maratona de programação

UNIFEI

6 – Grafos I

Introdução a Grafos

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação
sites.google.com/site/unifeimaratona/

Veremos o básico:

1) Conceitos básicos sobre grafos.

2) Explorando Grafos:

- DFS (Buscam em profundidade);
- Encontrando componentes conexas;
- Flood Fill;
- Ordenação Topológica;
- Pontos de Articulação e Pontes;
- Componentes fortemente conexas, em grafos direcionados (algoritmo de Tarjan).

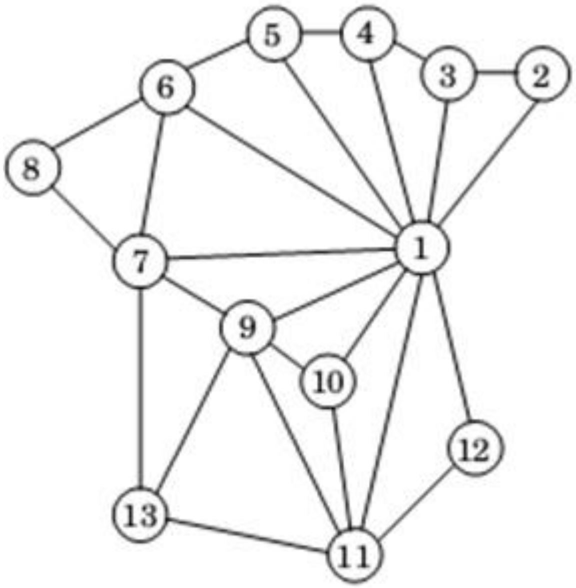
* Seções para auto-estudo, com exercícios relacionados na tarefa do *URI Academic*. Fique atento para o prazo de entrega.

Sobre grafos

Uma grande variedade de problemas pode ser expressa com clareza e precisão na linguagem gráfica e concisa dos grafos.

- Coloração de um mapa político, sem que nenhuma unidade possua fronteira com outro de mesma cor;
- Auxílio na busca de informações por uma máquina na web;
- Definição do roteiro mais curto para visitar as principais cidades de uma região turística;
- Fluxo máximo de algum fluido em sistemas de tubos;
- Controle de tráfego em uma cidade, etc.

Sobre grafos



Grafo: Vértices + Arestas

Notação: G = (V, E)

No exemplo do mapa, temos vários vértices {1, 2, 3 ... 13} e muitas arestas, como {1,2}, {9,11}, {7,13}.

O exemplo é um grafo não-direcionado, pois nele, as arestas possuem uma relação de simetria: “x faz fronteira com y” e “y faz fronteira com x”, sempre.

Sobre grafos

Muitas vezes, grafos representam cenários onde a relação entre os vértices não é, necessariamente, simétrica. Neste caso, o caminho de um vértice a outro não implica em um caminho no sentido contrário.

Para esta finalidade, existem os grafos direcionados.

Escrevemos:

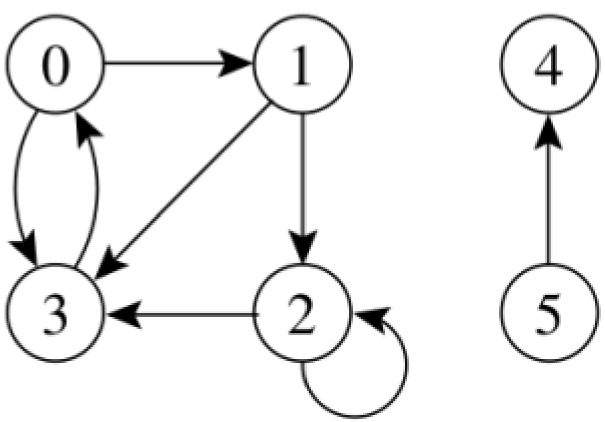
- e = (x, y), quando for direcionada de x para y
- e = (y, x), quando for direcionada de y para x

Observação importante: Nada impede que um grafo possua as duas arestas, (x,y) e (y,x), indicando que existe um caminho tanto de x para y quanto de y para x. A informação apenas precisa ser explícita.

Sobre grafos

Grafos Direcionados

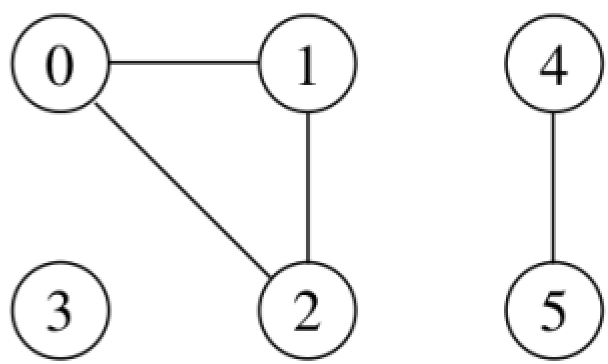
- Uma aresta (u,v) sai do vértice u e vai até v, indicando que o vértice v é adjacente ao vértice u (mas não o contrário)
- Podem existir arestas de um vértice para ele mesmo, que chamamos de *self-loops*.



Sobre grafos

Grafos Não-Direcionados

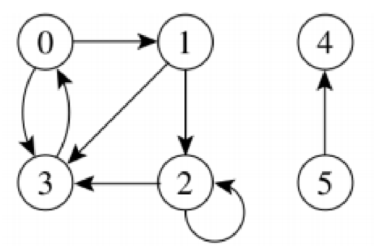
- As arestas (u, v) e (v, u) são consideradas a mesma aresta e podem ser representadas por {u,v}.
- A relação de adjacência é simétrica, ou seja, u é adjacente a v, que também é adjacente a u.
- Não podem existir arestas de um vértice para ele mesmo, ou *self-loops*.



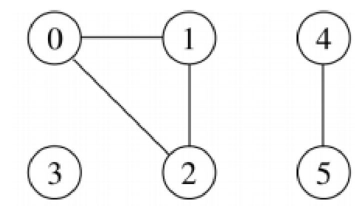
Como representar um grafo?

Matriz de Adjacência:

Se há n vértices no grafo, a matriz de adjacência é uma matriz de *n x n* cujo elemento na posição (i,j) é igual a 1, caso exista uma aresta de v_i para v_j ou igual 0 caso contrário.



	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						
5						



	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						
5						

Como representar um grafo?

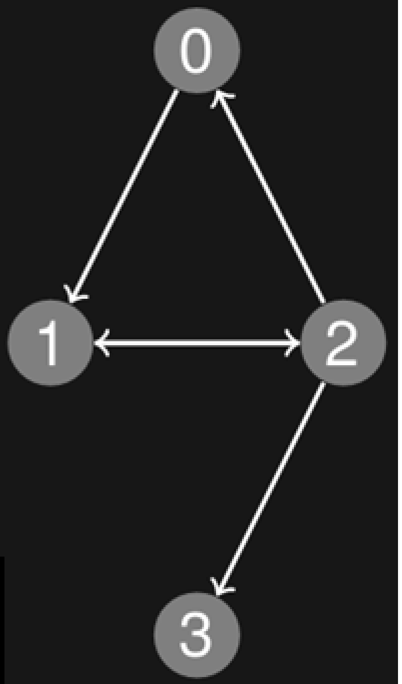
Matriz de Adjacência:

- Para grafos não-direcionados, a matriz é simétrica, pois uma aresta pode ser tomada em ambas as direções.
- Presença de uma aresta pode ser checada rapidamente, em tempo constante O(1), e independe do número de vértices (V) ou arestas (E).
- Espaço utilizado para armazenamento na memória é grande, proporcional ao quadrado do número de vértices ou $O(n^2)$, e, portanto, ler ou examinar a matriz tem complexidade de tempo $O(n^2)$.

Matriz de Adjacência:

```c
#define VERT 4
int mat_adj[VERT][VERT];

// initialize matrix
memset(mat_adj, 0, sizeof mat_adj);

// set edges
mat_adj[0][1] = 1;
mat_adj[1][2] = 1;
mat_adj[2][1] = 1;
mat_adj[2][0] = 1;
mat_adj[2][3] = 1;
```

```
0 1 0 0
0 0 1 0
1 1 0 1
0 0 0 0
```



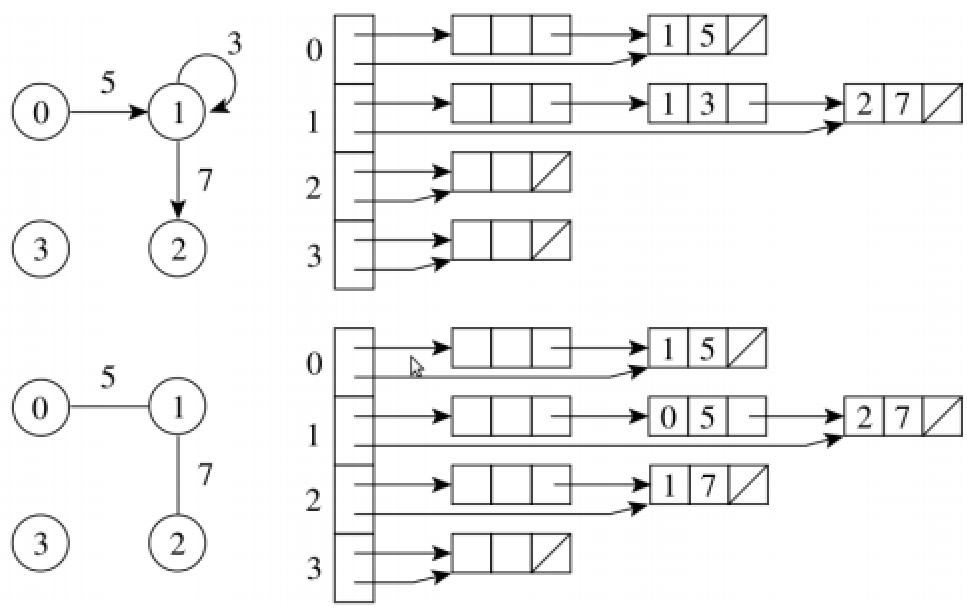
Como representar um grafo?

Listas de Adjacência:

– Um arranjo de n listas ligadas, uma para cada vértice de V.

– Tamanho proporcional ao número de arestas

– Para cada u pertencente a V, lista[u] contém todos os vértices adjacentes a u em G.



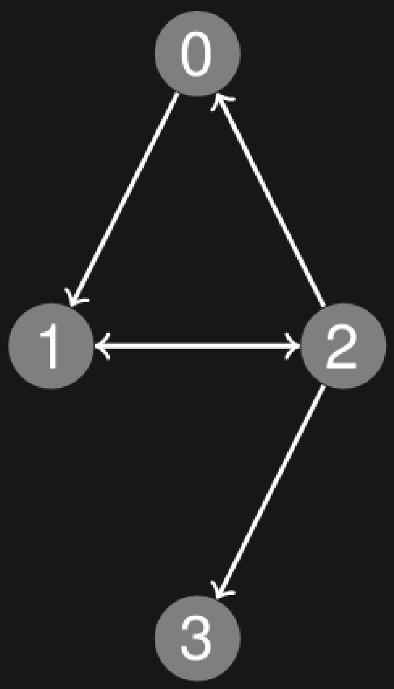
Como representar um grafo?

Listas de Adjacência:

- Cada aresta aparece em exatamente uma das listas ligadas quando o grafo é direcionado e em duas listas quando o grafo é não-direcionado.
- Tamanho total da estrutura de dados é O(V+E).
- Checar pela existência de uma determinada aresta (u,v) não é mais em tempo constante, ainda que seja uma tarefa simples: É preciso percorrer a lista de adjacência de u. Pode ter tempo O(V), pois podem haver |V| arestas saindo de u.

Listas de Adjacência:

```
0: 1
1: 2
2: 0, 1, 3
3:

vector<int> adj[4];
adj[0].push_back(1);
adj[1].push_back(2);
adj[2].push_back(0);
adj[2].push_back(1);
adj[2].push_back(3);
```



Qual a melhor representação?

Depende da relação entre $|V|$, o número de vértices do grafo, e $|E|$, o número de arestas.

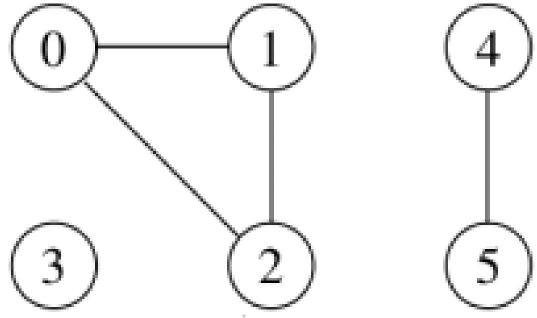
O número de arestas pode ser tão pequeno quanto o número de vértices (ou menor), ou tão grande quanto $|V|^2$.

- Quando $|E|$ estiver perto do limite superior desse intervalo, dizemos que o grafo é denso (*dense*), e a melhor opção para a sua representação é a matriz de adjacência.
- Quando $|E|$ estiver perto de $|V|$, no entanto, dizemos que o grafo é esparso (*sparse*), e a melhor opção para a sua representação é a lista de adjacência.

Sobre grafos – Grau de um vértice

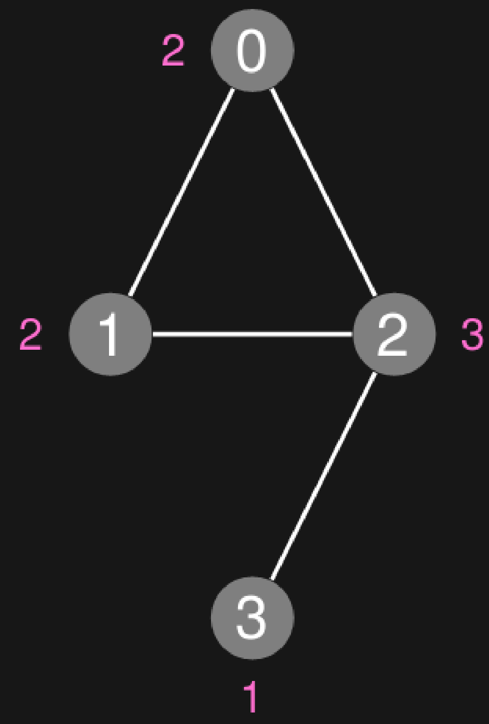
Em Grafos Não-Direcionados, o grau de um vértice é calculado pelo número de arestas incidem nele.

- Um vértice de grau zero é chamado de “isolado”, “não-conectado” ou “desconectado”.
- No exemplo abaixo, o vértice 1 possui grau 2 e o vértice 3 é desconectado.



Listas de Adjacência (Grau do vértice):

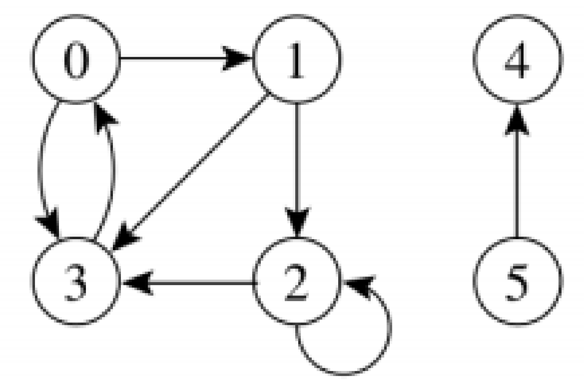
```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2

adj[0].size() // 2
adj[1].size() // 2
adj[2].size() // 3
adj[3].size() // 1
```



Sobre grafos – Grau de um vértice

Em Grafos direcionados, o grau de um vértice é o número de arestas que chegam nele (*in-degree*, ou grau de entrada) mais o número de arestas que saem dele (*out-degree*, ou grau de saída).

- No exemplo abaixo, o vértice 2 possui *in-degree* 2 e *out-degree* 2. Portanto, seu grau é 4.



Percorrendo o grafo...

Busca em profundidade (DFS, do inglês *Depth-First Search*) é um procedimento com tempo linear no tamanho de sua entrada, $O(|V|+|E|)$, que é surpreendentemente versátil e revela uma série de informações importantes sobre um grafo.

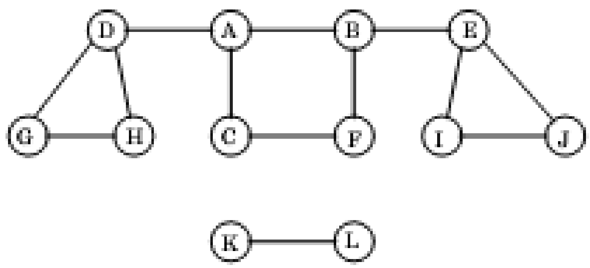
- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes conexas.

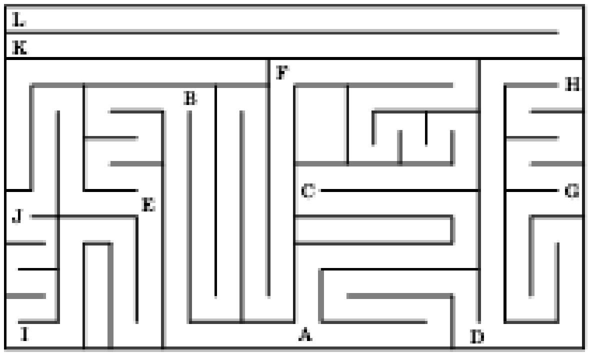
Uma das questões mais básicas que ela responde é:

- Quais partes do grafo são alcançáveis a partir de um dado vértice?

Percorrendo o grafo...

Quais partes do grafo são **alcançáveis** a partir de **um dado vértice**?





Digamos que o programa receba um grafo na forma de uma lista de adjacência. Essa representação apresenta uma operação básica: encontrar os vizinhos de um vértice. Com apenas essa primitiva, o problema da alcançabilidade é bem semelhante a explorar um labirinto.

Percorrendo o grafo...

Em um labirinto, você sempre começa a andar de um lugar fixo, mas pode se perder facilmente, andar em círculos e deixar de visitar algum ponto importante. Como os antigos resolviam esse problema? Um novelo de linha e giz!

Novelo de linha: Serve para encontrar o caminho de volta, e pode ser implementado em um programa através da recursão, que implementa implicitamente uma pilha (desenrola ou empilha para novo vértice, e enrola de volta ou desempilha quando volta para o vértice anterior)

Giz: Serve para marcar os vértices já visitados, e pode ser implementado como uma variável booleana indicando se ele já foi visitado.

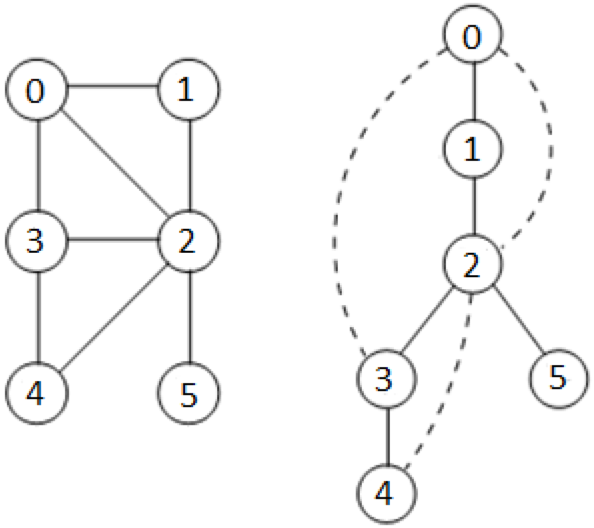
Depth-first search

```
vector<int> adj[1000];
vector<bool> visited(1000, false);

void dfs(int u) {
    if (visited[u]) {
        return;
    }

    visited[u] = true;

    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        dfs(v);
    }
}
```

Veja um exemplo no próximo slide.

```cpp
void dfs(int u)
{
    if(visited[u]) return;
    visited[u] = true;

    // visiting node
    cout << u << " ";

    for(int v : adj[u])
        dfs(v);
}
```

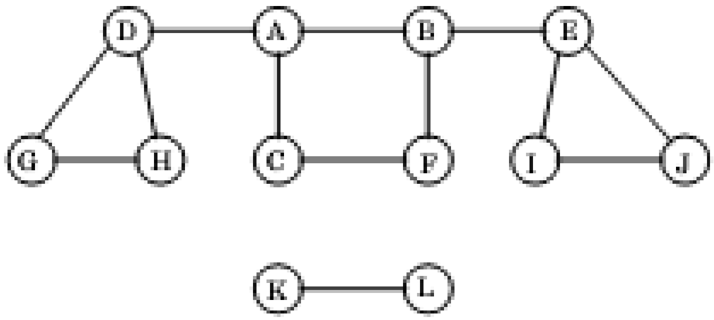
```cpp
int main()
{
    adj[0].push_back(1);
    adj[0].push_back(2);
    adj[0].push_back(3);

    adj[1].push_back(0);
    adj[1].push_back(2);

    adj[2].push_back(0);
    adj[2].push_back(1);
    adj[2].push_back(3);
    adj[2].push_back(4);
    adj[2].push_back(5);

    adj[3].push_back(0);
    adj[3].push_back(2);
    adj[3].push_back(4);

    adj[4].push_back(2);
    adj[4].push_back(3);

    adj[5].push_back(2);

    dfs(0);
}
```

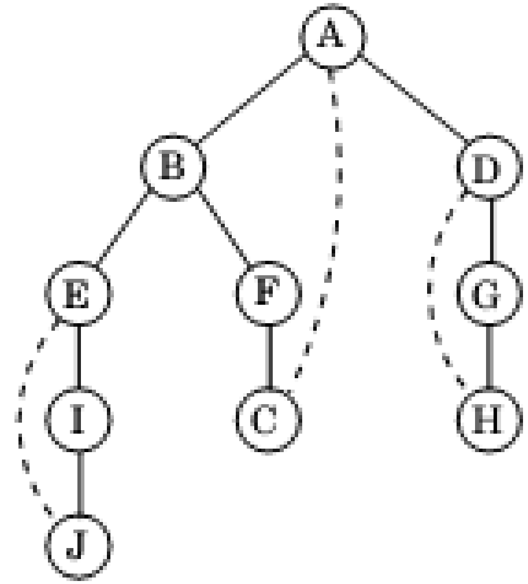


Mas esse algoritmo faz com que visitemos todos os nós do grafo?

Retire a aresta entre os vértices 2 e 5 e execute novamente o programa.

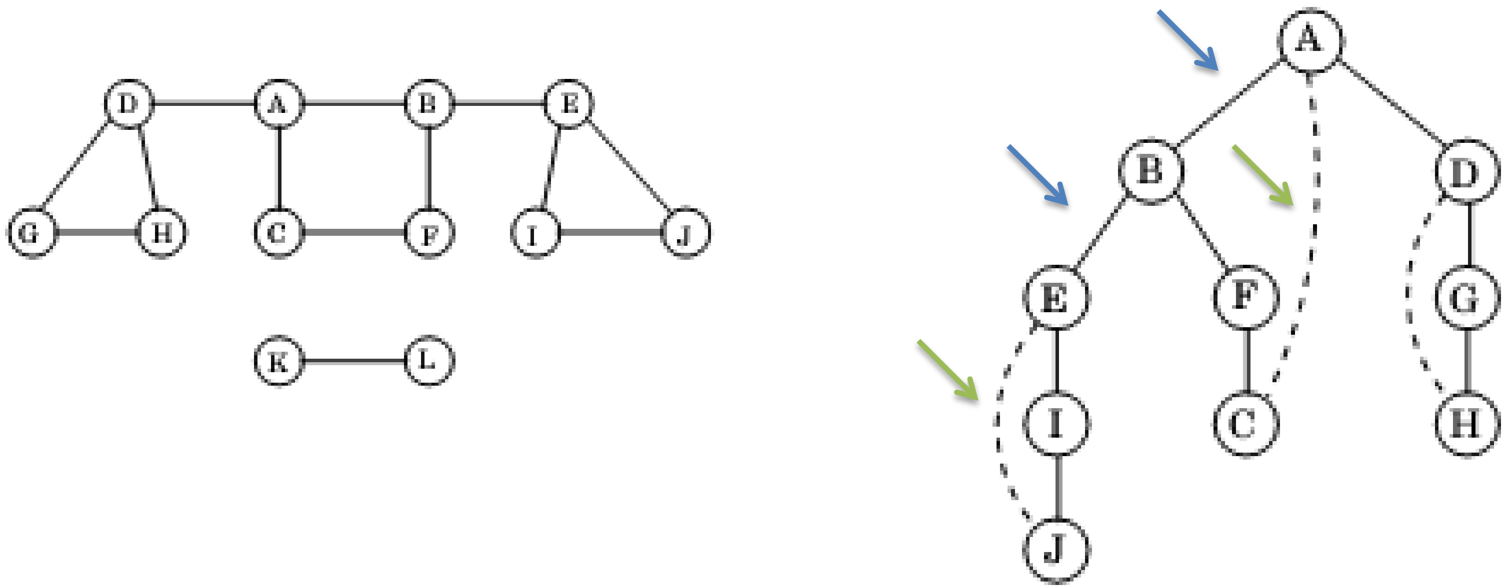
Percorrendo o grafo...





A resposta é não! O procedimento explorar visita somente a porção do grafo alcançável partindo de seu ponto de partida. Para completarmos, utilizaremos o algoritmo completo de busca em profundidade.

Percorrendo o grafo...



Antes de explorarmos o algoritmo de DFS, repare que:

- As arestas contínuas do grafo formam uma árvore (um grafo conexo sem ciclos) e são, portanto chamadas arestas de árvore.
- As arestas pontilhadas são ignoradas pois apenas levam a lugares já visitados e, portanto, são chamadas de arestas de retorno.

Percorrendo o grafo...

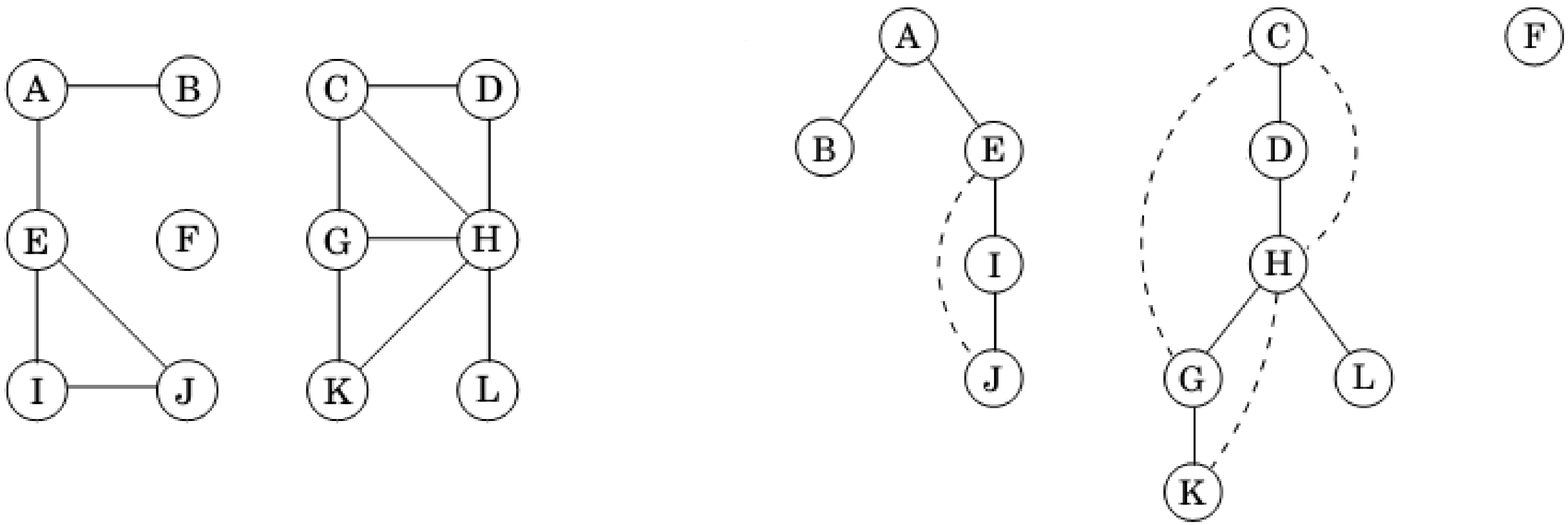
Para examinarmos o restante do grafo, é preciso recomeçar o procedimento em outro lugar: em algum vértice que ainda não tenha sido visitado.

O algoritmo de Busca em Profundidade, ou DFS (*Depth-first Search*), pode fazer isso repetidamente, até que o grafo esteja completamente explorado.

```
void dfs_explore()
{
    // none of the nodes were visited yet
    for(int i = 0; i < VERT; i++)
        if(!visited[i]) dfs(i);
}
```

Percorrendo o grafo...

Exemplo:



Resultado da exploração com busca em profundidade no grafo de 12 nós representado à esquerda:

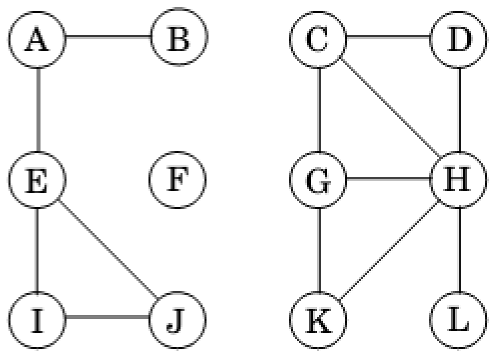
O algoritmo do DFS chama a função de exploração por três vezes, sobre A, C e F, resultando em três árvores, cada uma enraizada em seu ponto de partida. Juntas formam uma floresta.

Análise da busca em profundidade

- Podemos observar que cada vértice é **explorado** apenas uma vez, graças ao vetor **visitado**.
- Durante a exploração do vértice existem dois passos:
  1. Alguma quantidade fixa de trabalho – marcar como visitado, a pré-visita e a pós-visita.
  2. Um loop onde as arestas adjacentes são examinadas, para ver se levam a um lugar novo.
- **O trabalho total feito no passo 1 é O(|V|)**, pois é feita uma vez para cada vértice durante todo o curso da DFS.
- **No passo 2**, pelo curso de toda DFS, cada aresta {x, y} é examinada duas vezes, durante *explorar(G,x)* e *explorar(G,y)*. **O tempo total, portanto é O(|E|).**
- O **tempo total de execução do algoritmo** é portanto **O(|V| + |E|)**, linear no tamanho de sua entrada.
  - Somente a leitura de uma lista de adjacências já toma esse tempo. **Super eficiente!**

Componentes Conexas

Um grafo não-direcionado é conexo se existe um caminho entre qualquer par de vértices. O grafo do exemplo anterior não é conexo, pois, por exemplo, não há caminho entre A e G.



Este grafo possui três regiões conexas disjuntas, correspondentes aos seguintes conjuntos de vértices:

{A, B, E, I, J}

{C, D, G, H, K, L}

{F}

Essas regiões são chamadas de componentes conexas e cada uma delas é um sub-grafo internamente conexo que não possui arestas para os outros sub-grafos.

Componentes Conexas e DFS

Quando o procedimento de exploração é chamado em um determinado vértice, ele identifica precisamente a componente conexa que contém aquele vértice.

Cada vez que o loop do DFS chama explorar, uma nova componente conexa é identificada, pois são vértices que não fizeram parte da componente conexa criada na iteração anterior.

Podemos, então, adaptar de maneira muito simples o algoritmo da DFS para verificar se um grafo é conexo e, também, para atribuir a cada nó v um inteiro ccnum[v] identificando a componente conexa a que pertence com um número.

Componentes Conexas e DFS

Tudo o que é necessário é modificar o procedimento pré-visita:

<u>procedimento pré-visita</u>(v)
ccnum[v] = cc

Onde cc precisa, no início da busca, ser inicializado com zero e incrementado cada vez que a DFS chamar o procedimento explorar.

Caso, ao final da busca em profundidade, o valor de cc seja 1, o grafo é conexo. Caso contrário, cc conterá o número de componentes conexas e o vetor ccnum conterá a componente conexa à qual cada um dos vértices pertence.

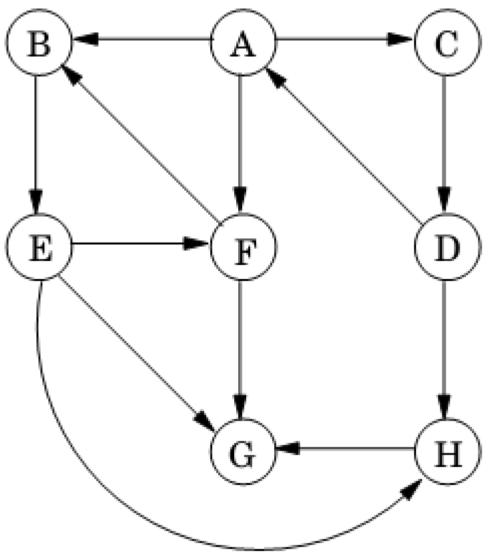
```cpp
vector<int> adj[1000];
vector<bool> visited(1000, false);
vector<int> ccnum(1000,0);

void dfs(int u, int comp)
{
    if(visited[u]) return;
    visited[u] = true;

    // setting vertex connected component
    ccnum[u] = comp;

    for(int v : adj[u])
        dfs(v, comp);
}


void dfs_explore()
{
    int cc = 0;
    // none of the nodes were visited yet
    for(int i = 0; i < VERT; i++)
        if(!visited[i]) dfs(i, cc++);
}
```

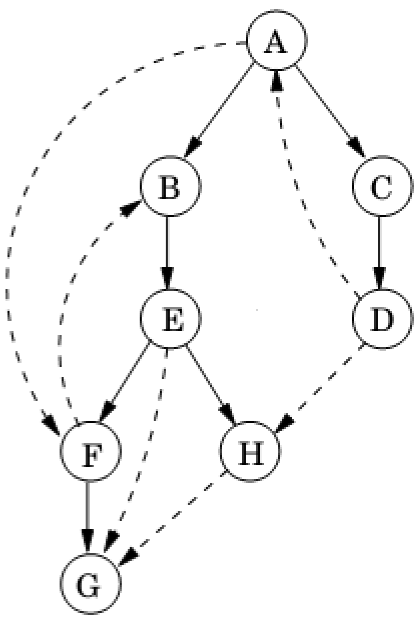
Ao final da chamada de dfs_explore, o vetor ccnum conterá o número da componente conexa de cada um dos vértices.

DFS em grafos direcionados

O algoritmo pode ser executado sem modificação para grafos direcionados, tomando sempre o cuidado para percorrer as arestas somente nas direções definidas.

Exemplo:





DFS em grafos direcionados

Para cada nó, podemos anotar o tempo de **dois eventos importantes**:

- O momento da primeira descoberta (pré-visita)
- O momento da partida final (pós-visita)

Uma maneira simples de implementar vetores de pré e pós é definir um **contador**, que é inicializado com 1 e atualizado como se segue:

<u>procedimento pré-visita</u>(v)
pré[v] = contador;
contador++;

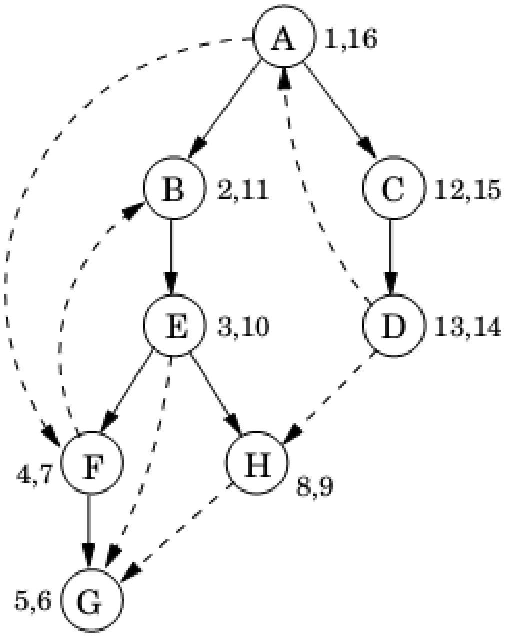
<u>procedimento pós-visita</u>(v)
pós[v] = contador;
contador++;

O intervalo [pré[u], pós[u]] revela o tempo durante o qual o vértice u esteve na pilha.

DFS em grafos direcionados

A figura abaixo mostra os valores de pré[u] e pós[u] para cada um dos vértices.

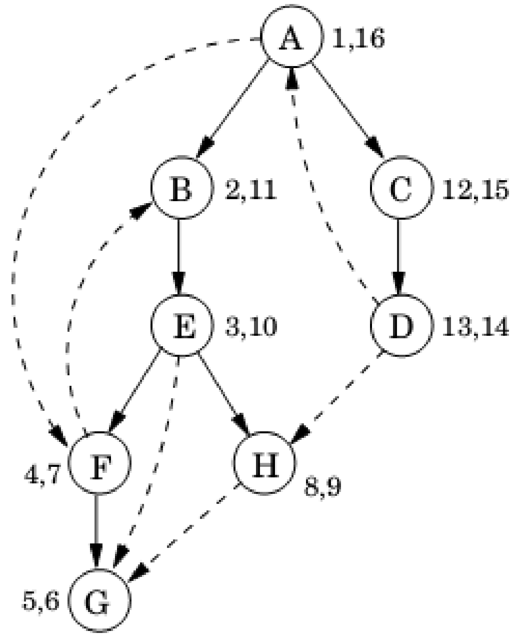
Exemplo: O quinto evento é a descoberta de G. O 14º evento consiste em deixar D definitivamente.



DFS em grafos direcionados

Em uma análise mais aprofundada do caso direcionado, é útil ter uma terminologia para os relacionamentos entre os nós da árvore

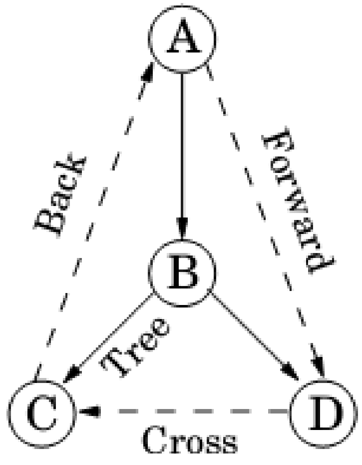
- A é a raiz da árvore de busca e todos os demais são seus descendentes.
- De forma similar, E possui F, G e H como descendentes e, de forma oposta é um ascendente desses três nós.
- C é o pai de D, que é seu filho.



DFS em grafos direcionados

Para grafos não direcionados, também distinguimos entre as arestas da árvore:

- Arestas de árvore: a aresta (u,v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u,v).
- Arestas de retorno: Levam a um ascendente na árvore DFS.
- Arestas de avanço: Conectam um vértice a um descendente que não é filho mas pertence à árvore DFS.
- Arestas de cruzamento: Não levam nem a um descendente nem a um ascendente, mas a um nó que já foi completamente explorado (pós-visitado).



DFS em grafos direcionados

- As relações de ascendência e descendência e os tipos de arestas podem ser retiradas diretamente dos números de pré e pós visita.
  - O vértice u é um ascendente de v exatamente quando u é descoberto primeiro e v é descoberto durante o explorar(u). Portanto, pré[u] < pré[v] < pós[v] < pós[u]
  - O caso de descendentes é simétrico, pois u somente é descendente de v caso v seja ascendente de u.
  - Tipos de arestas (u, v):
    - Árvore/Avanço: pré[u] < pré[v] < pós[v] < pós[u]
    - Retorno: pré[v] < pré[u] < pós[u] < pós[v]
    - Cruzada: pré[v] < pós[v] < pré[u] < pós[u]

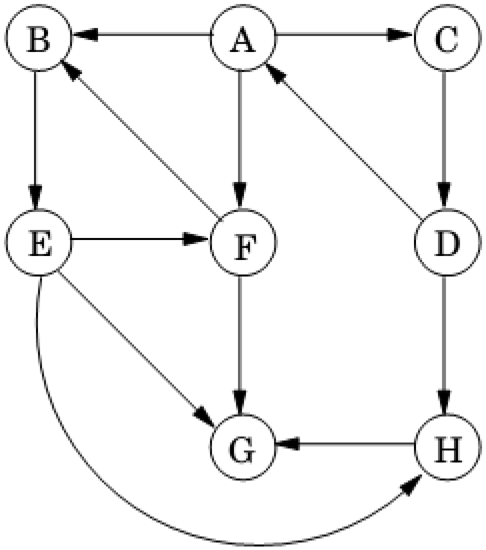
Grafos direcionados acíclicos (DAGs)

Um ciclo em um grafo direcionado é um caminho circular
$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$.
A figura anterior possui alguns exemplos: B, E, F, B, ou A, C, D, A.

Um grafo sem ciclos é dito acíclico.

Grafos Direcionados Acíclicos (ou *dags*, do inglês *directed acyclic graphs*) aparecem o tempo todo e são bons para modelar relações de causalidades, hierarquias e dependências temporais.

É possível definir se um grafo é acíclico em tempo linear, utilizando o algoritmo de busca em profundidade.



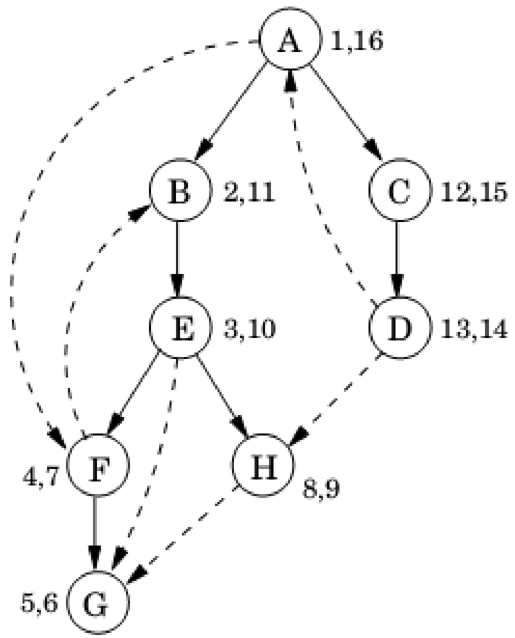
Grafos direcionados acíclicos (DAGs)

Um grafo direcionado possui um ciclo se e somente se a sua busca em profundidade revelar uma aresta de retorno.

Se (u,v) é uma aresta de retorno, existe um ciclo consistindo nesta aresta junto com o caminho de v até u na árvore.

Exemplo:

Tomando a aresta de retorno (F,B), podemos perceber que ela forma um ciclo com o caminho entre B e F: F, B, E, F



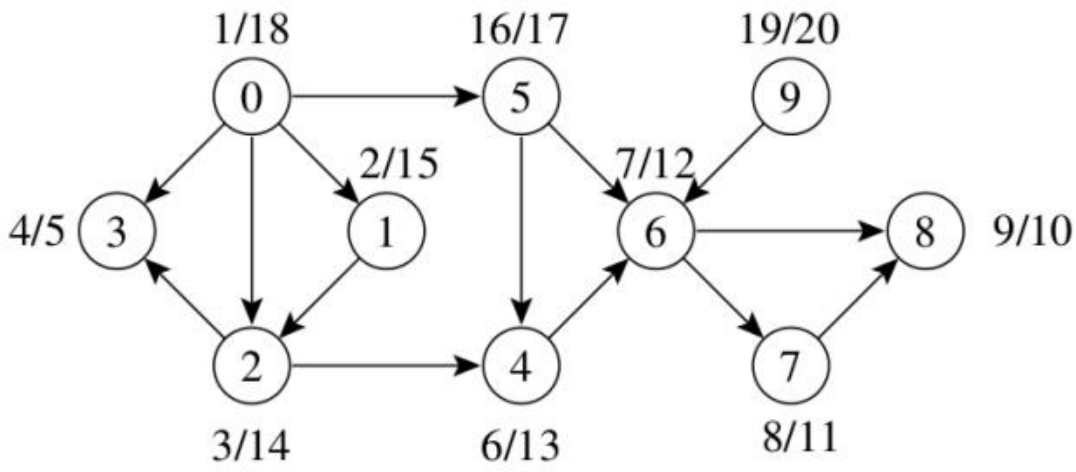
Grafos direcionados acíclicos (DAGs)

Exercício:

Modifique o algoritmo de DFS para verificar se um grafo é acíclico. (20 minutos)

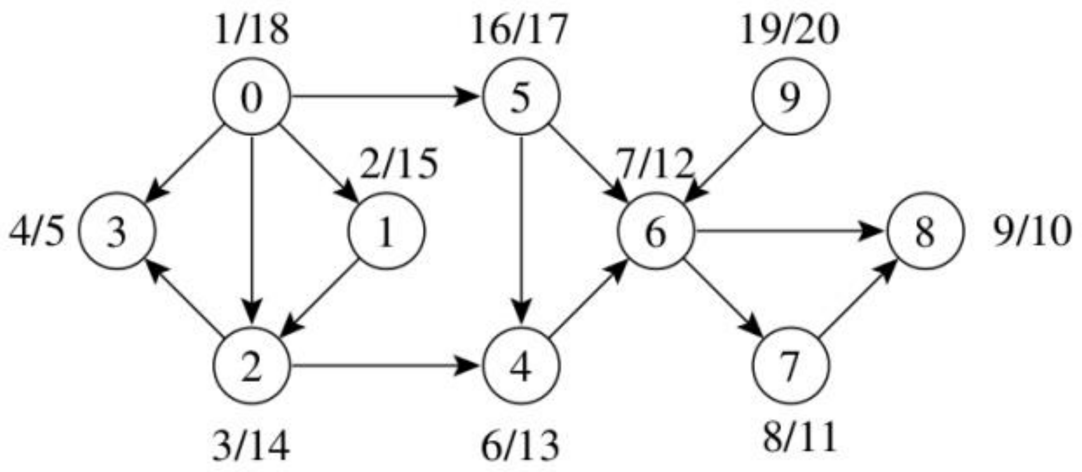
Grafos direcionados acíclicos (DAGs)

- *Dags* são utilizados com muita freqüência para indicar precedência entre eventos.
  - Imagine, por exemplo, que uma pessoa precise realizar uma grande quantidade de tarefas, mas algumas delas não podem ser feitas até que outras sejam completadas.
- Uma aresta direcionada (u,v) indica que uma atividade u precisa ser realizada antes da atividade v.



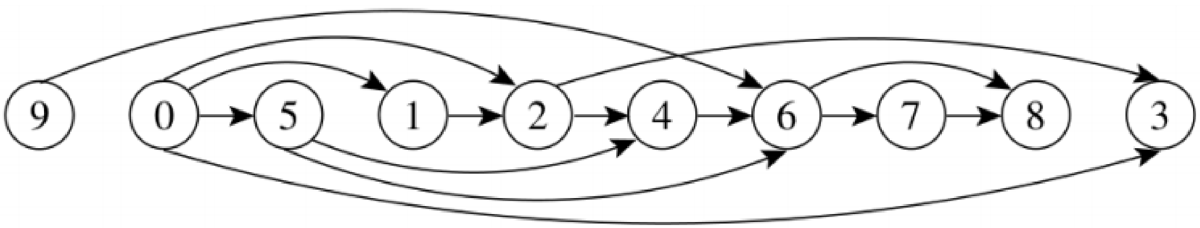
Grafos direcionados acíclicos (DAGs)

- *Dags* são utilizados com muita freqüência para indicar precedência entre eventos.
  - Imagine, por exemplo, que uma pessoa precise realizar uma grande quantidade de tarefas, mas algumas delas não podem ser feitas até que outras sejam completadas.
- Uma aresta direcionada (u,v) indica que uma atividade u precisa ser realizada antes da atividade v.



Ordenação Topológica

- Como encontrar uma ordem válida para realizar as tarefas?
- Utilizando Grafos direcionados acíclicos e o algoritmo de Ordenação Topológica.



Ordenação topológica é a ordenação linear dos vértices do grafo, de maneira que, para cada aresta direcionada (u,v), u vem antes de v na ordenação.

O **algoritmo de ordenação** consiste em:

- **Chamar a DFS** para obter os tempos de término pós[u] para cada vértice u.
- Ao término de cada vértice, **insira-o na frente de uma lista encadeada**.
- **Retorne a lista** encadeada de vértices já ordenados.

Ordenação Topológica

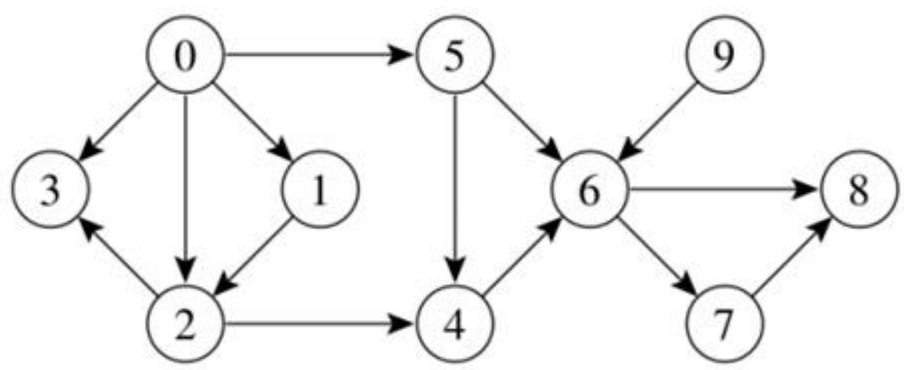
O custo é novamente linear **O (|V| + |E|)** , uma vez que a busca em profundidade tem essa complexidade de tempo e o custo para inserir cada um dos |V| vértices na frente da lista linear encadeada custa O (1).

Basta realizar uma chamada para o procedimento **de inserção na lista** no procedimento **dfs**, logo após o momento onde pós[u] for determinado. Imprima a lista ao final da execução.

Grafos direcionados acíclicos (DAGs)

Exercício:

Modifique o algoritmo de DFS para realizar a ordenação topológica de um grafo acíclico. Utilize o grafo da figura como exemplo:



Componentes Fortemente Conexas

- A definição de **conectividade para grafos direcionados** é um pouco mais sutil do que para grafos não-direcionados.
- Neles, dois nós u e v estão conectados se existe um caminho de u até v e um caminho também de v até u.
- Essa relação particiona V em conjuntos disjuntos, que chamamos de **Componentes Fortemente Conexas**. Veja o exemplo: