Aula 08:

Algoritmos em Grafos

Parte I

PCO001 – Algoritmos e Estruturas de Dados
Prof. João Paulo Reus Rodrigues Leite
joaopaulo@unifei.edu.br

Podemos expressar muitas situações e problemas do mundo real através da estrutura de dados conhecida como **grafo**.

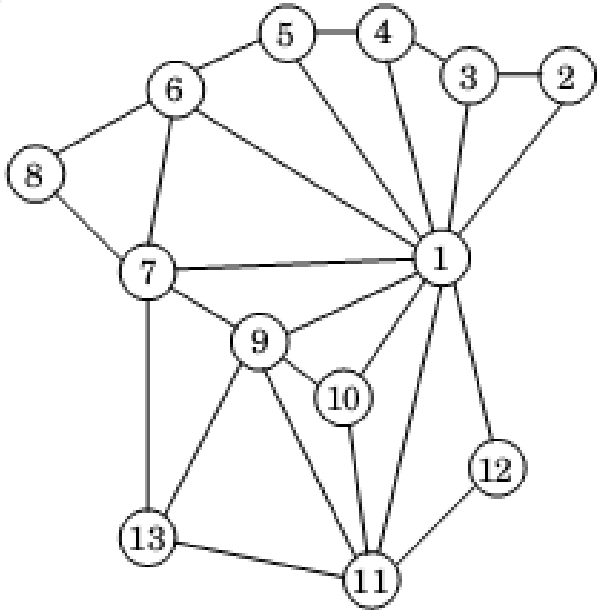
Geralmente, a utilização deste recurso deixa claro o cerne do problema, que passa a ser representado de maneira gráfica e concisa, ou seja, sem informação supérflua.

- 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 um sistemas de tubos;
- **Controle de tráfego** em uma cidade;
- Interações em **redes sociais**, etc.
- **Coloração** de um mapa político, sem que nenhuma unidade possua fronteira com outro de mesma cor;

Problema: Coloração de mapas políticos

(a)

(b)



Problema: Qual é o **número mínimo** de cores para que nenhum país faça fronteira com outro de mesma cor?

O mapa da figura (a) está **saturado com informação irrelevante.** Enquanto isso, a figura (b) representa apenas a informação necessária através de um grafo, onde os **vértices** são os países e as **arestas** são as fronteiras.

Um algoritmo que dá uma boa solução para o problema de coloração e vértices é o Algoritmo de Welsh-Powell.

Seus passos são os seguintes:

1. Encontre a **valência** (ou grau) de cada vértice.
2. Liste os vértices em **ordem descendente** de valência (podendo quebrar empates da maneira que preferir).
3. Pinte o **primeiro vértice da lista** (maior valência) com a cor 1.
4. Desça na lista e **pinte com a mesma cor** todos os vértices que não estão conectado aos vértices já coloridos acima dele.
5. **Remova** todos os vértices coloridos da lista.
6. **Repita o processo** nos vértices ainda não coloridos utilizando uma nova cor – sempre trabalhando na ordem descendente de valência até que todos os vértices tenham sido coloridos.

A ideia é que, ao iniciar com os vértices que possuem maiores valências, você terá cuidado primeiro dos vértices mais propensos a gerar conflito.

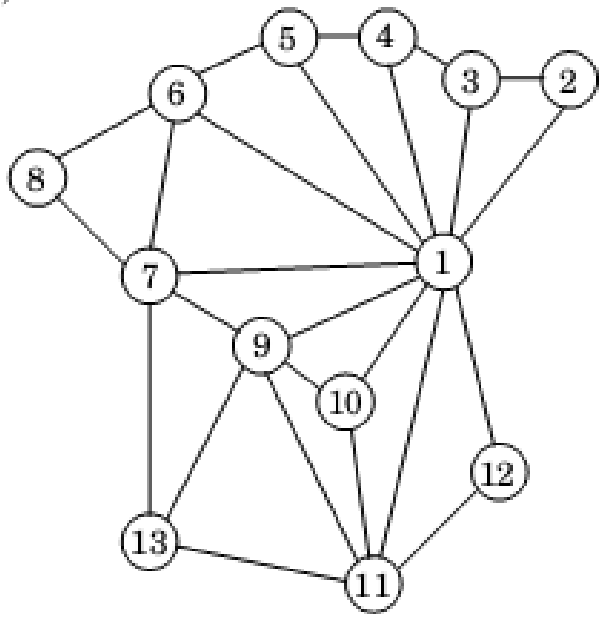
*Hands-**On**!*

Aplique o algoritmo de Welsh-Powell no grafo abaixo e responda:

1) Quantas cores são necessárias para pintar todos os vértices (países)?
2) Quais são as cores de cada vértice (país)?

(a)

(b)



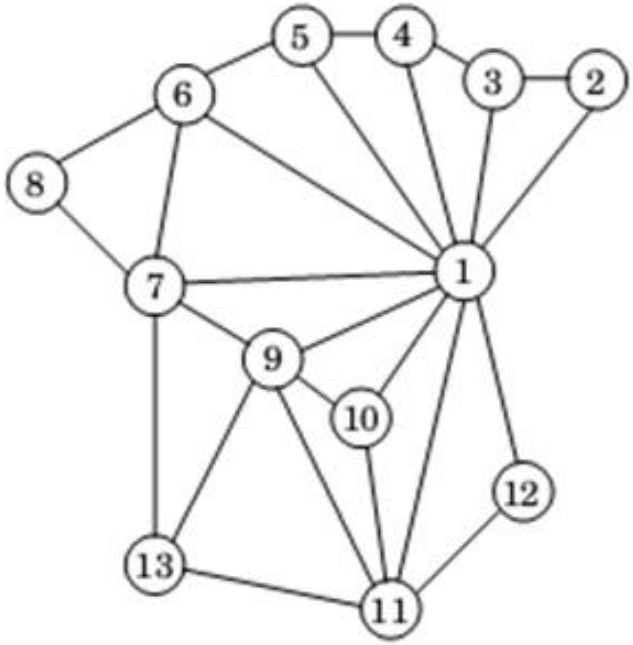
Mas **colorir mapas** é tão importante assim?

Na verdade, é possível representar problemas de contextos completamente diferentes como o problema de coloração. Veja um exemplo:

- Uma universidade precisa agendar exames para todas as suas turmas e quer utilizar o menor número possível de horários.
  - Dois exames não podem ser agendados ao mesmo tempo se algum aluno irá fazer os dois.
- Podemos utilizar um **vértice** para cada exame e colocar **arestas** entre exames que possuam conflito (algum aluno que participará dos dois).
- Se eu rotular **cada horário com uma cor**, atribuir horários torna-se exatamente a mesma tarefa de colorir grafos com a menor quantidade possível de cores.

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.

Em muitos casos, grafos representam cenários onde a relação entre os vértices não é necessariamente simétrica (“ida e volta”). Neste novo caso, o caminho de um vértice a outro não implica automaticamente em um caminho no sentido contrário.

Para esta finalidade, existem os grafos direcionados (ou dígrafos).

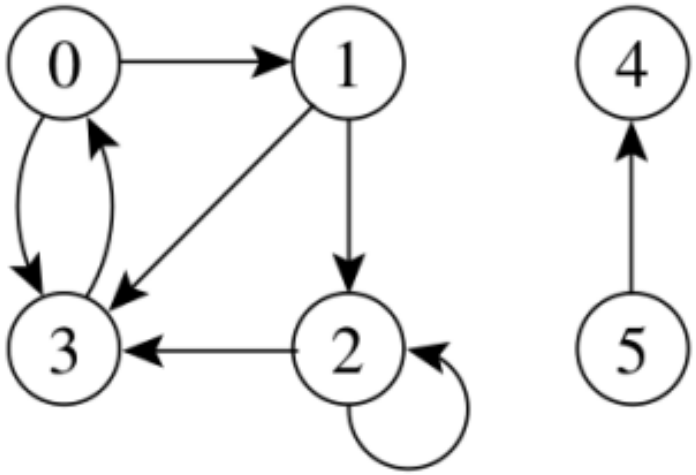
Escrevemos:

- 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, no entanto, precisa ser explícita.

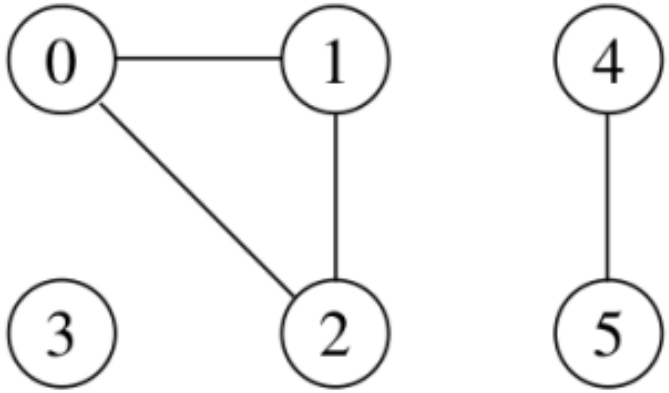
Grafos **Direcionados**

- 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*.



Grafos **Não-Direcionados**

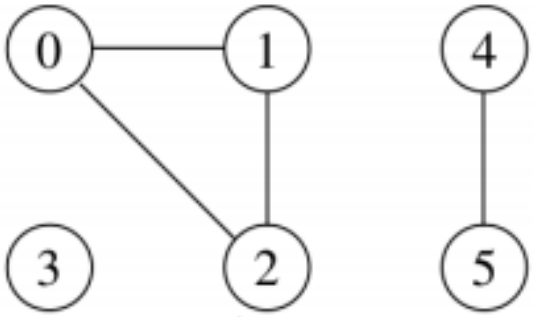
- As arestas (u, v) e (v, u) são consideradas a mesma aresta e podem ser representadas por **{u,v}**.
- A relação de adjacência é simétrica, ou seja, u é adjacente a v, que também é adjacente a u.
- Não podem existir arestas de um vértice para ele mesmo, ou *self-loops*.



Grau do Vértice

Em **Grafos Não-Direcionados**, o **grau** de um vértice (ou **valência**) é dado pelo número de arestas que incidem nele.

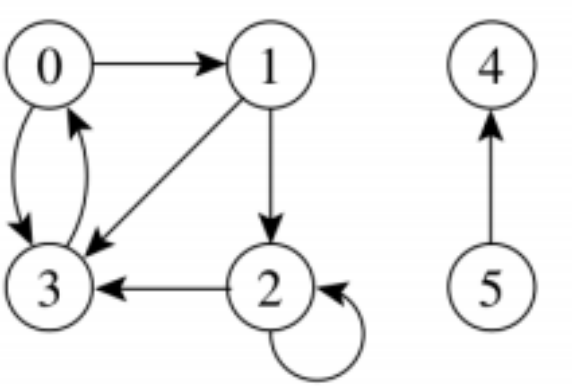
- Um vértice de grau zero é chamado de “isolado”, “não-conectado” ou “desconectado”.
- No exemplo abaixo, o vértice 1 possui grau 2 e o vértice 3 é desconectado (grau zero).



Grau do 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 i*n-degree* 2 e *out-degree* 2. Portanto, seu grau é 4.
- O vértice 1 possui *in-degree* 1 e *out-degree* 2. Seu grau é 3.

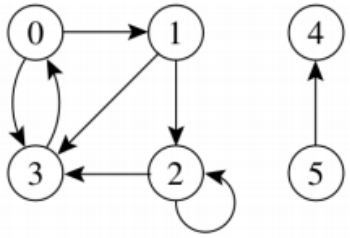


E como **representar um grafo** em linguagem de programação?

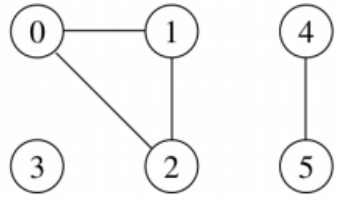
E como **representar** um grafo?

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					1	



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

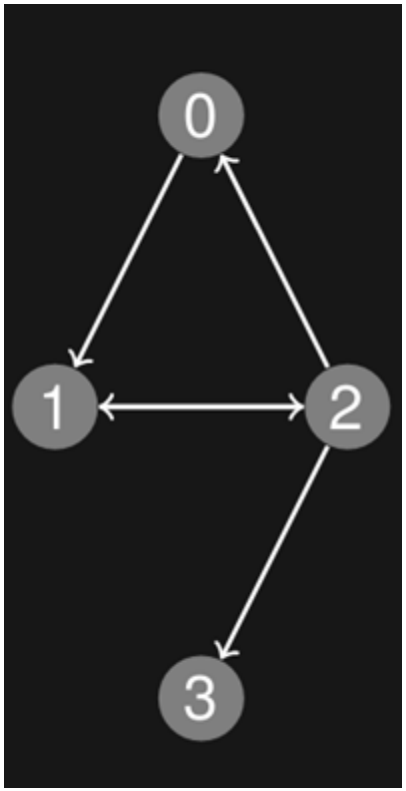
E se as aresta tiverem “**pesos**”?

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 **|V|** de vértices ou **O(|V|²)**, e, portanto, ler ou examinar a matriz tem complexidade de tempo **O(|V|²)**, ou $O(n^2)$ para n == |V|.

```cpp
#include <iostream>
#include <cstring>
#define VERT 4
using namespace std;

int main() {
  int mat_adj[VERT][VERT];
  memset(mat_adj, 0, sizeof mat_adj);

  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;

  for(int i = 0; i < VERT; i++) {
    for(int j = 0; j < VERT; j++)
      cout << mat_adj[i][j] << " ";
    cout << "\n";
  }
}
```

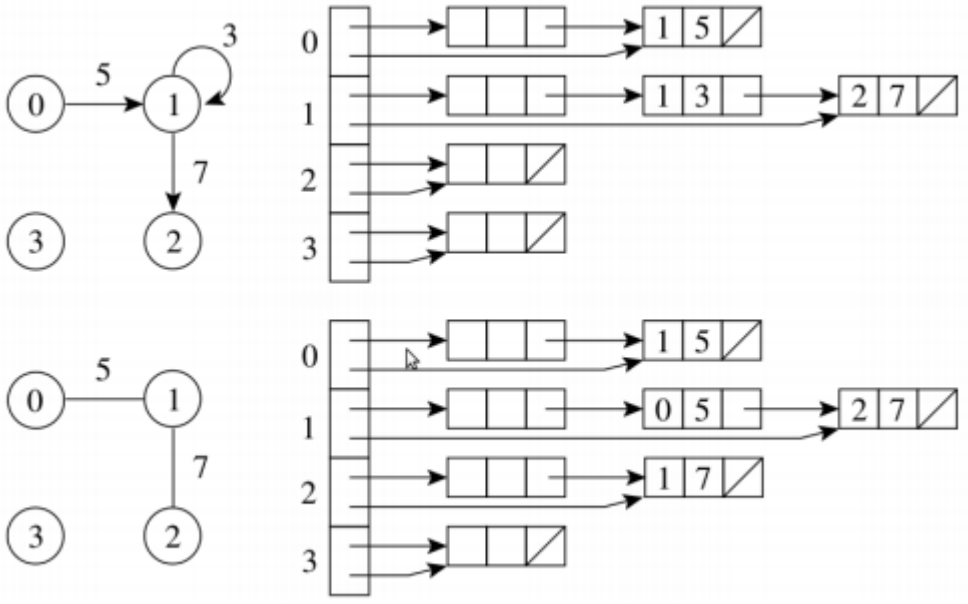



```
0 1 0 0
0 0 1 0
1 1 0 1
0 0 0 0
```

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 vértice **u**, lista[u] contém todos os vértices adjacentes a **u** no grafo G.

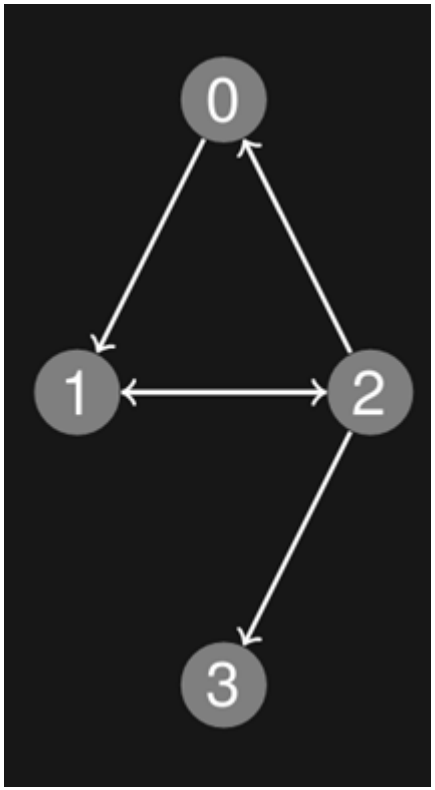


Como **representar** um grafo?

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. Portanto, percorrer todo o grafo leva, no pior caso, tempo **O(|E|)**.
- Tamanho total da estrutura de dados é Θ(|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|)**, no pior caso, pois podem haver |V| arestas saindo de u (conectado a todos os demais).

```cpp
#include <iostream>
#include <vector>
#define VERT 4
using namespace std;

int main() {
  vector<int> adj[VERT]; // C++ STL vector
  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);

  for(int i = 0; i < VERT; i++) {
    cout << "Lista[" << i << "]: ";
    for(auto elem : adj[i])
      cout << elem << " ";
    cout << "\n";
  }
}
```




```
Lista[0]: 1
Lista[1]: 2
Lista[2]: 0 1 3
Lista[3]:
```

E se as aresta tiverem “**pesos**”?

Qual é a **melhor** representação?

Qual a **melhor** representação?

Depende da relação entre |V|, que é 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 |V| (ou menor), ou tão grande quanto $|V|^2$. Portanto:

- 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.

Percorrendo **Grafos**

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 bastante versátil e revela várias informações importantes sobre um grafo.

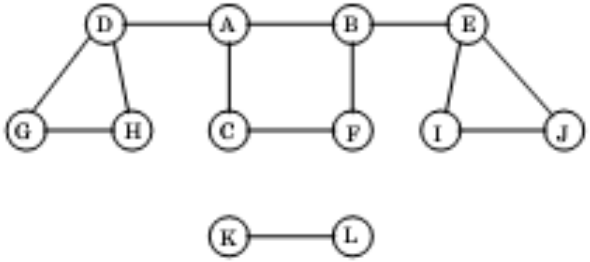
> O algoritmo é base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes conexas.

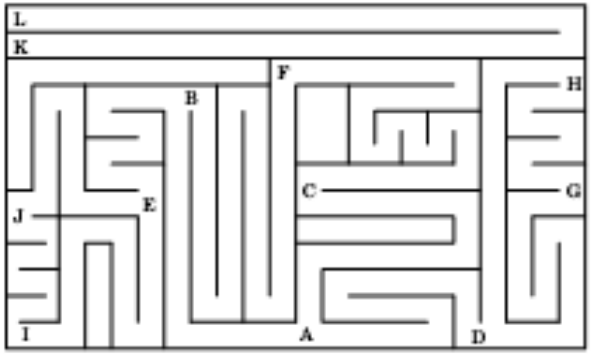
Uma das questões básicas que essa busca responde é:

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

Explorando **Labirintos**

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.

Explorando **Labirintos**

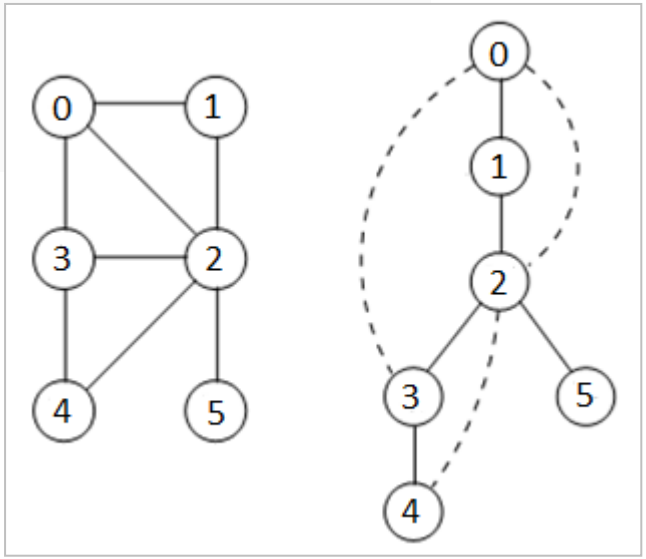
Em um labirinto, você sempre começa a andar a partir de um lugar fixo, mas pode se perder facilmente, andar em círculos e deixar de visitar algum ponto importante. Como os antigos resolviam esse problema? Um novelo de linha e giz!

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.

```cpp
#include <iostream>
#include <vector>
#define VERT 6
using namespace std;

// grafo como variavel global
vector<int> adj[VERT]; // grafo como lista de adj.
vector<bool> visited(VERT, false); // foi visitado?

// busca em profundidade começando no vértice u
void dfs(int u) {
  if(visited[u]) return; // já foi visitado? Sai.
  visited[u] = true;

  // para cada elemento v, conectado a u, chame dfs recursivamente.
  for(int i = 0; i < adj[u].size(); i++) {
    int v = adj[u][i];
    dfs(v);
  }
}
```

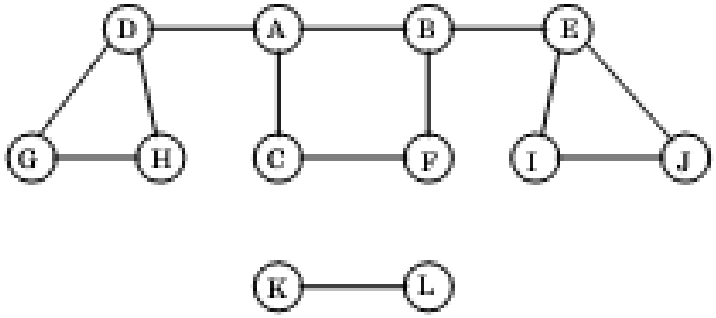
```
void dfs(int u) {
  if(visited[u]) return; // já foi visitado? Sai.
  visited[u] = true;
  cout << u << " - "; // imprimo o vértice no ato da visita.

  // para cada elemento v, conectado a u, chame dfs recursivamente.
  for(int i = 0; i < adj[u].size(); i++) {
    int v = adj[u][i];
    dfs(v);
  }
}
```

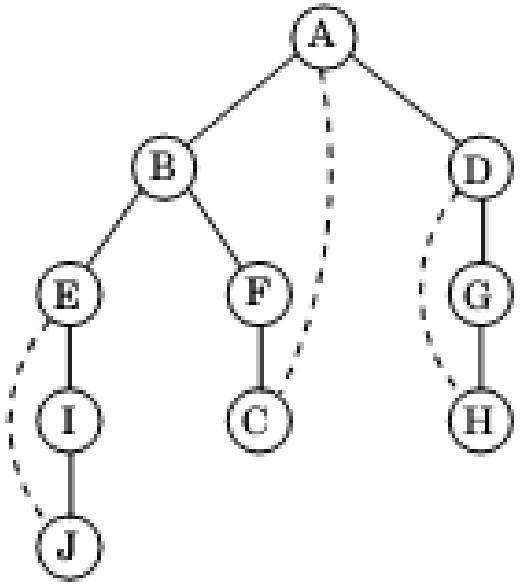


Se colocarmos um simples procedimento de impressão no início de cada visita, conseguimos ver a **ordem** em que os vértices são visitados. Veja o resultado para o grafo ao lado:

```
C:\Users\joaop\Desktop\src>dfs
0 - 1 - 2 - 3 - 4 - 5 -
```

Mas a DFS faz com que sempre visitemos todos os nós do grafo?



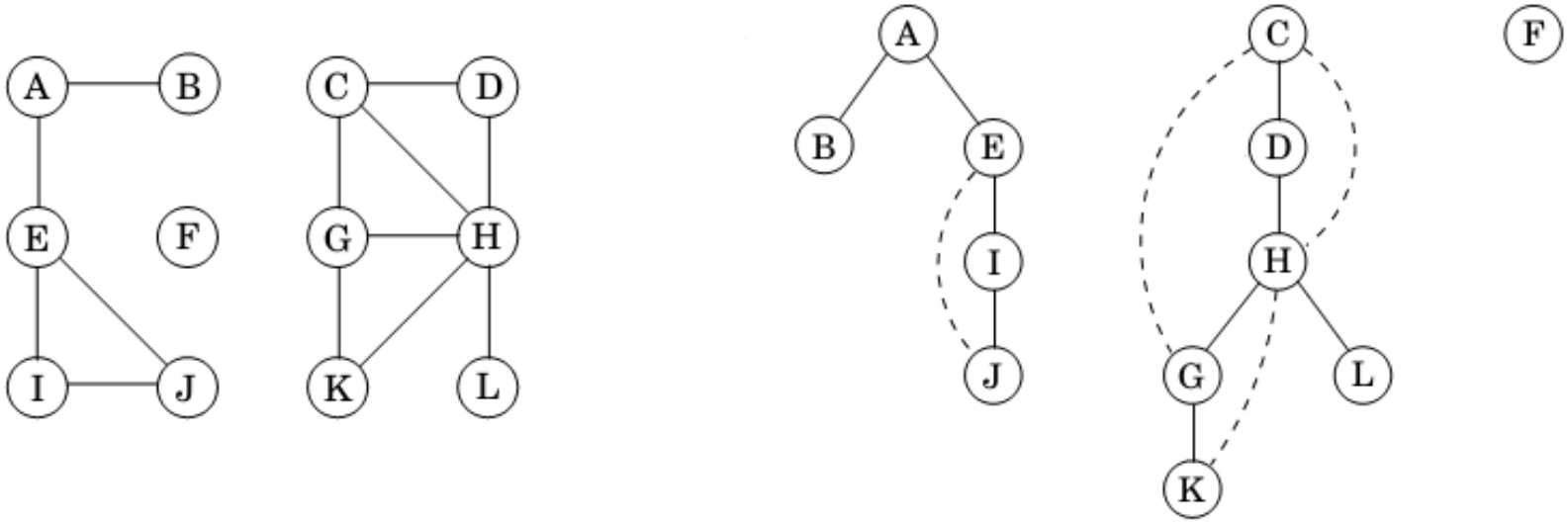


Não! A chamada da função explora a porção do grafo que é alcançável a partir de um determinado vértice inicial. Se quisermos visitar TODO o grafo, precisamos escrever outra função.

Para examinarmos o restante do grafo, é preciso **recomeçar a busca em profundidade** (DFS) a partir de outro vértice, entre aqueles que ainda não tenham sido visitados.

O algoritmo pode fazer isso repetidamente, até que o grafo esteja completamente explorado.

```
// explora todo o grafo utilizando dfs
void dfs_explore() {
  for(int i = 0; i < VERT; i++)
    if(!visited[i]) dfs(i);
}
```



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

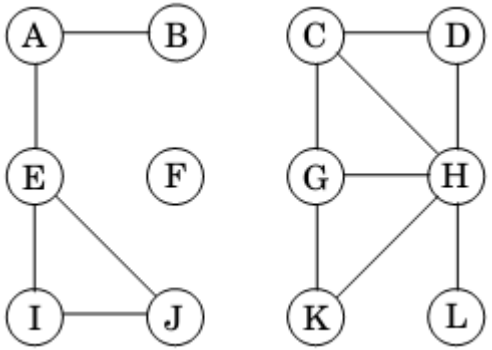
O algoritmo do DFS chama a função DFS por três vezes, partindo dos vértices A, C e F, o que resulta em três árvores, cada uma enraizada em seu ponto de partida. Juntas formam uma “floresta”.

E qual é o **custo**?

- Cada vértice é **explorado** apenas uma única vez, graças ao vetor **visitado**.
- Durante a exploração do vértice existem dois passos:
  1. Alguma quantidade fixa de trabalho – marcar como visitado, imprimir, ou algum outro procedimento pré e pós-visita.
  2. Um laço ***for*** onde as arestas adjacentes são examinadas, para verificar 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 uma (dígrafo) ou duas vezes (grafo não direcionado), durante *dfs(x)* e *dfs(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.
- A própria leitura de uma lista de adjacências já toma tempo semelhante. Portanto, a DFS é muito **eficiente!**

Componentes Conexas

Um grafo não-direcionado é conexo se existe um caminho entre qualquer par de vértices. O grafo do exemplo abaixo 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 a busca em profundidade é chamada a partir de um determinado vértice, ela identifica os vértices que compõem a componente conexa daquele vértice.

Cada vez que o laço da função **dfs_explore** chama a DFS, uma nova componente conexa é identificada, pois são vértices que não fizeram parte da componente conexa criada na iteração anterior.

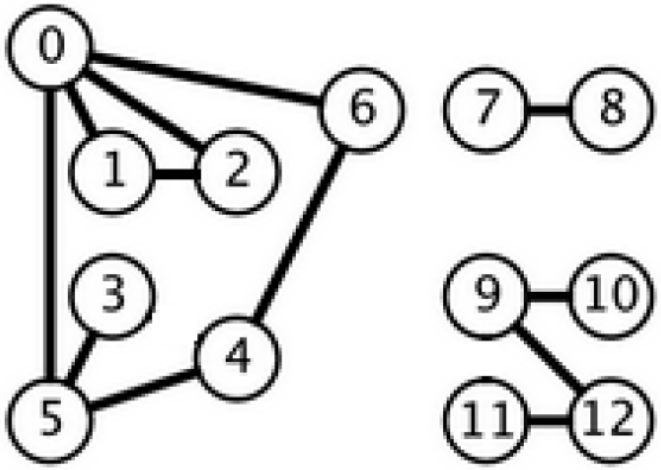
Podemos, então, adaptar de maneira muito simples o algoritmo da DFS para verificar se um grafo é conexo e, também, para atribuir a cada nó v um inteiro *ccnum* identificando a componente conexa a que pertence com um número.

```cpp
vector<int> adj[VERT]; // grafo como lista de adj.
vector<bool> visited(VERT, false); // foi visitado?
vector<int> ccnum(VERT,0);

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

  // seta a componente conexa do vértice
  ccnum[u] = comp;

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

void dfs_explore() {
  int cc = 0;
  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, e cc terá o número de componentes conexas do grafo.

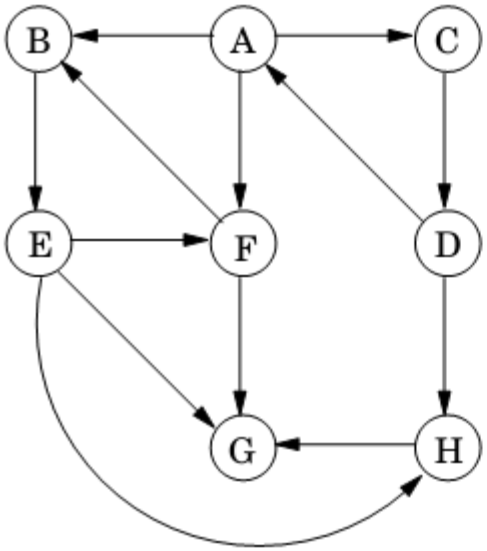
*Hands-**On**!*

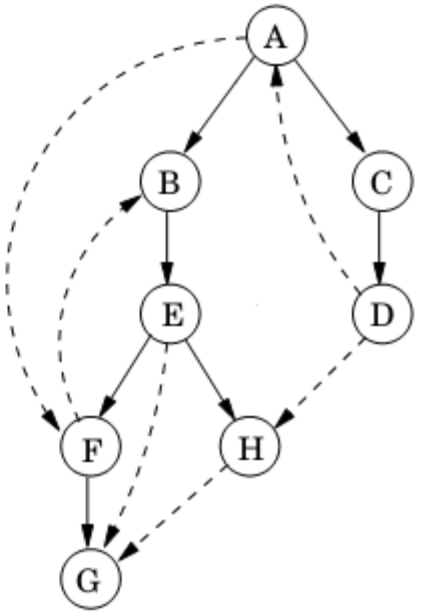
Simule o algoritmo de busca em profundidade no grafo abaixo. Começando no vértice 0, todos os vértices do grafo são visitados? Faça o teste utilizando seu código escrito em C++. Modifique o código para que todo o grafo seja explorado e ele imprima a quantidade de componentes conexas e, além disso, a qual componente cada vértice pertence.



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.





Grafos Direcionados **Acíclicos** (**DAG**)

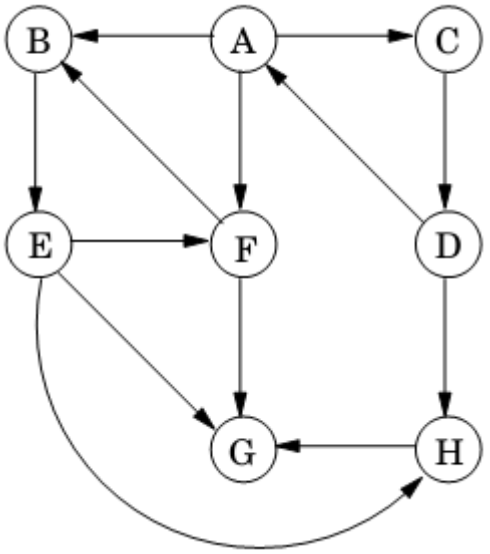
Um ciclo em um grafo direcionado é um caminho circular $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$.

A figura abaixo 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.

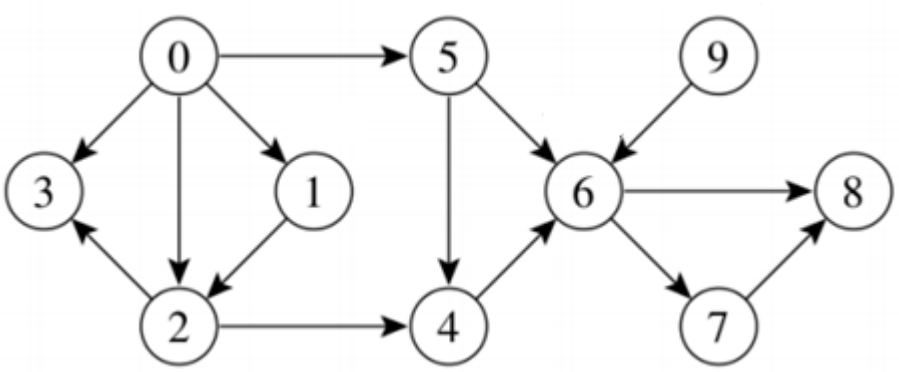
É possível definir se um grafo é acíclico em tempo linear, utilizando o algoritmo de busca em profundidade. Pesquise sobre isso!



Dags são utilizados com muita freqüência para indicar precedência entre eventos.

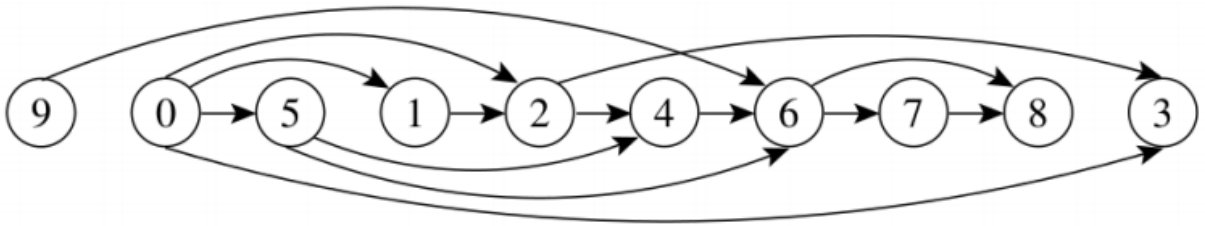
- Imagine, por exemplo, que uma pessoa precise realizar uma grande quantidade de tarefas para gerar um produto, mas algumas delas não podem ser feitas até que outras sejam completadas.

Uma aresta direcionada (u,v) indica que uma atividade u precisa ser realizada antes da atividade v.



Como encontrar uma **ordem válida** para realizar as tarefas?

Utilizando Grafos direcionados acíclicos e o algoritmo de Ordenação Topológica.



O **algoritmo de ordenação** consiste em:

- **Chamar a função** dfs_explore.
- Ao término da visita de cada vértice, **insira-o na frente de uma lista encadeada**.

Ao final da execução, a lista encadeada terá as tarefas (vértices) na ordem em que precisam ser realizadas para que não haja conflitos de precedência.

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 **laço for**. Imprima a lista ao final da execução.

Os elementos demoram para entrar na lista quando tem outros que dependem deles, e entram imediatamente no caso contrário. Portanto, os últimos a entrar precisam ser os primeiros a serem executados.

Como **exercício**, escreva o código para ordenação topológica e teste no grafo do slide anterior.