ACH2024

Aula 07 – Grafos: Busca em Profundidade - APLICAÇÕES

Prof. Helton Hideraldo Bíscaro

Na aula passada...

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 ← ++(*tempo);
    cor[v] ← preto;
}
```

COMPLEXIDADE:

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 ← ++(*tempo);
    cor[v] ← preto;
}
```

COMPLEXIDADE: O(V+A)

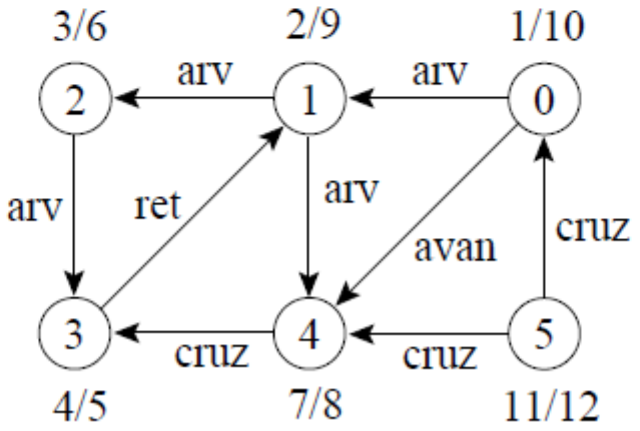
Classificação de Arestas

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 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?

Algumas aplicações de busca em profundidade

Identificar se um grafo é acíclico 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?

Pergunta-Exercício

Por que a busca precisa ser em profundidade (não espalhada) para detectar um ciclo? (dica: veja por exemplo o grafo do slide 6).

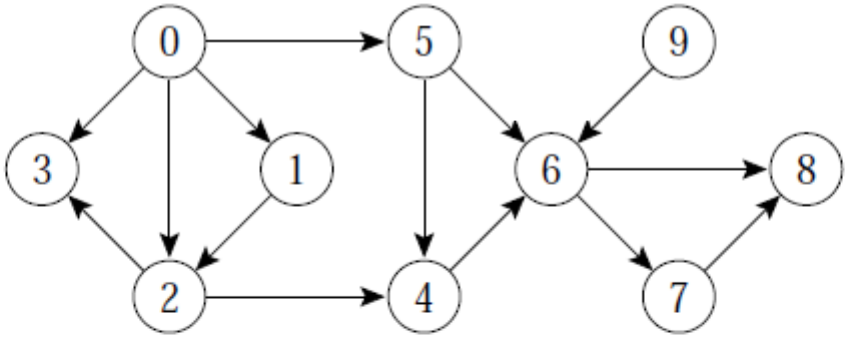
Pergunta

Por que a busca precisa ser em profundidade (não "espalhada") para detectar um ciclo? (dica: veja por exemplo o grafo do slide 6):

- Começando do vértice 0, supondo que a aresta (3,1) não existisse, a busca encontraria um vértice cinza
- Se a ideia é verificar se existe um caminho do vértice u até o vértice u, usar uma busca "espalhada" daria certo, mas daí teria que repetir o processo para cada vértice do grafo...

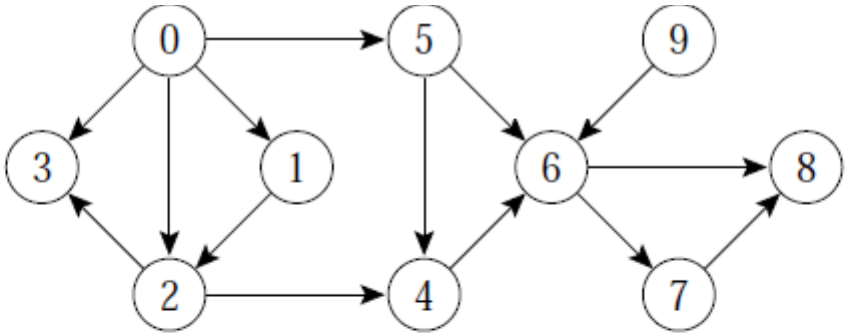
Ordenação Topológica

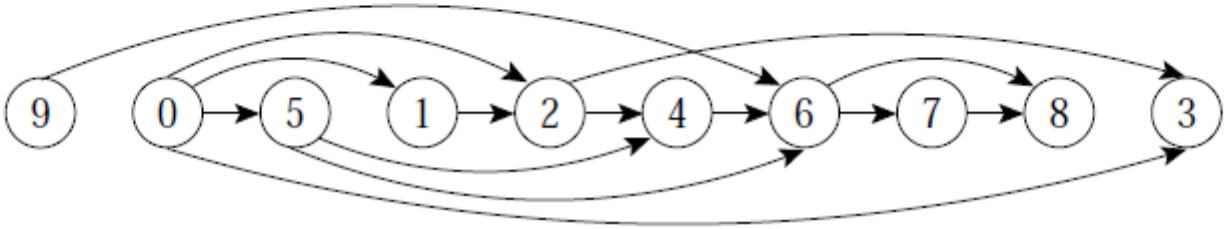
- Os grafos direcionados acíclicos são usados para indicar precedências entre eventos.
- Uma aresta direcionada (u, v) indica que a atividade u tem que ser realizada antes da atividade v.



Ordenação Topológica

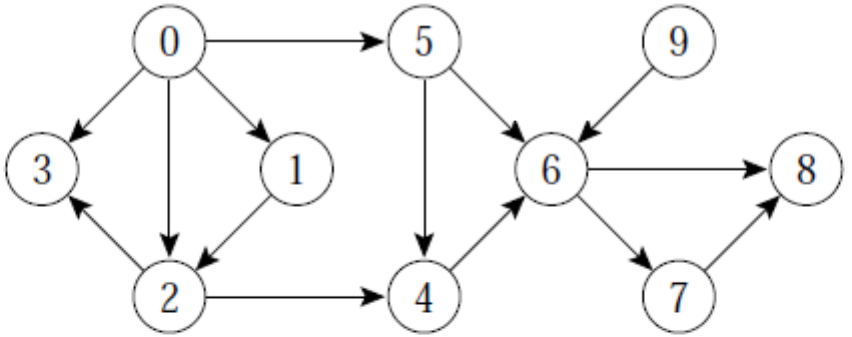
- Ordenação linear de todos os vértices, tal que se G contém uma aresta (u, v) então u aparece antes de v.
- Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita.

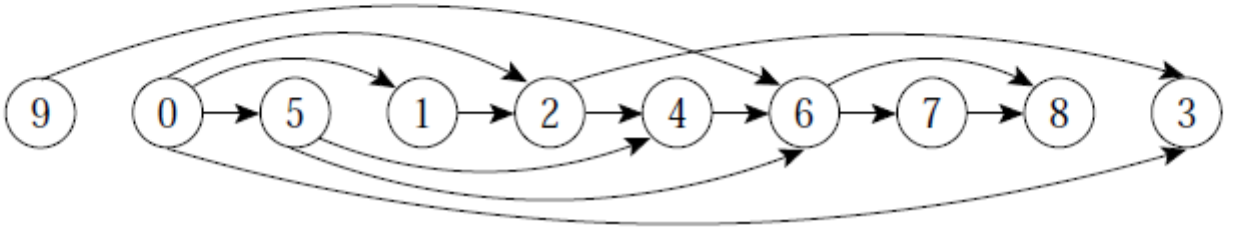




Ordenação Topológica

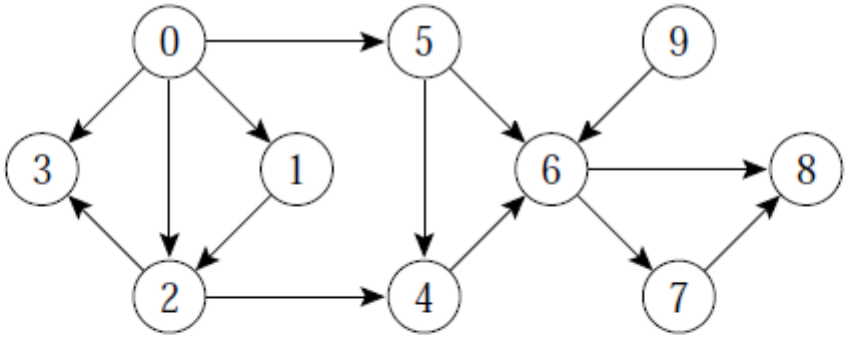
Como poderíamos obter tal ordenação de uma forma “ingênua”?
Quem são as últimas “tarefas”?

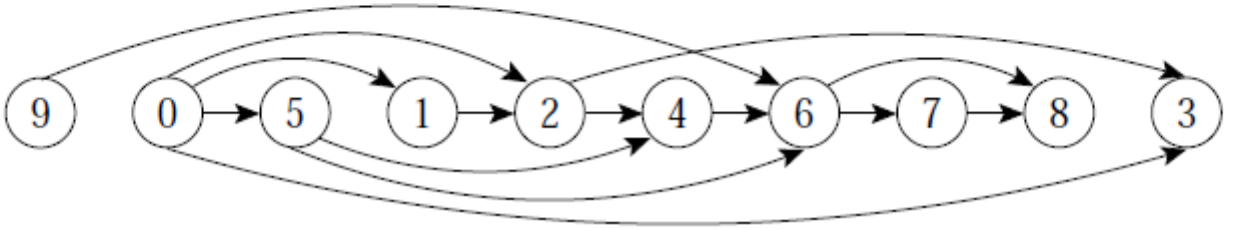




Ordenação Topológica

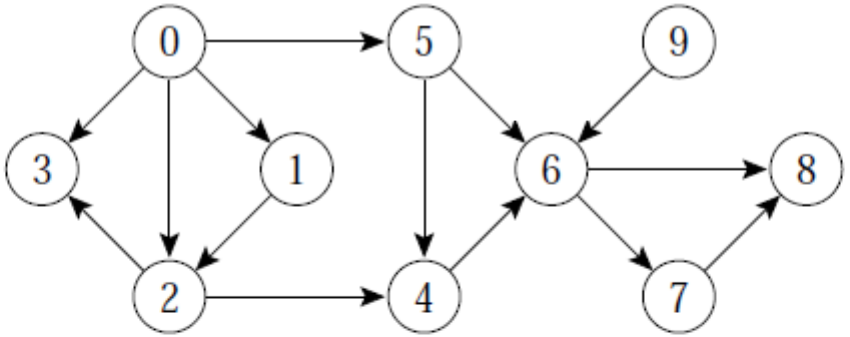
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.

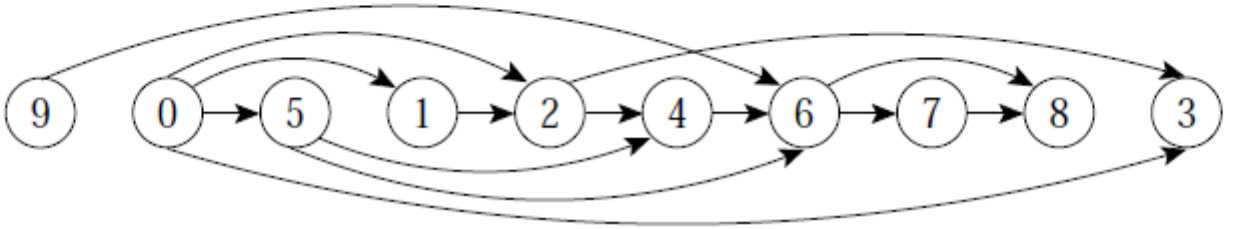




Ordenação Topológica

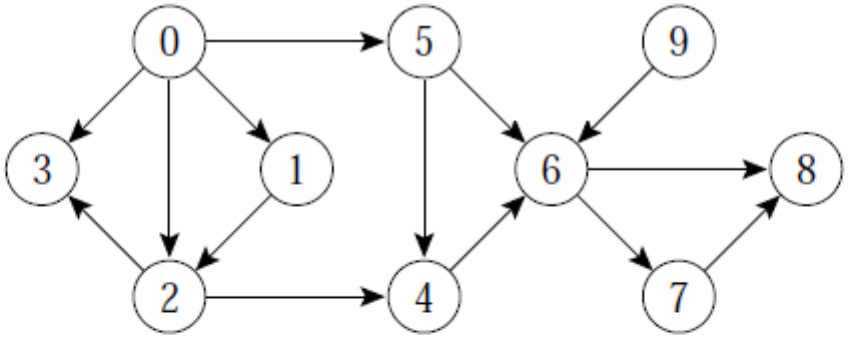
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista (ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)

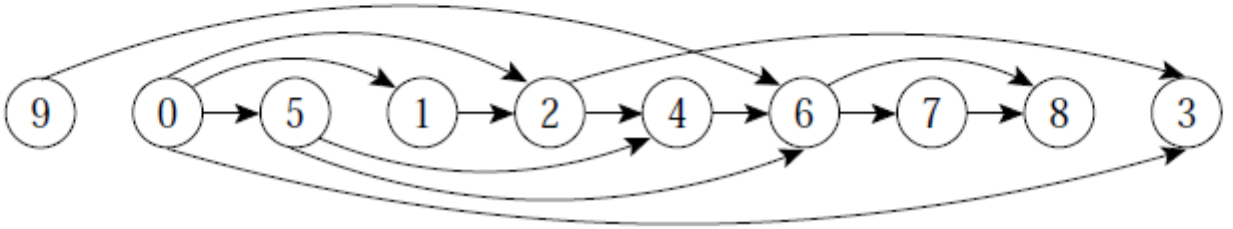




Ordenação Topológica

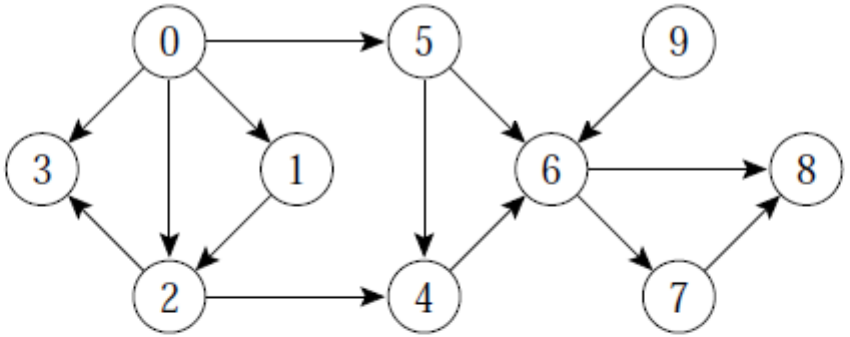
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista (ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?

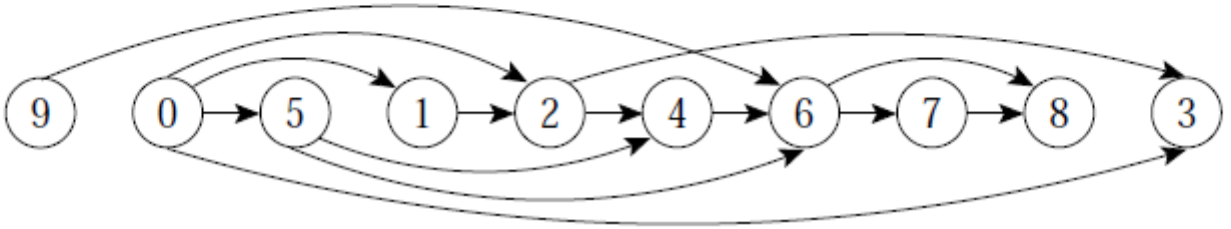




Ordenação Topológica

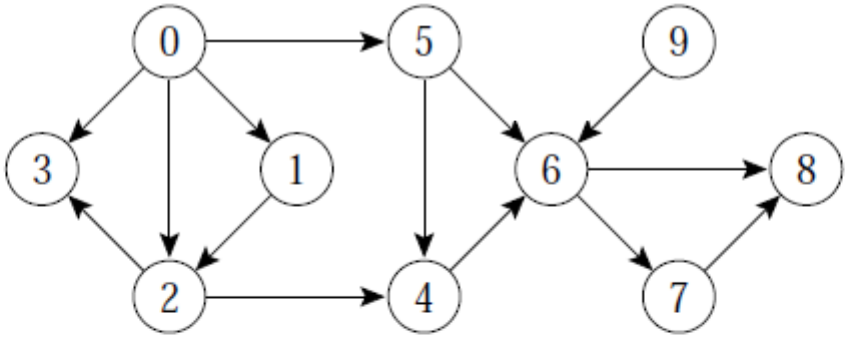
Como poderíamos obter tal ordenação?
Quem são as últimas “tarefas”? As que não têm tarefas adjacentes.
Então os vértices que não possuem arestas SAINDO deles devem ficar no fim da lista (ou seja, serem inseridos primeiro em uma lista ligada, com inserção sempre na frente)
E depois?
Remove do grafo esse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

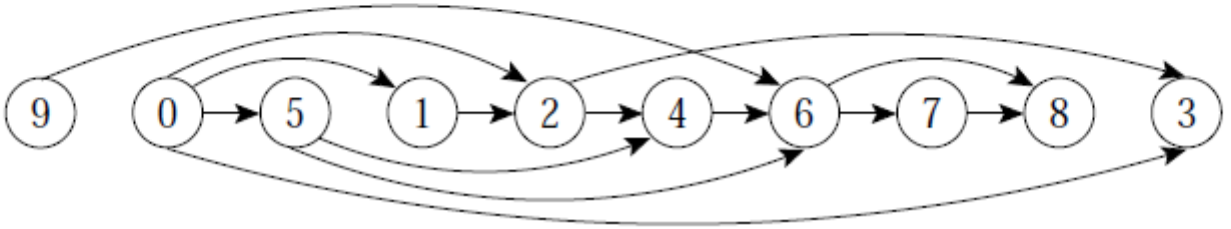




Ordenação Topológica

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.

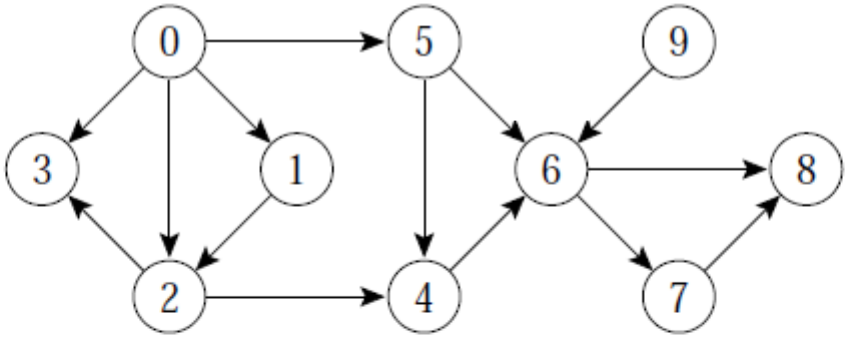


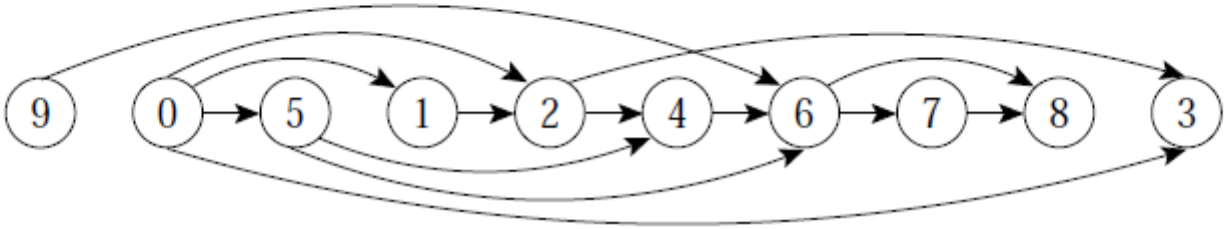


Qual seria a complexidade desse algoritmo?

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.

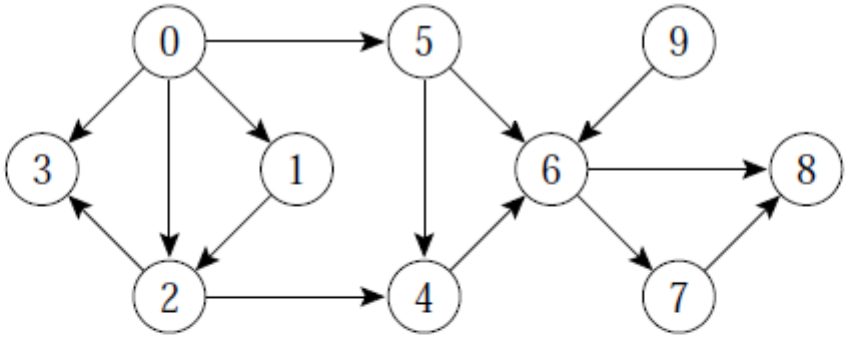


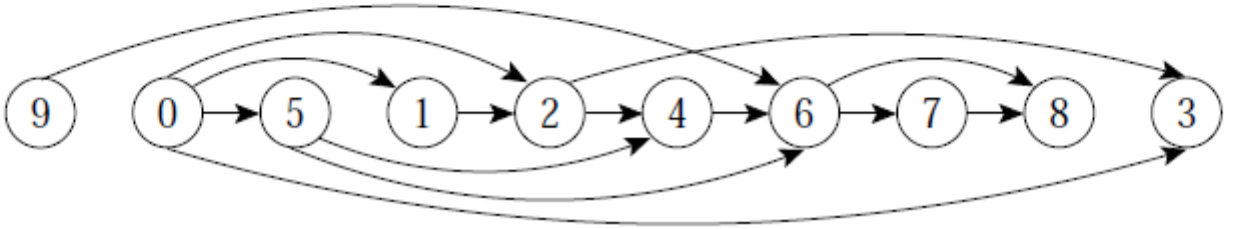


Qual seria a complexidade desse algoritmo? Pode chegar a $O(V^3)$:
Para cada vértice v, se listaAdjVazia(v) (O(V) para matriz de adjacencia), coloca-o na lista (O(1)) e remove-o do grafo e as arestas que chegam nele (O(V) usando matriz)
A questão é como achar, de forma eficiente, os vértices sem adjacentes...

Ordenação Topológica

Como poderíamos obter essa informação dos vértices sem adjacentes, a cada passo, de forma eficiente?

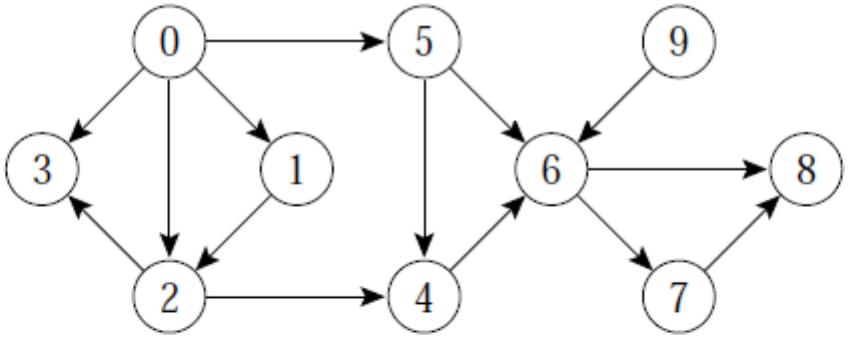


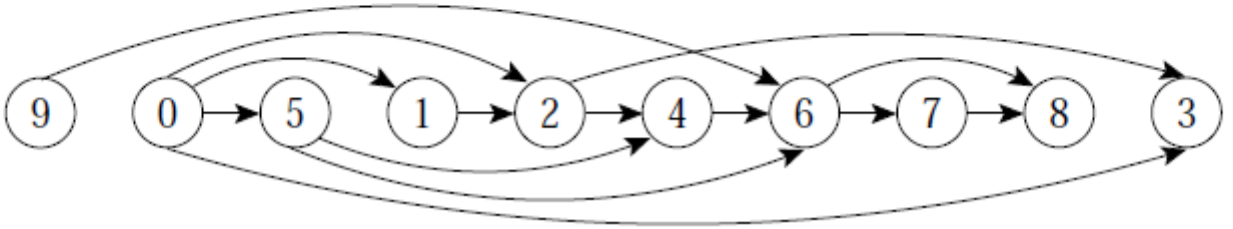


Ordenação Topológica

Como poderíamos obter essa informação dos vértices sem adjacentes, a cada passo, de forma eficiente?

BUSCA EM PROFUNDIDADE!



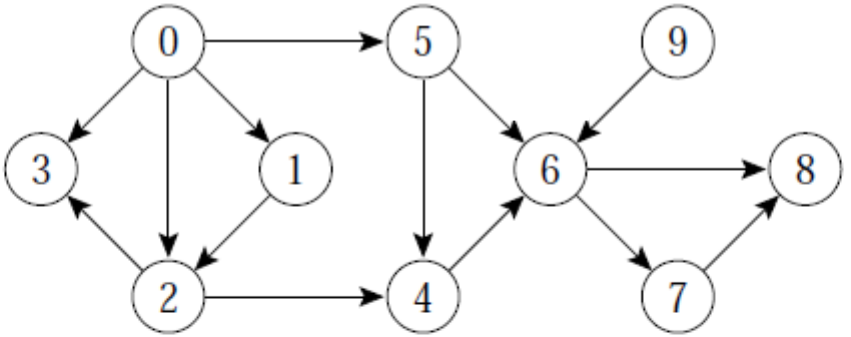


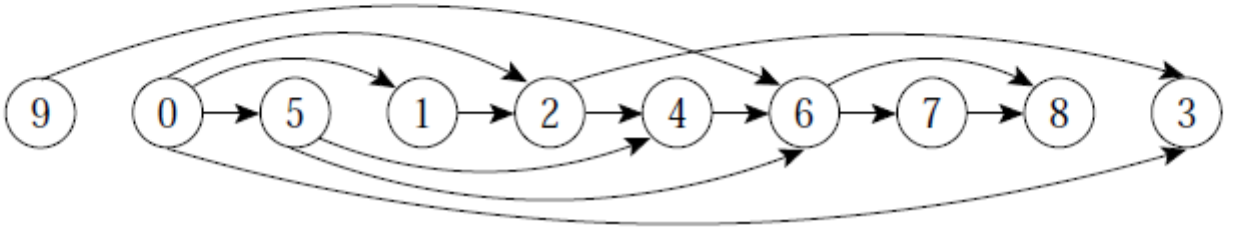
Ordenação Topológica

Como poderíamos obter essa informação dos vértices sem adjacentes, a cada passo, de forma eficiente?

BUSCA EM PROFUNDIDADE!

Como?



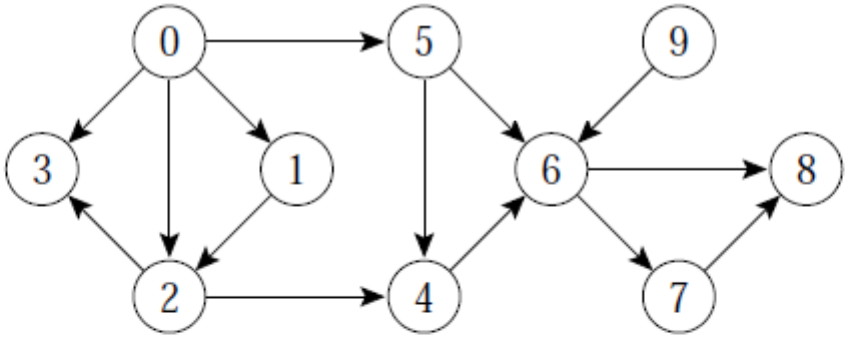


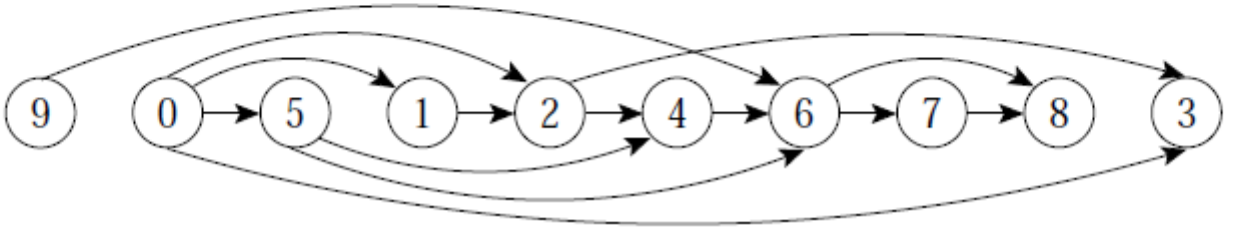
Ordenação Topológica

Como poderíamos obter essa informação dos vértices sem adjacentes, a cada passo, de forma eficiente?

BUSCA EM PROFUNDIDADE!

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

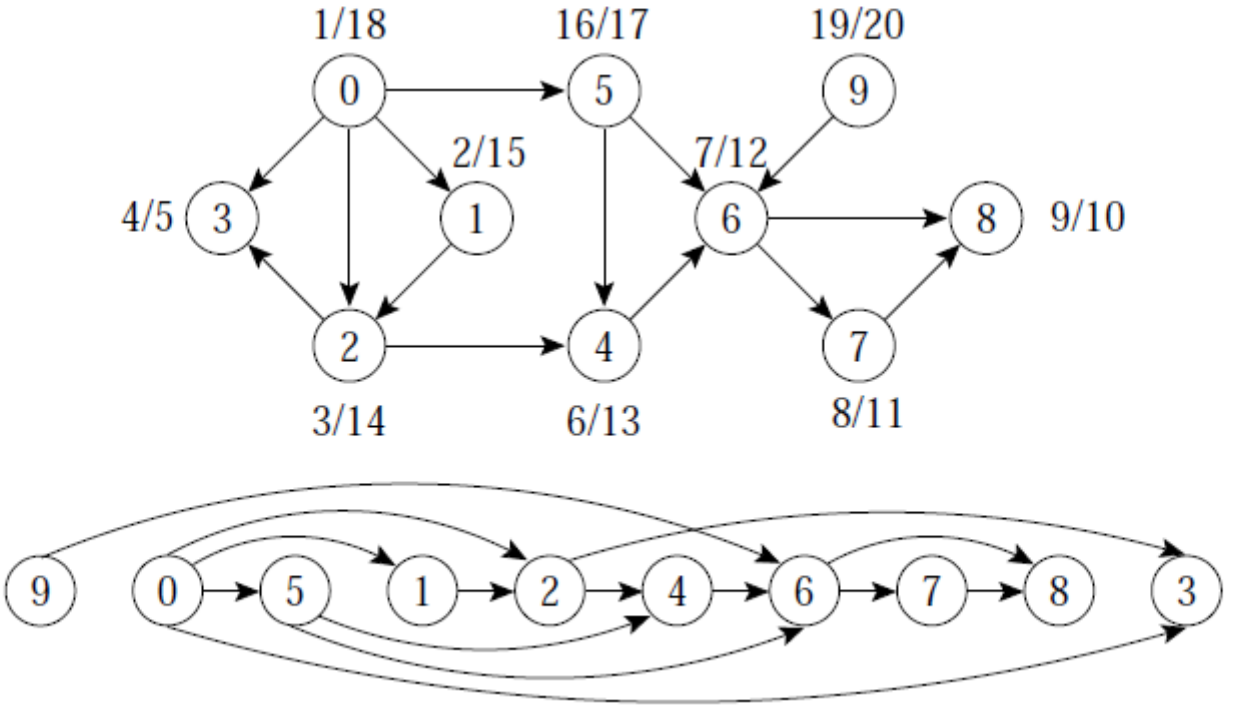




Ordenação Topológica

Desta forma, os vértices ficaram ordenados decrescentemente pelo tempo de término.

Por quê?



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 $O(1)$.

Encontrar um caminho entre dois vértices u e v

Encontrar **um** caminho entre dois vértices u e v

Busca em profundidade pode ser utilizado para encontrar (se existir) UM caminho entre dois vértices u e v

Partindo de u, o vértice v deve ser visitado (ou seja, v devendo pertencer à árvore de busca em profundidade com raiz em u)

Encontrar **um** caminho entre dois vértices u e v

Busca em profundidade pode ser utilizado para encontrar (se existir) UM caminho entre dois vértices u e v

Partindo de u, o vértice v deve ser visitado (ou seja, v devendo pertencer à árvore de busca em profundidade com raiz em u)

Esse caminho é o mais curto?

Encontrar **um** caminho entre dois vértices u e v

Busca em profundidade pode ser utilizado para encontrar (se existir) UM caminho entre dois vértices u e v

Partindo de u, o vértice v deve ser visitado (ou seja, v devendo pertencer à árvore de busca em profundidade com raiz em u)

Esse caminho é o mais curto? Não necessariamente...

Componentes conexos de um grafo não direcionado

Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo

- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v

Componentes conexos de um grafo não direcionado

Cada árvore de busca em profundidade corresponde a um componente conexo

Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo

  (isto também vale para grafos direcionados e componentes fortemente conexos?)

- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v

Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo

  (isto também vale para grafos direcionados e componentes fortemente conexos? SIM!)

- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v

Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo

  (isto também vale para grafos direcionados e componentes fortemente conexos? SIM!)

- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v

  (isto também vale para grafos direcionados e componentes fortemente conexos?)

Componentes conexos de um grafo não direcionado

Em um grafo **não** direcionado :

- cada vértice está contido em exatamente um componente conexo

  (isto também vale para grafos direcionados e componentes fortemente conexos? SIM!)

- para cada vértice v, o componente conexo que contém v é exatamente o conjunto de todos os vértices alcançáveis por v

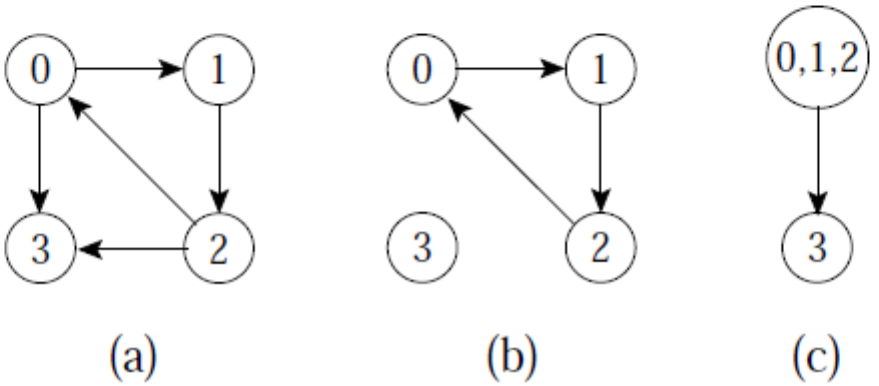
  (isto também vale para grafos direcionados e componentes fortemente conexos? NÃO!)

Exercício

Mostre um exemplo de grafo DIRECIONADO onde isso não acontece

Componentes Fortemente Conectados

- Um componente fortemente conectado de $G = (V, A)$ é um conjunto maximal de vértices $C \subseteq V$ tal que para todo par de vértices u e v em C, u e v são mutuamente alcançáveis
- Podemos particionar V em conjuntos V_i, $1 \leq i \leq r$, tal que vértices u e v são equivalentes se e somente se existe um caminho de u a v e um caminho de v a u.



Componentes Fortemente Conectados: Algoritmo

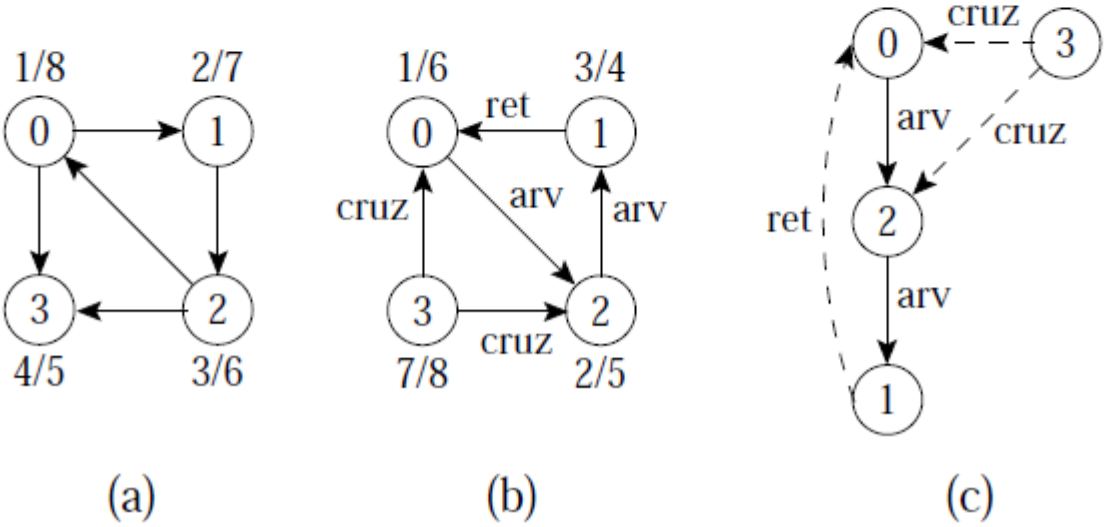
- Usa o **transposto** de G, definido $G^T = (V, A^T)$, onde $A^T = \{(u, v) : (v, u) \in A\}$, isto é, A^T consiste das arestas de G com suas direções invertidas.
- G e G^T possuem os mesmos componentes fortemente conectados, isto é, u e v são mutuamente alcançáveis a partir de cada um em G se e somente se u e v são mutuamente alcançáveis a partir de cada um em G^T.

Componentes Fortemente Conectados: Algoritmo

1. Chama *BuscaEmProfundidade*(G) para obter os tempos de término $t[u]$ para cada vértice u.
2. Obtem G^T.
3. Chama *BuscaEmProfundidade*(G^T), realizando a busca a partir do vértice de maior $t[u]$ obtido na linha 1. Inicie uma nova busca em profundidade a partir do vértice de maior $t[u]$ dentre os vértices restantes se houver.
4. Retorne os vértices de cada árvore da floresta obtida como um componente fortemente conectado separado.

Componentes Fortemente Conectados: Exemplo

- A parte (b) apresenta o resultado da busca em profundidade sobre o grafo transposto obtido, mostrando os tempos de término e a classificação das arestas.
- A busca em profundidade em G^T resulta na floresta de árvores mostrada na parte (c).



Componentes Fortemente Conectados: Análise

- Utiliza o algoritmo para busca em profundidade duas vezes, uma em G e outra em G^T.
- Logo, a complexidade total é $O(|V| + |A|)$.