Grafos – Parte 1

Prof. **João Paulo** R. R. Leite
joaopaulo@**unifei**.edu.br
ECOE44 & ECOE45 – Maratona de Programação

Veremos...

1) **Conceitos básicos** sobre grafos.

2) **Explorando** Grafos:

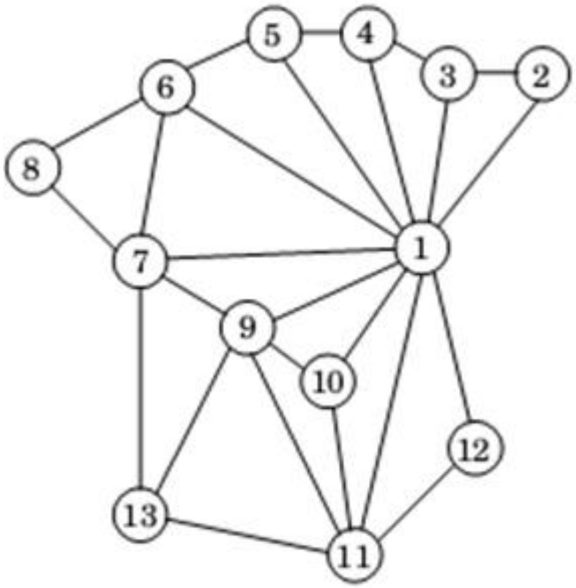
- DFS (Busca em profundidade);
- Encontrando componentes conexas;

Muitos **problemas podem ser expressos** 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.

Grafo: Vértices + Arestas

Notação: G = (V, E)



No exemplo ao lado, temos vários vértices {1, 2, 3 ... 13} e arestas, como {1,2}, {9,11}, {7,13}.

O exemplo é um grafo não-direcionado, pois nele, as arestas possuem uma relação de simetria (*mão-dupla*): é possível “caminhar” de x para y e de y para x.

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.

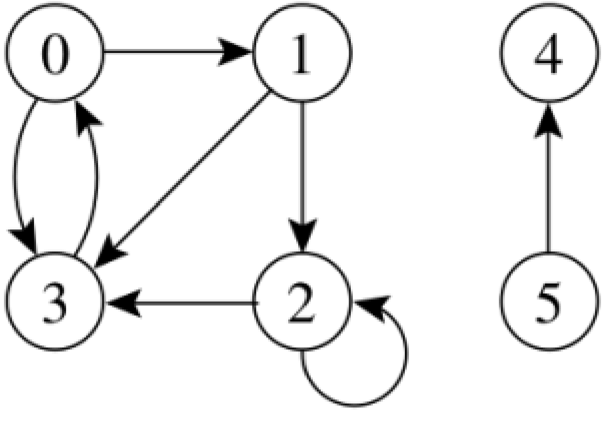
Escrevemos:

- 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.

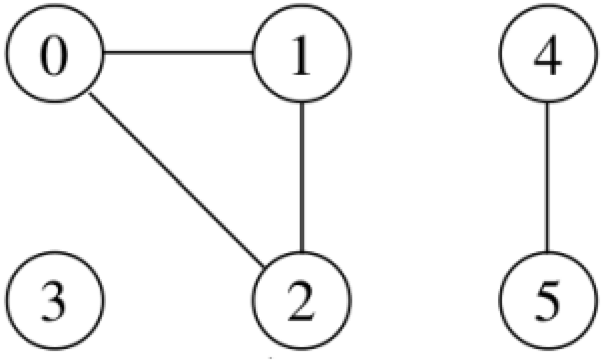
Grafos **Direcionados**

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



Grafos **Não-Direcionados**

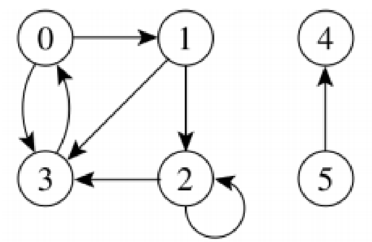
- As arestas (u, v) e (v, u) são consideradas a mesma aresta, que é representada por {u,v}.
- A relação de adjacência é simétrica, ou seja, u é adjacente a v, que também é adjacente a u.
- Não podem existir arestas de um vértice para ele mesmo, ou *self-loops*.

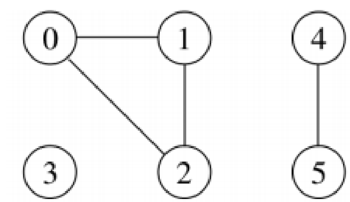


Como **representar** um grafo?

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

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

Como **representar** um grafo?

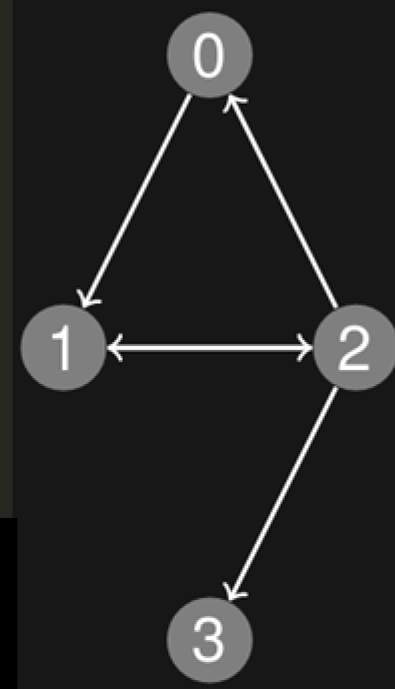
Matriz de Adjacência:

- 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:

```
#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
```

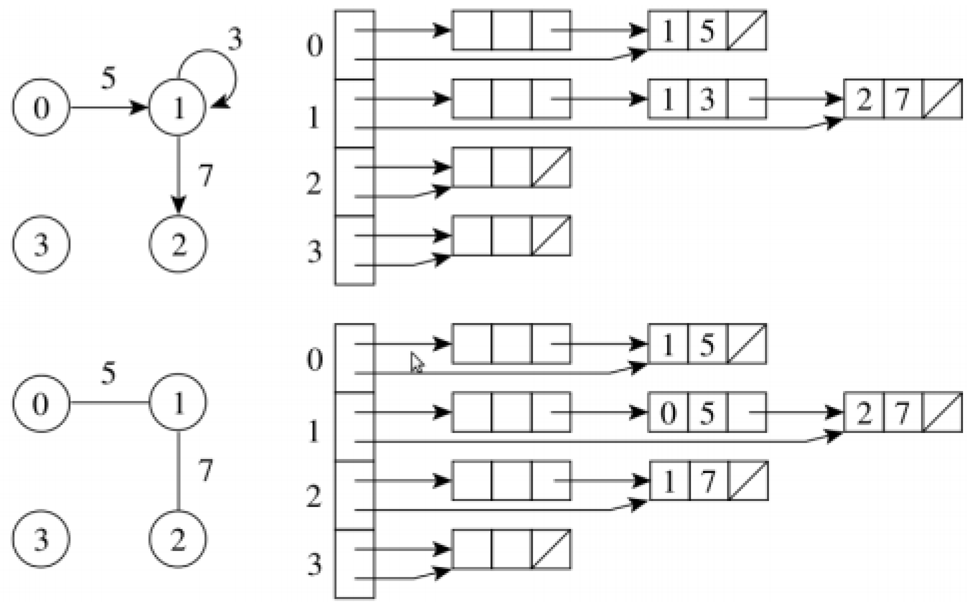


A função **memset** é da biblioteca **`<cstring>`**

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* pertencente ao conjunto V, a lista[u] contém todos os vértices adjacentes a u em G.



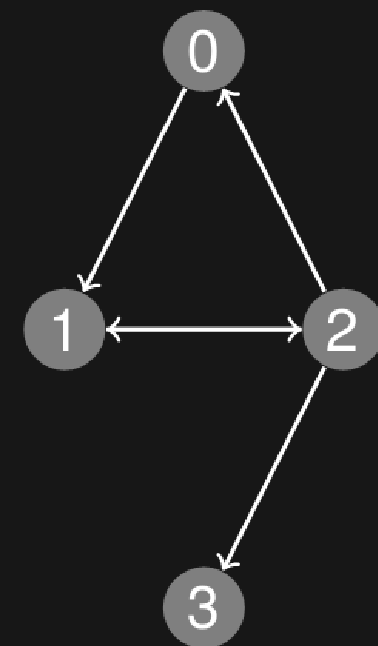
Como **representar** um grafo?

Listas de Adjacência:

- Cada aresta aparece em exatamente uma das listas 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);
```



A classe **vector<T>** é da biblioteca **`<vector>`**,
como vocês devem se lembrar!

Qual a **melhor representação**?

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

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

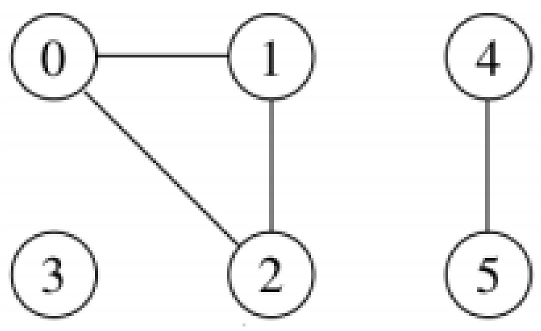
- Quando $|E|$ estiver perto do limite superior desse intervalo, dizemos que o grafo é denso (*dense*), e a melhor opção para a sua representação é a matriz de adjacência.
- Quando $|E|$ estiver perto de $|V|$, no entanto, dizemos que o grafo é esparso (*sparse*), e a melhor opção para a sua representação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

Grau de um vértice

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

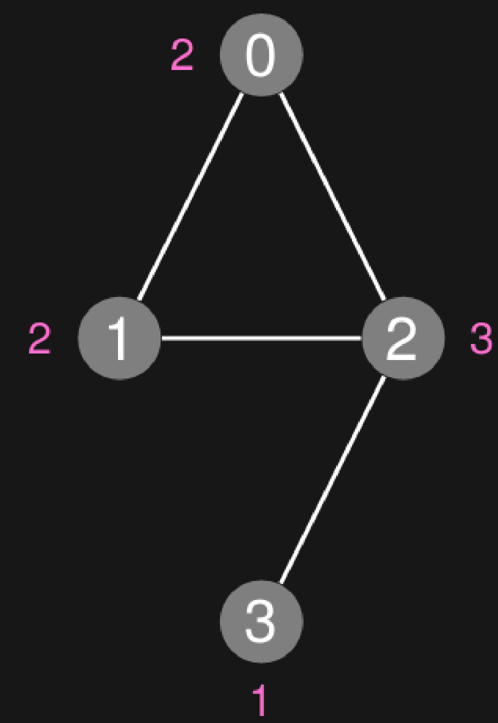
- Um vértice de grau zero é chamado de “isolado ou “desconectado”.

No exemplo abaixo, o vértice 1 possui grau 2 e o vértice 3 é desconectado.



Listas de Adjacência (**Grau** do vértice):

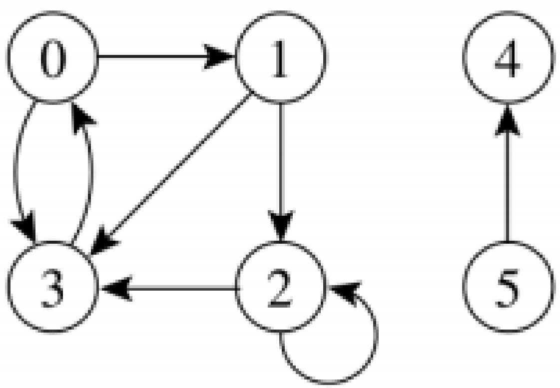
```
0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2

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



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 - **DFS**

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

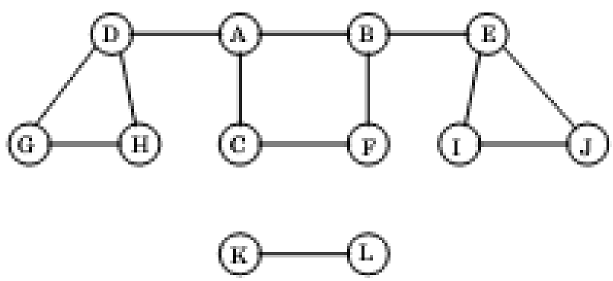
- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes conexas.

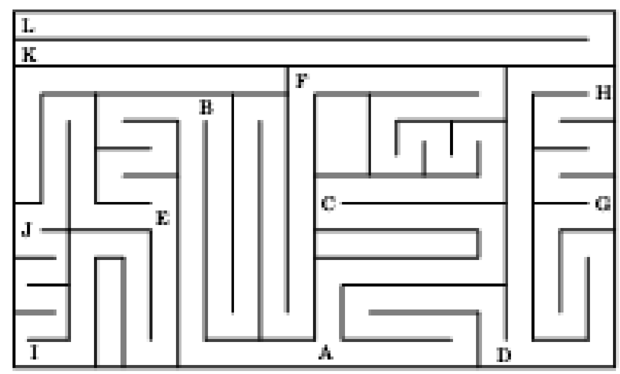
Uma das questões mais básicas que ela responde é:

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

Percorrendo o grafo - **DFS**

Quais partes do grafo são **alcançáveis** a partir de **um dado vértice**? Onde eu consigo chegar?





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 (adjacentes). O problema da alcançabilidade é bem semelhante à **exploração de um labirinto**.

Percorrendo o grafo - **DFS**

Em um labirinto, você começa a caminhar a partir de um lugar fixo, mas pode se perder facilmente, andar em círculos e deixar de visitar algum ponto importante. Como este problema era resolvido nas antigas histórias e na mitologia? 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 (*true/false*) indicando se ele já foi visitado.

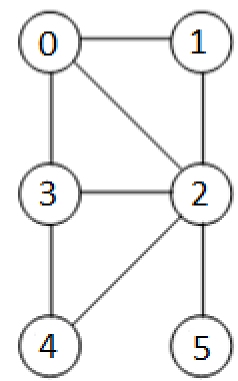
Depth-first search

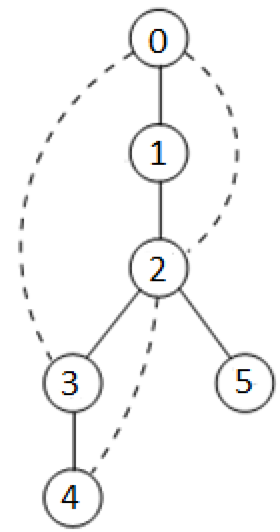
```cpp
vector<int> adj[1000];
vector<bool> visited(1000, false);

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);
}
```

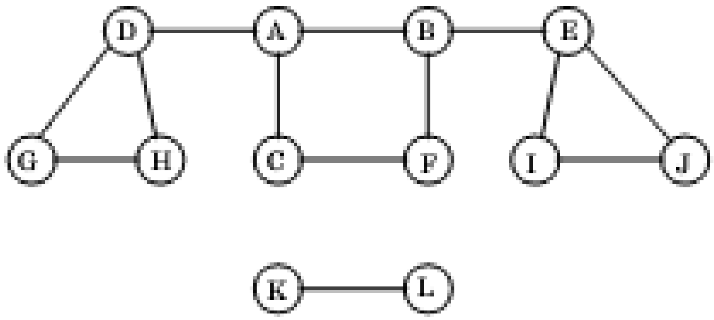


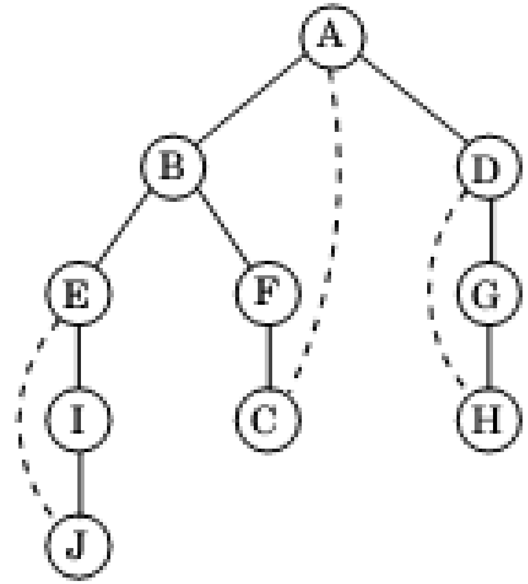


Mas esse algoritmo faz com que visitemos todos os nós do grafo?

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

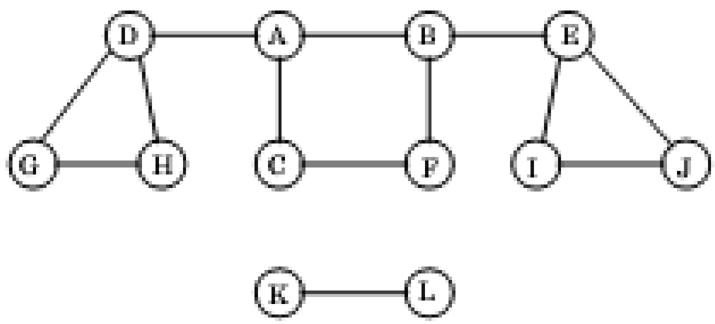
Percorrendo o grafo - DFS

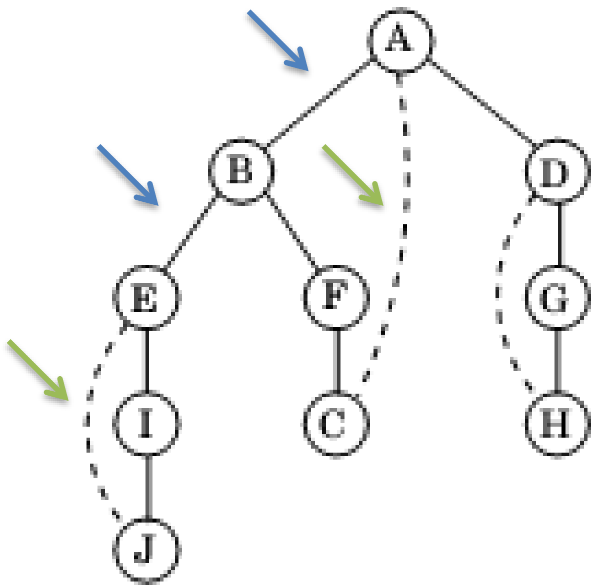




A resposta é não! O procedimento explorar visita somente a porção alcançável do grafo partindo de um ponto de partida. Para completar a busca, vamos ver o algoritmo completo de busca em profundidade.

Percorrendo o grafo - **DFS**





Antes de explorarmos o algoritmo de DFS, repare que:

- As arestas contínuas do grafo formam uma árvore (que é um grafo conexo que não tem 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 (formariam ciclos).

Percorrendo o grafo - **DFS**

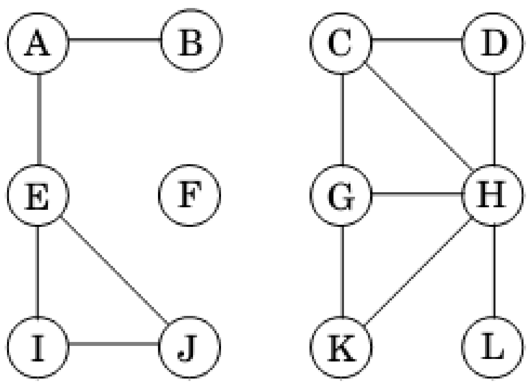
Para examinarmos o restante do grafo, é preciso recomeçar o procedimento em outro lugar, ou seja, em algum vértice que ainda não tenha sido visitado.

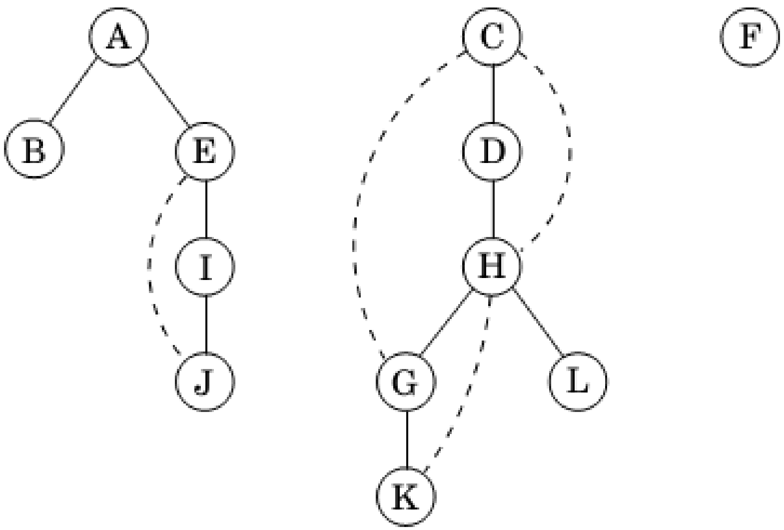
O algoritmo de Busca em Profundidade, ou DFS (*Depth-first Search*), pode fazer isso algumas vezes, até que o grafo esteja completamente explorado.

```
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 - **DFS**

Exemplo:





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

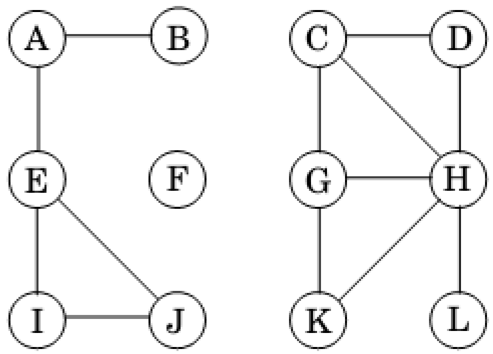
O algoritmo do DFS chama a função de exploração por três vezes, sobre A, C e F, resultando em três árvores, cada uma enraizada em seu ponto de partida. Juntas formam uma floresta.

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. Uma 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|).**
- Por isso, o **tempo total de execução do algoritmo** é **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 pelo menos um caminho entre qualquer par de vértices.



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 <u>subgrafo internamente conexo</u> que não possui arestas para os outros subgrafos.

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 acrescentar um comando pré-visita:

```
ccnum[v] = cc
```

Onde `cc` precisa 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 <u>grafo é conexo</u>. Caso contrário, cc conterá o número de componentes conexas e o vetor ccnum conterá a componente conexa a que pertence cada um dos vértices.

```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.

Exercícios – URI Academic

2959 - ParaTudo!

2854 – Árvore Genealógica