

## Paradigmas e projetos de algoritmos

- Algoritmos recursivos
- Força bruta
- Gulosos

**Algoritmos recursivos:** Algoritmos recursivos são algoritmos que chamam a si mesmo como por exemplo uma árvore de busca.

Quando não utilizar recursão.

Em resumo, evite algoritmos recursivos quando o desempenho, uso de memória ou legibilidade forem preocupações significativas, ou quando a recursão não oferecer benefícios claros em relação a abordagens iterativas.

pythonCopy code

```
def contador_infinito():  
    print("Contando...")  
    contador_infinito() # Chama a si mesma infinitamente  
  
contador_infinito()
```

Neste exemplo, a função `contador_infinito()` chama a si mesma repetidamente sem qualquer condição de parada. Isso levará a um estouro de pilha, já que cada chamada adiciona um novo contexto de função à pilha de chamadas do programa. Eventualmente, o programa ficará sem espaço na pilha e lançará um erro de estouro de pilha.

É importante sempre ter uma condição de parada em funções recursivas para evitar esse tipo de problema. Sem uma condição de parada adequada, a recursão continuará indefinidamente, consumindo recursos até que o programa falhe.

Faz-se necessário uma *condição de parada*

- Alguma condição que, se satisfeita, finaliza as chamadas recorrentes

Um procedimento recursivo  $P$  deve estar sujeito a uma condição de parada  $B$

- Deve-se **garantir** que  $B$  não seja satisfeita em algum momento
  - $P! = NULL$

Exemplo de algoritmo não eficiente em recursão é o **cálculo de fibonacci**

**cálculo de fibonacci:** sequência matemática onde número da série é somado pelos dois numero anteriores.

**Algoritmos de força bruta:** algoritmos de força bruta (tentativa e erro) tende a encontrar todas as soluções possíveis, ou seja suponha que tem um problema específico, o algoritmo de força bruta vai analisar todas as soluções possíveis e testar todas essas soluções para ter a solução correta final

```
#include <stdio.h>

int encontrar_maximo(int array[], int tamanho) {
    int maximo = array[0]; // Supõe que o primeiro elemento é o máximo
    for (int i = 1; i < tamanho; i++) {
        if (array[i] > maximo) {
            maximo = array[i]; // Atualiza o máximo se encontrar um número maior
        }
    }
    return maximo;
}

int main() {
    int numeros[] = {7, 12, 5, 9, 18, 3};
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);

    int maximo = encontrar_maximo(numeros, tamanho);
    printf("O máximo é: %d\n", maximo);
}
```

Nesse exemplo de algoritmo de força bruta podemos observar que o resultado final vai ser o maior número do array, mas pra isso ele vai "ler" todas as soluções possíveis ou seja testar (o teste nesse caso é a comparação) e entregar o resultado correto no final.

**Pergunta da prova 2 de 2023:** algoritmos de força bruta são úteis para resolver problemas da classe P, apresenta uma pequena discussão sobre isso.

Sabemos que algoritmo de força bruta tem o objetivo de analisar todas as soluções possíveis e entregar o resultado final, algoritmos da classe P são problemas que podem ser resolvidos de forma polinomial em máquinas determinísticas, algoritmos de força bruta pode ser usado para resolver problemas na classe P, como por exemplo encontrar o maior número de um array, então sim, existem algoritmos que podem ser utilizados e são eficientes em algoritmos da classe P embora não seja a forma mais eficiente de ser utilizada, e principalmente se o tamanho da entrada algoritmo for grande.


**pergunta bônus:** algoritmos de força bruta são úteis para resolver problemas da classe NP

Sabemos que problemas da classe NP está na classe de problemas exponencial ou seja tempo de execução de forma não polinomial em comparação ao tamanho da entrada, algoritmos de força bruta são utilizados para testar todos os problemas possíveis para achar a solução, com isso percebemos que utilizar força bruta para analisar problemas da classe NP pode ser utilizado mas vai levar muito tempo para chegar em uma solução.

### Algoritmos gulosos:

Imagine que você está em uma loja de doces com a quantia limitada de dinheiro e quer comprar a maior quantidade possível de doces. Você pode usar uma abordagem gulosa para alcançar seu objetivo. Aqui está como funciona:

1. **Escolha imediata:** Em cada passo, você escolhe o doce mais barato disponível. Isso é uma escolha "gulosa" porque você não está pensando no preço dos doces futuros, apenas no que é mais barato agora.
2. **Continuar escolhendo:** Você continua escolhendo os doces mais baratos até que não tenha mais dinheiro suficiente para comprar outro doce.
3. **Fim:** Quando não puder mais comprar doces, você termina e conta quantos doces conseguiu comprar.

Esse é um exemplo básico de um algoritmo guloso. Em cada etapa, você faz a escolha que parece ser a melhor naquele momento, esperando  que essa escolha leve a uma solução globalmente ótima.

O algoritmo guloso só faz a melhor escolha para o problema atual(problema imediato), o algoritmo não se importa com o problema global, ele faz escolhas ótimas a fim de chegar em uma solução ótima

Os algoritmos gulosos são como "pensamento de curto prazo". Eles não olham para o quadro completo do problema, apenas para a situação imediata. Às vezes, essa abordagem funciona e leva a uma solução ótima, mas em outros casos pode levar a uma solução subótima.

Os algoritmos gulosos são frequentemente utilizados em problemas de otimização, onde você está tentando maximizar ou minimizar algo, como no exemplo dos doces. No entanto, é importante lembrar que nem todos os problemas podem ser resolvidos com algoritmos gulosos, e é necessário cuidado para garantir que essa abordagem leve a uma solução correta e eficiente.

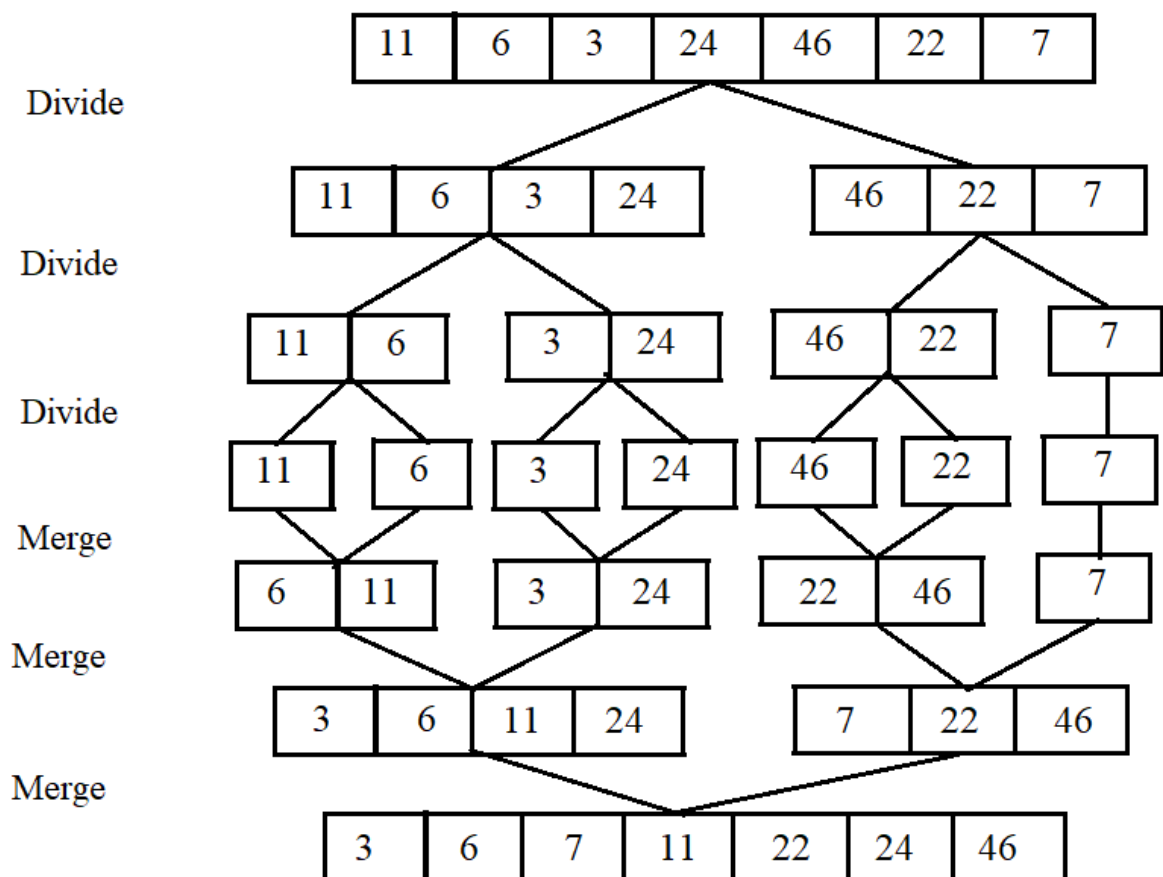
**Pergunta da prova 2 de 2023:** Quais são os pontos fracos e fortes de algoritmos gulosos, apresente uma pequena discussão sobre o tema

algoritmos gulosos tem o objetivo de fazer a melhor escolha de forma imediata, ou seja ele no algoritmo ele tem que fazer escolhas, ele analisa a melhor escolha no momento e aceita sem analisar os problemas futuros, o ponto forte é que racionalmente isso leva a uma solução ótima ou sub ótima e que também como são escolhas imediatas isso leva a um tempo de execução mais rapido do problema principal e são algoritmos eficientes em problemas de otimização, o ponto fraco é ele não garante que essas escolhas ótimas imediatas não vai atrapalhar escolhas futuras e também não consideram escolhas passadas feita no algoritmo e dependem do tipo de estrutura utilizado no problema para ser utilizado o método guloso.

**problemas a ser analisados :** problema da mochila binária, problema do caixeiro viajante problema da coloração dos grafos

- Divisão e conquista
- Programação dinâmica

**Algoritmos de divisão e conquista:** como o próprio nome já diz é o ato de você dividir e conquistar, esse tipo de algoritmo tem como o objetivo de você pegar um problema grande no momento e dividir em pequenas partes sendo que o resultado obtido da parte principal que foi dividida tem que ter tamanhos parecidos e o resultado deve ser simples.



vantagens e desvantagens de utilizar divisão e conquista:

**vantagens:** problemas muito grandes ao ser divididos tornam-se exponencialmente mais simples, ou seja, pega problemas considerados difíceis de forma que fique mais fácil de chegar na solução.

são fáceis de entender ou seja, de implementar, já que é o fato de dividir o problema para chegar na solução final.

**desvantagens:** se você vai dividir um problema em partes, isso significa que você vai dividir várias vezes, ocorrendo uma recursão, isso utilizando mais memória do computador para fazer essas recursões.

Nem todos os problemas podem ser resolvidos utilizando divisão e conquista, muitas das vezes vai ser necessário pegar um problema e adaptar para utilizar a divisão ou seja, transformando em outro problema para ser resolvido, isso gastando o tempo do que utilizando outra forma eficiente para resolver.

**Programação dinâmica:** programação dinâmica aparenta ser parecido com divisão e conquista mas tem detalhes específicos que se destacam em diferença a algoritmos de divisão e conquista.

Na programação dinâmica, também quebramos o problema em subproblemas menores, mas a diferença chave é que, ao invés de resolver cada subproblema independentemente, como na divisão e conquista, nós memorizamos as soluções dos subproblemas já resolvidos. Isso significa que se um subproblema já foi resolvido, ao invés de recalculá-lo, nós apenas utilizamos a solução previamente computada. Isso leva a uma economia de tempo, especialmente quando há sobreposição de subproblemas. Um exemplo clássico de programação dinâmica é o algoritmo para encontrar o caminho mais curto em um grafo ponderado, conhecido como algoritmo de Dijkstra.

Resumindo, na divisão e conquista, resolvemos cada subproblema independentemente, enquanto na programação dinâmica, aproveitamos soluções de subproblemas já resolvidos para economizar tempo e evitar cálculos desnecessários.

A definição da estrutura desta tabela é de extrema importância para a eficiência do algoritmo

- Quais serão as dimensões da tabela?
- O que significa cada dimensão da tabela
- Como determinar os índices de entrada da tabela?
- O que será salvo em cada entrada da tabela?
  - Valor da solução?
  - Componentes da solução?

- **Conceitos de grafos e estruturas de dados para representação de grafos**

▶ Estrutura de Dados - Aula 23 - Grafos - Conceitos básicos

- **Algoritmos de busca em grafos - profundidade e largura**

▶ Estrutura de Dados - Aula 27 - Grafos - Busca em largura

▶ Estrutura de Dados - Aula 26 - Grafos - Busca em profundidade

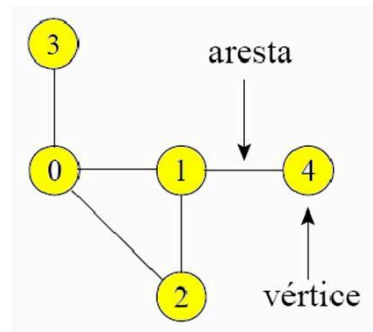
## O que é um grafo:

Grafo é uma estrutura matemática de conjuntos chamados de vértices ou conhecidos como no

A estrutura de um grafo consiste em **vértices e arestas(arcos)** que são conectados entre cada vértice determinado

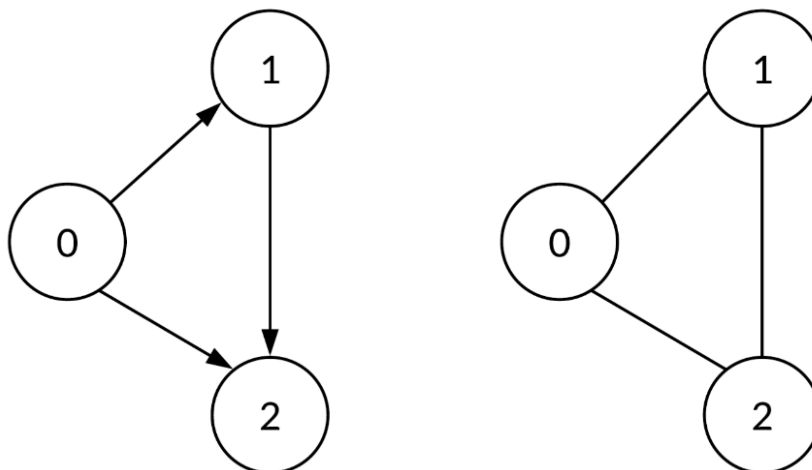
## Siga a representação abaixo de um grafo:

- Grafo: Conjunto de Vértices e Arestas
- Vértice: Objeto simples que pode possuir um nome e outros atributos
- Aresta: Conexão entre dois vértices
- Notação:  $G = (V, A)$ 
  - V : Conjunto de Vértices
  - A: Conjunto de Arestas



Podemos observar que cada bolinha amarela é representada como **vértice**, e as arestas são representadas pelas linhas conectadas entre os vértices

## Os grafos podem ser direcionados e não direcionados



## Qual a diferença entre grafo direcionado e grafo não direcionado

No primeiro caso, representado pelo grafo direcionado, imagine que cada vértice representa um usuário em uma rede social, como o Instagram. Se houver uma seta direcionada do usuário 0 para o usuário 1, isso significa que o usuário 0 está seguindo o usuário 1. No entanto, não há nenhuma indicação de que o usuário 1 está seguindo o usuário 0. Ou seja, a relação é unidirecional, como quando você segue alguém, mas essa pessoa não necessariamente te segue de volta.

No segundo caso, representado pelo grafo não direcionado, cada vértice ainda representa um usuário na mesma rede social. No entanto, agora não há direção nas arestas. Isso significa que se houver uma aresta conectando o usuário 1 ao usuário 2, então automaticamente existe uma aresta conectando o usuário 2 ao usuário 1. Nesse caso, todos estão seguindo uns aos outros reciprocamente, o que é mais semelhante a uma relação de amizade mútua ou "seguir de volta" em uma rede social.

### Self-loops:

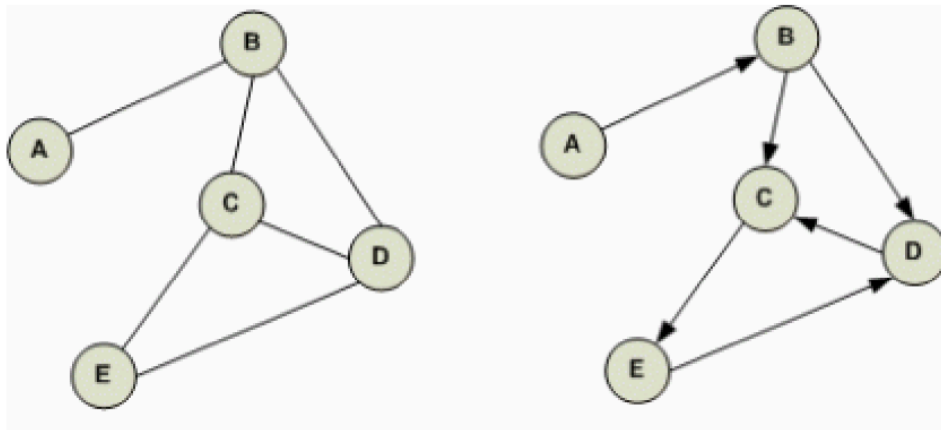
no grafo direcionado o usuário pode seguir ele mesmo

no grafo não direcionado isso já não é possível

### Grau de vértices:

referem ao número de arestas que incidem sobre cada vértice.

**Considere os grafos abaixo:**



Primeiro caso é um grafo **não direcionado**

então podemos representar o grau de cada vértice como:

A = grau 1 ( só tem uma aresta conectando a A que está vindo de B)

B = grau 3 ( tem a aresta de A, C e D conectados a B)

C = grau 3 ( tem as arestas de B, D e E conectados a C)

D = grau 3 ( tem as arestas de C, B e E conectados a D)

E = grau 2 (tem as arestas de C e B conectados a E)

Segundo caso é um grafo **direcionado**

No caso desse grafo devemos representar o **grau de saída** e o **grau de entrada**.

**grau de saída:**representação de quantas arestas então saindo do vértice.

**grau de entrada:** representação de quantas arestas então entrando no vértice.

A: grau de entrada = 0; grau de saída = 1 ( aresta do A a B está saindo de A)

B: grau de entrada = 1; grau de saída = 2

C: grau de entrada = 2; grau de saída = 1

D: grau de entrada = 2; grau de saída = 1

E :grau de entrada = 1; grau de saída = 1

## Caminhos, comprimentos e ciclos

**Caminhos:** é a representação de caminhos de um vértice para outro, considerando o último grafo direcionado podemos observar que de A podemos chegar até E seguindo os vértices abaixo

**Comprimentos:** é a quantidade de arestas de um ponto inicial de vértice para o ponto de destino

Caminho de A até E:

A,B,C e E

Comprimento de A até E = 3( pelo fato de ter tres arestas de A até E)

Caminho de A até D:

A,B e D ou A,B, C, E e D

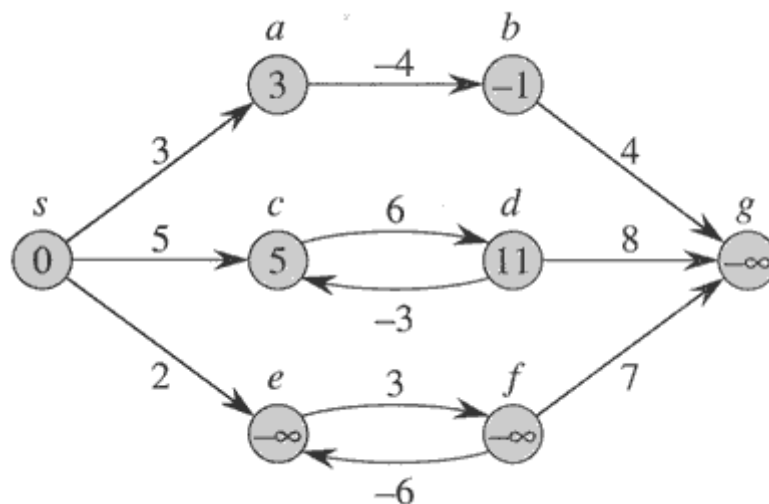
comprimento de A até D = 2

comprimento de A até D passando por C = 4

### Ciclo:

Um ciclo em um grafo ocorre quando existe um caminho formado por uma sequência de arestas que começa e termina no mesmo vértice. Esse ciclo é a representação de um vértice que possui uma aresta que parte dele mesmo e retorna a ele próprio, criando assim uma conexão fechada dentro do grafo.

### Ciclo em grafo direcionado:

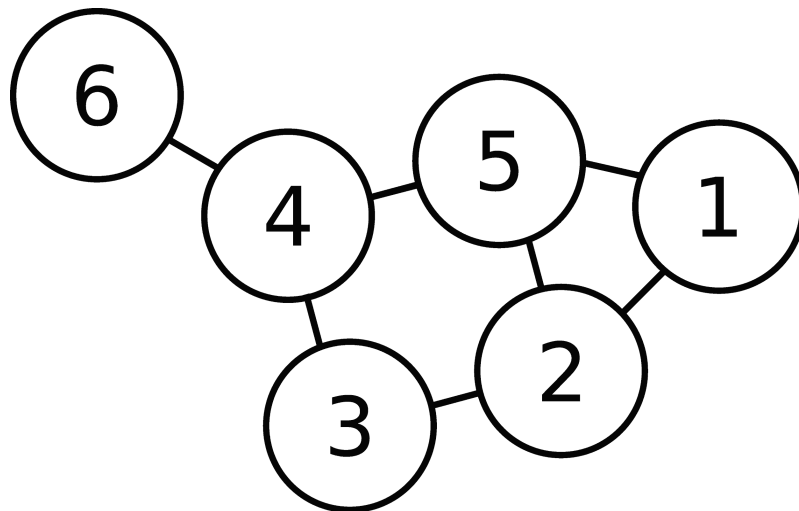


Nesse caso podemos observar que temos alguns ciclos definimos devemos ter pelo menos uma aresta

Começando pelo vértice 0 podemos voltar nele mesmo causando assim um ciclo o tamanho do ciclo é a quantidade de arestas utilizado para formar o determinado ciclo



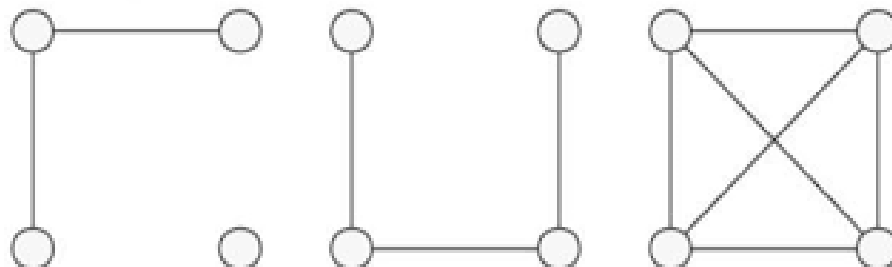
### Ciclo em grafo não direcionado:



Em grafo não direcionado para ter um ciclo deve ter pelo menos três arestas para conectar os vértices. No grafo acima podemos começar pelo vértice 4 e assim retornar nele novamente passando pelo caminho dos vértices 5,2,3 e chegando no 4 criando assim um ciclo.

Outro ciclo que pode ser considerado é começar pelo número 1 e passando pelos vértices 5 e 2 criando um novo ciclo no grafo.

### Grafos conexo e grafos desconexo:



grafo I

grafo II

grafo III

Em **grafos não direcionados** podemos afirmar se ele é conexo ou desconexo

**Grafo I:** podemos observar que é um grafo desconexo pelo fato de ter um vértice fora da estrutura do grafo

**Grafo II:** já é um grafo conexo pelo fato de não ter vértices fora da principal estrutura do grafo

**Grafo III:** isso também acontece causando assim um grafo conexo

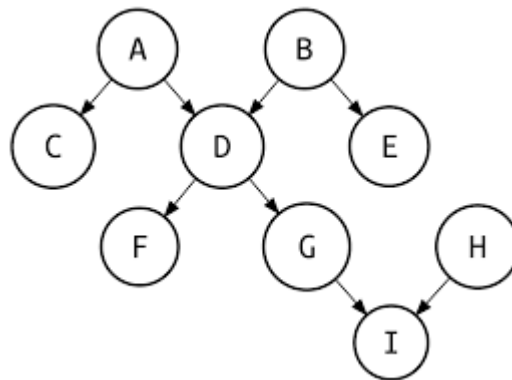
Em grafos direcionados podemos afirmar que ele é conexo ou não mas podemos afirmar outros detalhes

podemos afirmar se ele é:

- **Fortemente conexo**
- **Conexo**
- **Fracamente conexo**
- **Desconexo**

Exemplo de um grafo muito utilizado em estrutura de dados:

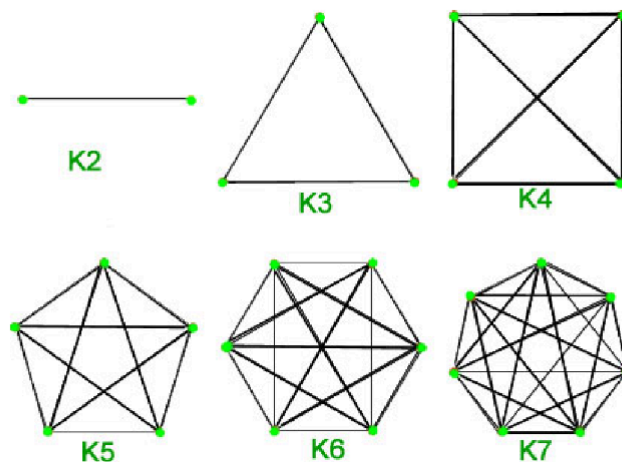
Árvore:



Estrutura de dados muito utilizado em programação muito das vezes sendo utilizado sem saber que é um grafo, podemos afirmar que árvore é um grafo conexo e que não é direcionado e acíclico( ou seja não tem ciclo)

Podemos saber quantas arestas vamos ter sabendo o número de vértices ?

Sim, para isso temos uma fórmula:  $N \cdot (N-1) / 2$  ( sendo N o número de vértices)



Utilizando o segundo grafo(K3) sabemos que tem 3 vértices, com isso podemos fazer  $3 \cdot (3-1) / 2 = 3$  ( que é exatamente o número de arestas do segundo Grafo)

## **Busca em largura e Busca em profundidade:**

**Busca em largura:** começamos a exploração a partir de um vértice inicial e visitamos todos os seus vizinhos antes de seguirmos para os vizinhos dos vizinhos. Esse algoritmo expande uniformemente em todas as direções a partir do vértice inicial, níveis por níveis, utilizando uma fila para armazenar os vértices que precisam ser explorados.

**Busca em profundidade:** exploramos tão longe quanto possível ao longo de um ramo do grafo antes de retrocedermos. Esse algoritmo segue um único caminho até chegar ao final do ramo, e então retrocede para explorar outros ramos, utilizando uma pilha (ou recursão) para armazenar os vértices que precisam ser explorados. A BFS é eficaz na determinação da menor distância entre dois vértices e na descoberta de caminhos mínimos em grafos ponderados não negativos, sendo utilizada em problemas como encontrar o caminho mais curto em um grafo e encontrar todos os vértices alcançáveis a partir de um vértice inicial. Por outro lado, a DFS é eficaz na identificação de componentes conectados em grafos e na busca de soluções em problemas de backtracking. É utilizada em problemas como a identificação de ciclos em grafos, ordenação topológica e busca de soluções em jogos. Em resumo, enquanto a busca em largura prioriza a exploração horizontalmente, expandindo uniformemente em todas as direções, a busca em profundidade prioriza a exploração verticalmente, seguindo um caminho até o final antes de retroceder. Ambos os algoritmos têm suas aplicações específicas e são essenciais para a análise e manipulação de grafos.

## **Aviso: Detalhes sobre a criação do PDF e suas atualizações**

Este documento é um resumo elaborado por um aluno e pode conter erros ou imprecisões. O conteúdo foi criado com base em entendimento pessoal e pode não refletir completamente as informações originais.

Caso identifique qualquer informação incorreta ou imprecisa, por favor, entre em contato com o gerenciador do PDF para que as devidas correções possam ser feitas.

Este PDF está sujeito a atualizações futuras para seguir as normas da ABNT (Associação Brasileira de Normas Técnicas) e para melhorar a explicação de conteúdos.

Obrigado pela compreensão.