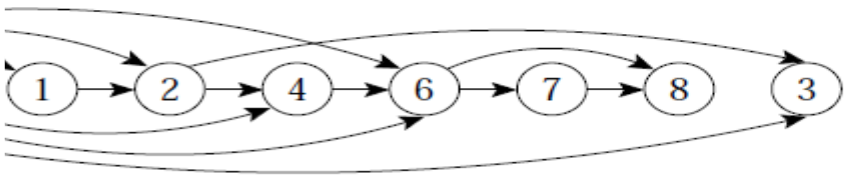




Ordenação Topológica

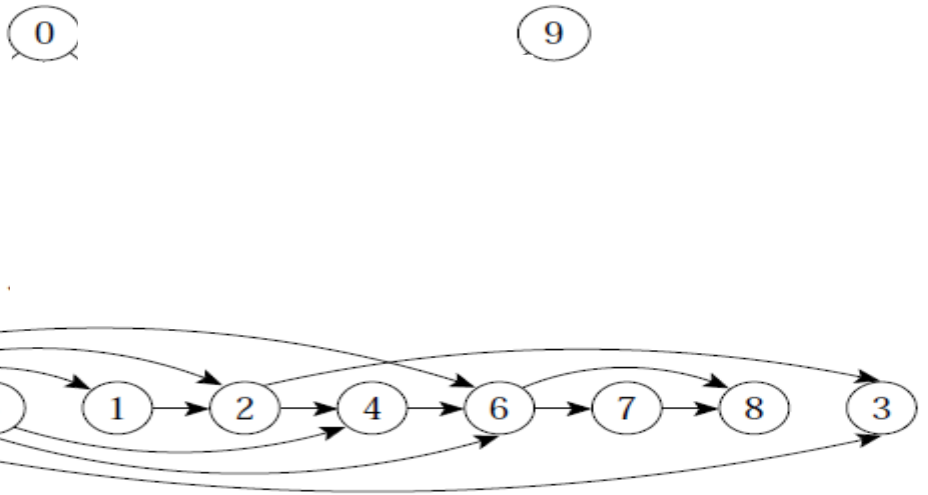
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça o processo.





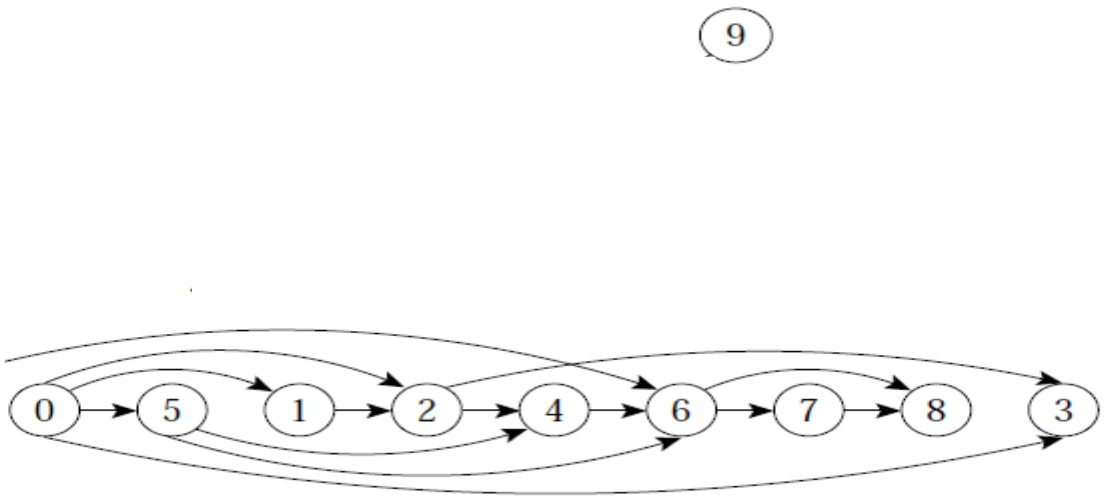
Ordenação Topológica

Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça
o processo.



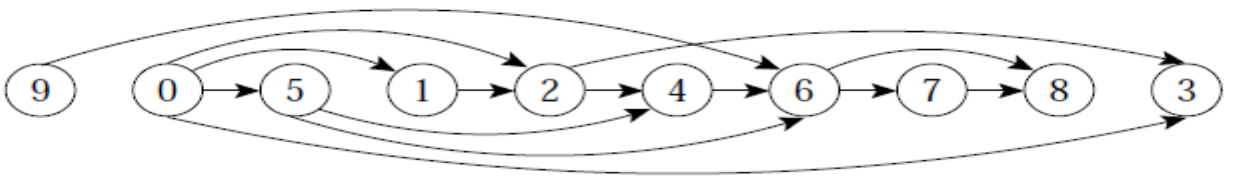
Ordenação Topológica

Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça
o processo.



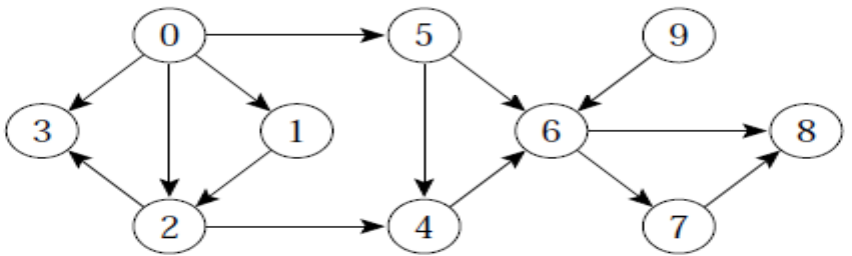
Ordenação Topológica

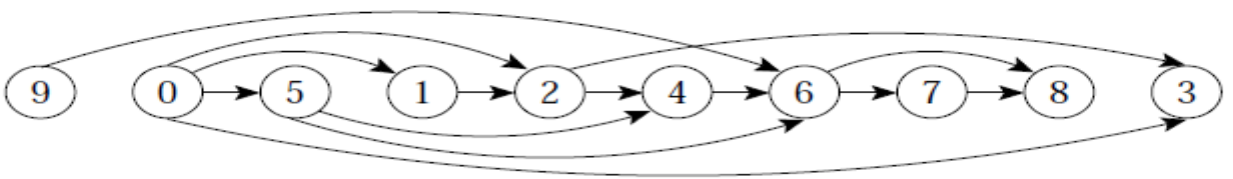
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça
o processo.



Ordenação Topológica

Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça o processo.

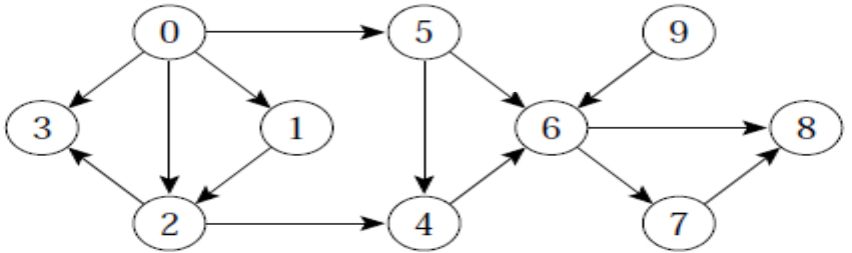


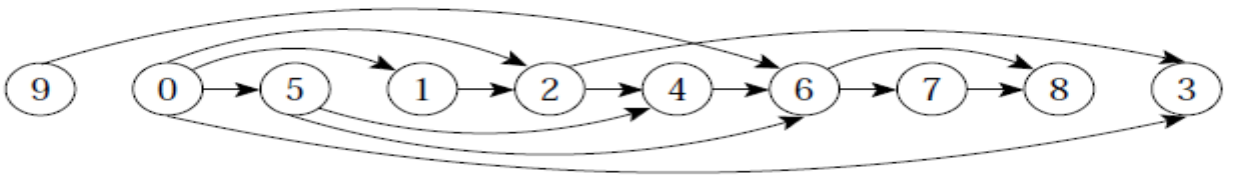


Qual seria a complexidade desse algoritmo?

Ordenação Topológica

Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista
(ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse vértice recém inserido (e as arestas que chegam nele) e recomeça o processo.

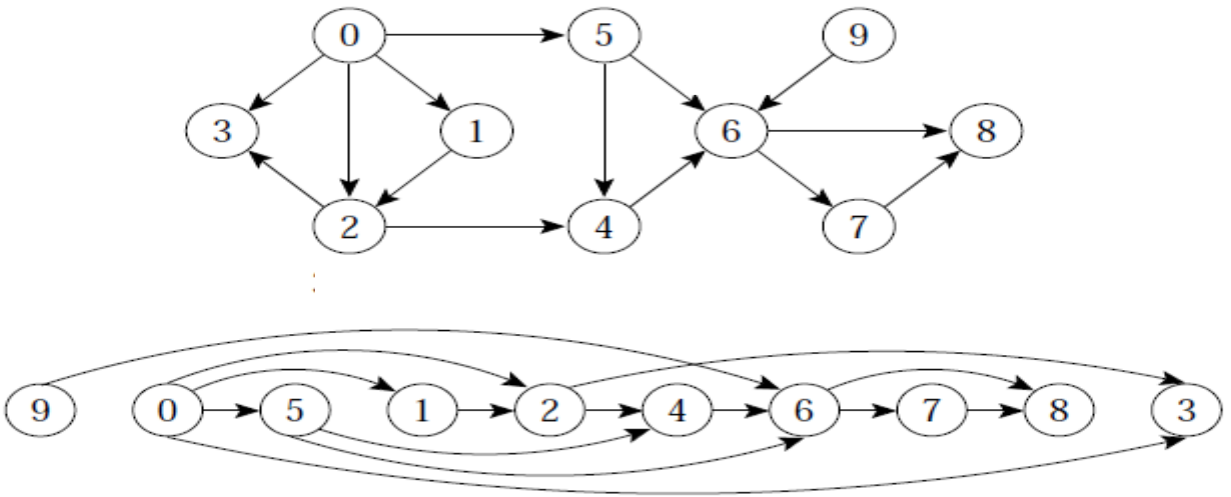




Qual seria a complexidade desse algoritmo? Pode chegar a $O(V^3)$:
Para cada vértice v, se listaAdjVazia(v) (O(V) para matriz de adjacência), coloca-o na lista (O(1))
e remove-o do grafo e as arestas que chegam nele (O(V) usando matriz)
A questão é como achar, de forma eficiente, os vértices sem adjacentes...

Ordenação Topológica

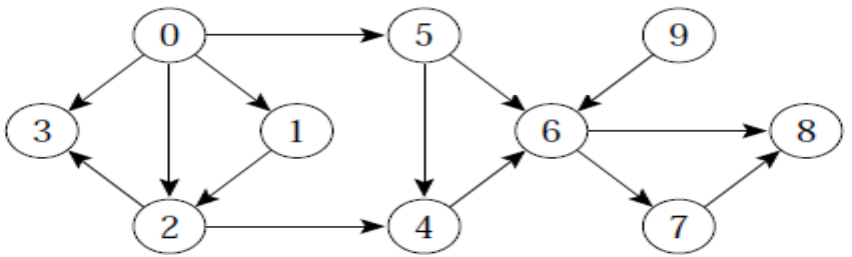
Como poderíamos obter essa informação dos vértices “sem adjacentes” (ou adjacentes já processados), a cada passo, de forma eficiente?

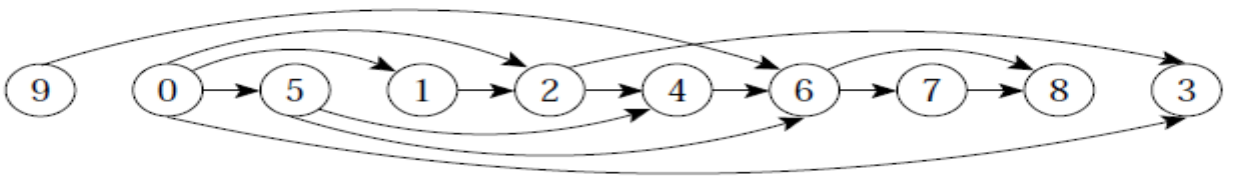


Ordenação Topológica

Como poderíamos obter essa informação dos vértices “sem adjacentes” (ou adjacentes já processados), a cada passo, de forma eficiente?

BUSCA EM PROFUNDIDADE!



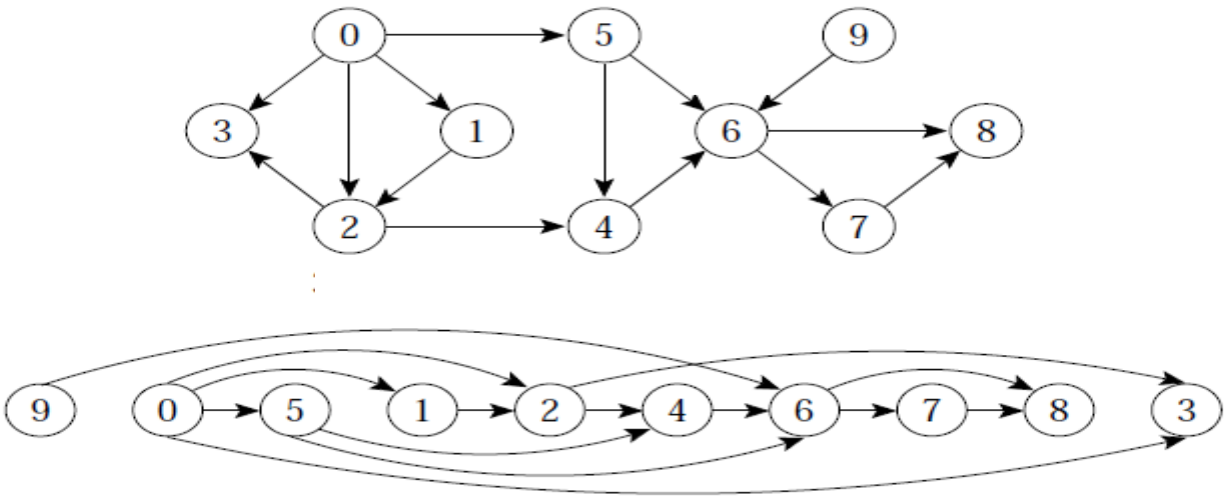


Ordenação Topológica

Como poderíamos obter essa informação dos vértices “sem adjacentes” (ou adjacentes já processados), a cada passo, de forma eficiente?

BUSCA EM PROFUNDIDADE!

Como?

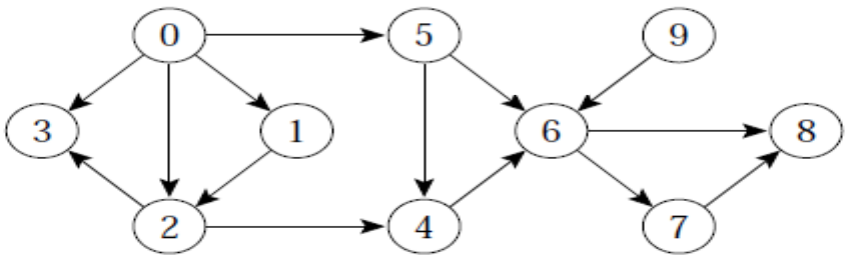


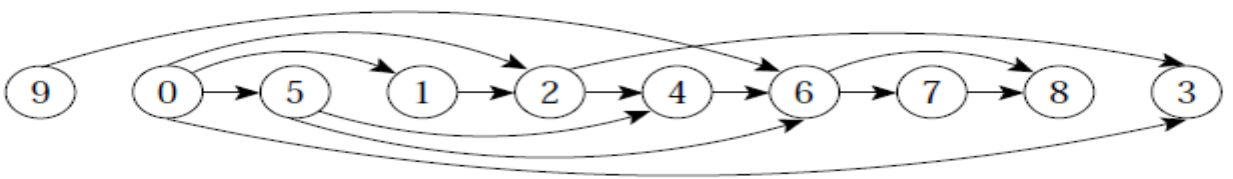
Ordenação Topológica

Como poderíamos obter essa informação dos vértices “sem adjacentes” (ou adjacentes já processados), a cada passo, de forma eficiente?

BUSCA EM PROFUNDIDADE!

Como? Insere o vértice no início da lista quando ele se tornar preto...



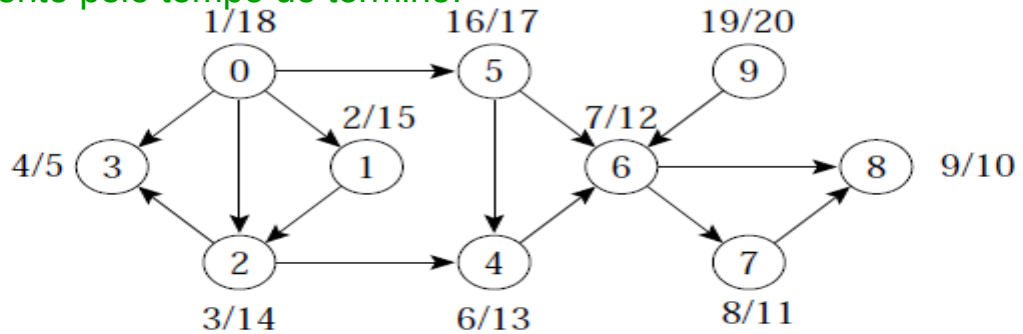


Ordenação Topológica

Como poderíamos obter essa informação dos vértices “sem adjacentes” (ou adjacentes já processados), a cada passo, de forma eficiente?

BUSCA EM PROFUNDIDADE!

Como? Insere o vértice no início da lista quando ele se tornar preto… Desta forma, os vértices ficaram ordenados decrescentemente pelo tempo de término.



Ordenação Topológica

- Algoritmo para ordenar topologicamente um grafo direcionado acíclico $G = (V, A)$:
  1. Chama *BuscaEmProfundidade(G)* para obter os tempos de término $t[u]$ para cada vértice u.
  2. Ao término de cada vértice insira-o na frente de uma lista linear encadeada.
  3. Retorna a lista encadeada de vértices.
- A Custo $O(|V| + |A|)$, uma vez que a busca em profundidade tem complexidade de tempo $O(|V| + |A|)$ e o custo para inserir cada um dos $|V|$ vértices na frente da lista linear encadeada custa $O(1)$.

Referências

Livro do Ziviani: cap 7

Livro do Cormen: seção 22.2