Algoritmos e Grafos

1. Representação de Grafos

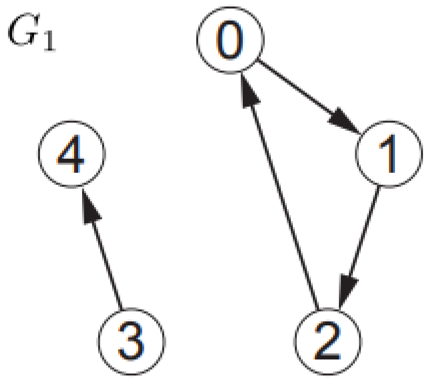
Um grafo $G = (V, E)$ é um objeto matemático composto por um conjunto de vértices (V), também chamados de nós, e por um conjunto de arestas (E), onde cada aresta em E é um subconjunto de dois elementos de vértices V, ou seja, um par ordenado.
Seja u e v vértices de G. Uma aresta é direcionada se seu par de subconjuntos for ordenado, por exemplo, (u, v), e não direcionada se seu par de subconjuntos for não ordenado, por exemplo, $\{u, v\}$ ou, alternativamente, tanto (u, v) quanto (v, u)

Com isso grafos podem ser **direcionados** e **não direcionados**

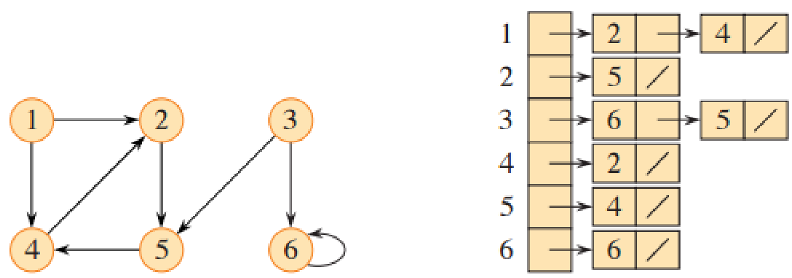
Grafo Direcionado

Uma aresta direcionada $e = (u, v)$ estende-se do vértice u para o vértice v ($u \longrightarrow v$), com e sendo uma aresta de *entrada* de v e uma aresta de *saída* de u.

$$G_1 = (V_1, E_{1)} \quad V_1 = \{0, 1, 2, 3, 4\} \quad E_1 = (0, 1), (1, 2), (2, 0), (3, 4)$$



Outros exemplos de representação:



(a) (b)

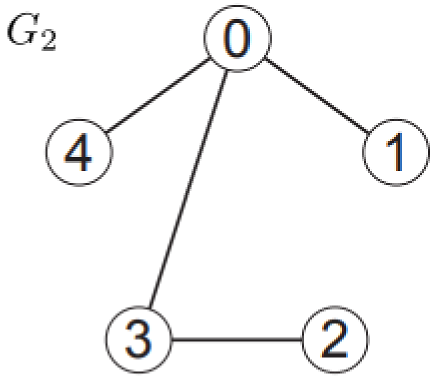
$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

(c)

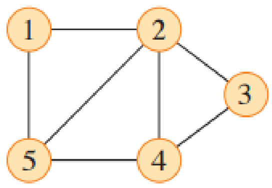
Grafo Não Direcionado

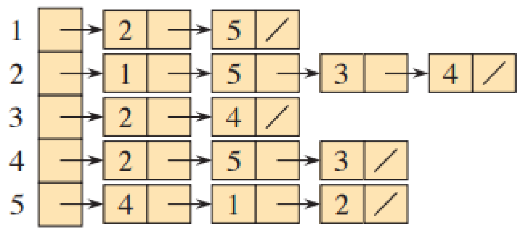
Em um grafo não direcionado, toda aresta é de entrada e de saída.

$$G_2 = (V_2, E_2) \quad V_2 = \{0, 1, 2, 3, 4\} \quad E_2 = \{\{0, 1\}\{0, 3\}, \{0, 4\}, \{2, 3\}\}$$



Outros exemplos:





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

Há outra representação de grafos:

- **Grafo Valorado**, onde cada aresta possui um **peso**
- **Grafo Completo**, onde todos vértices são vizinhos, todos n vértices tem $n - 1$ arestas

Vizinhança

Um vértice u é dito *vizinho* de v se tiver uma aresta E ligando os vértices.
($u \longrightarrow v$)

Caminho

Um caminho de tamanho k do vértice u ao vértice v é uma sequência de vértices $(v_0, v_1, \ldots, v_n)$ onde para cada par ordenado (v_{i-1}, v_i) existe uma aresta, onde $k =$ número de arestas (ou N^o de vértices $- 1$)
Grafo fortemente conectado: Para cada vértice há um caminho para cada outro vértice

Ciclo

Em um grafo direcionado dado um caminho $(v_0, v_1, \ldots, v_k)$, ele forma um ciclo se, somente se, $v_0 = v_k$.
Em um grafo não direcionado, o caminho forma um ciclo, se $v_0 = v_k$, tem pelo menos uma aresta $(\#V > 0)$ que conecta um par ordenado, e todos os vértices e arestas no trajeto (exceto o inicial/final) são **distintos**.

Representação computacional

Há duas formas padrão de se representar um grafo $G = (V, E)$

- listas de adjacências
- matriz de adjacências
  Ambos funcionam para grafos direcionados e não direcionados.
- A **lista de adjacência** é mais compacta para grafos *esparsos* (quando $|E| \ll |V|^2$). Por isso, é geralmente a representação escolhida, e a maioria dos algoritmos do livro assume esse formato.
- A **matriz de adjacência** é preferida para grafos *densos* (quando $|E|$ é próximo de$|V|^2$) ou quando precisamos saber rapidamente se existe uma aresta entre dois vértices.

Listas de Adjacências

Estrutura: um vetor **Adj** de tamanho $|V|$, onde cada posição corresponde a um vértice e contém uma lista de seus vizinhos.

- Se o grafo é direcionados: o tamanho total das listas é $|E|$.
- Se o grafo é não direcionados: o tamanho total é $2|E|$, porque cada aresta (u, v) aparece em ambas as listas.
  Espaço necessário: $\Theta(V + E)$.
- Pode ser adaptado para grafos **ponderados** (valorados), armazenando o peso junto ao vértice vizinho.
  Desvantagem: para verificar se uma aresta (u, v) existe, é preciso percorrer a lista de u.

Matriz de Adjacências

Estrutura: uma matriz $|V| \times |V|$, onde

$$a_{ij} = \begin{cases} 1 & \text{, se } (i, j) \in E \\ 0 & \text{, c.c.} \end{cases}$$

- Ocupa sempre $\Theta(V^2)$ de memória, independentemente do número de arestas.
- A busca de uma aresta é imediata, $O(1)$.
- Para grafos não dirigidos, a matriz é simétrica.

Também pode representar grafos ponderados armazenando o peso em vez de apenas 0 ou 1.

Pode ser otimizada armazenando apenas metade da matriz (parte superior da diagonal

Para grafos pequenos, é mais simples que listas de adjacência e pode usar apenas 1 bit por posição em grafos não ponderados.

Atributos de vértices e arestas

- Algoritmos costumam manter atributos (ex.: cor, distância, pai).
- Esses atributos podem ser armazenados em arrays auxiliares ou junto às estruturas.
- A forma de implementação varia conforme a linguagem de programação e necessidades do algoritmo.

2. Algoritmos básicos

Busca

- Matriz de Adjacências

#Vértice
Verifica se um vértice existe no grafo

```
busca_vertice(grafo G, int id)
    if 1 <= id <= G.n:
        return i    // índice do vértice (ou objeto associado)
    else:
        return NIL
```

#Aresta
Verifica se existe uma aresta entre o vértice `u` e o vértice `v`

```
busca_aresta(grafo G, vertice u, vertice v)
    retornar (A[u][v] == 1)
```

- Lista de Adjacências

#Vértice

```
BUSCA-VERTICE-LISTA(G, id)
    se id ∈ V[G] então
        retornar id
```

```
    senão
        retornar NIL
```

#Aresta

```
busca-aresta(Grafo G, vertice u, vertice v)
    para cada x em Adj[u]:
        se x == v:
            retornar VERDADEIRO
    retornar FALSO
```

Inserir

- Matriz de Adjacências

#Vértice
expandir a matriz para dimensão $(|V|+1) \times (|V|+1)$ e inicializar nova linha/coluna com 0

```
//Vertice
inserir_vertice(grafo G, vertice v)
    expandir matriz A
    para todo i:
        A[v][i] ← 0
        A[i][v] ← 0
```

#Aresta

```
insere_aresta(grafo G, vertice u, vertice v, bool direcionado)
    A[u][v] ← 1
    if not direcionado:
        A[v][u] ← 1
```

- Lista de Adjacências

#Vértice
criar nova lista vazia `Adj[v]`.

```
INSERE-VERTICE(grafo G, vertice v)
    Adj[v] ← lista vazia
```

#Aresta

Grafo dirigido: adiciona v na lista de u.

Grafo não dirigido: adiciona v na lista de u **e** u na lista de v.

```
FUNÇÃO ADICIONAR_ARESTA(Grafo G, Vértice u, Vértice v, é_direcionado)
    // Adiciona v à lista de adjacência de u
    G.Adj[u].adicionar(v)

    SE não é_direcionado ENTÃO
            // Se o grafo não for direcionado, a aresta é recíproca
            G.Adj[v].adicionar(u)
    FIM-SE
FIM-FUNÇÃO
```

Remover

- Matriz de Adjacências
  remover linha e coluna correspondentes a `v`.
  #Vértice

```
REMOVE-VERTICE-MATRIZ(G, v)
    remover linha v e coluna v da matriz A
```

#Aresta

```
REMOVE-ARESTA-MATRIZ(G, u, v, direcionado)
    A[u][v] ← 0
    if not direcionado:
        A[v][u] ← 0
```

- Lista de Adjacências
  remover a lista `Adj[v]` e apagar todas as ocorrências de `v` em outras listas.
  #Vértice

```
REMOVE-VERTICE-LISTA(G, v)
    para cada u em V:
        remover v de Adj[u]
    remover lista Adj[v]
```

#Aresta

```
REMOVE-ARESTA-LISTA(G, u, v, direcionado)
    remover v de Adj[u]
    if not direcionado:
        remover u de Adj[v]
```

3. Busca em Largura (BFS - Breadth-First Search)

- Explora vértices por **camadas**: primeiro os a distância k, depois $k+1$.
- Garante cálculo de **menores distâncias** em grafos não ponderados.
- Constrói conjuntos L_i: vértices a distância i da fonte.
- **Complexidade**: $O(V+E)$.
- **Propriedade**: BFS é correto porque cada aresta adiciona apenas vértices na camada seguinte.

Dado um grafo `G = (V, E)` e um vértice de origem `s`, a busca em largura explora sistematicamente as arestas de `G` para "descobrir" todos os vértices alcançáveis a partir de `s`.

Como funciona: O algoritmo avança em "ondas" a partir da origem `s`. Primeiro, visita todos os vizinhos de `s` (vértices a uma distância de 1 aresta). Em seguida, visita os vizinhos desses vizinhos (vértices a uma distância de 2 arestas), e assim por diante, até que todos os vértices alcançáveis tenham sido visitados.

Caminho mais curto: Uma propriedade crucial da busca em largura é que ela calcula a distância do caminho mais curto (em termos de número de arestas) de `s` para cada vértice alcançável.
BFS também serve de base para algoritmos de **caminho mínimo em grafos não ponderados**

Árvore de busca em largura: Durante a busca, o algoritmo constrói uma "árvore de busca em largura" com raiz em `s`, que contém todos os vértices alcançáveis. Para qualquer vértice `v` alcançável a partir de `s`, o caminho simples na árvore de `s` para `v` corresponde a um caminho mais curto no grafo original.

Estrutura de dados: Para gerenciar a fronteira de vértices descobertos, o algoritmo utiliza uma fila (queue) no estilo FIFO (primeiro a entrar, primeiro a sair).

Coloração de Vértices: Para acompanhar o progresso, cada vértice é colorido de branco, cinza ou preto.

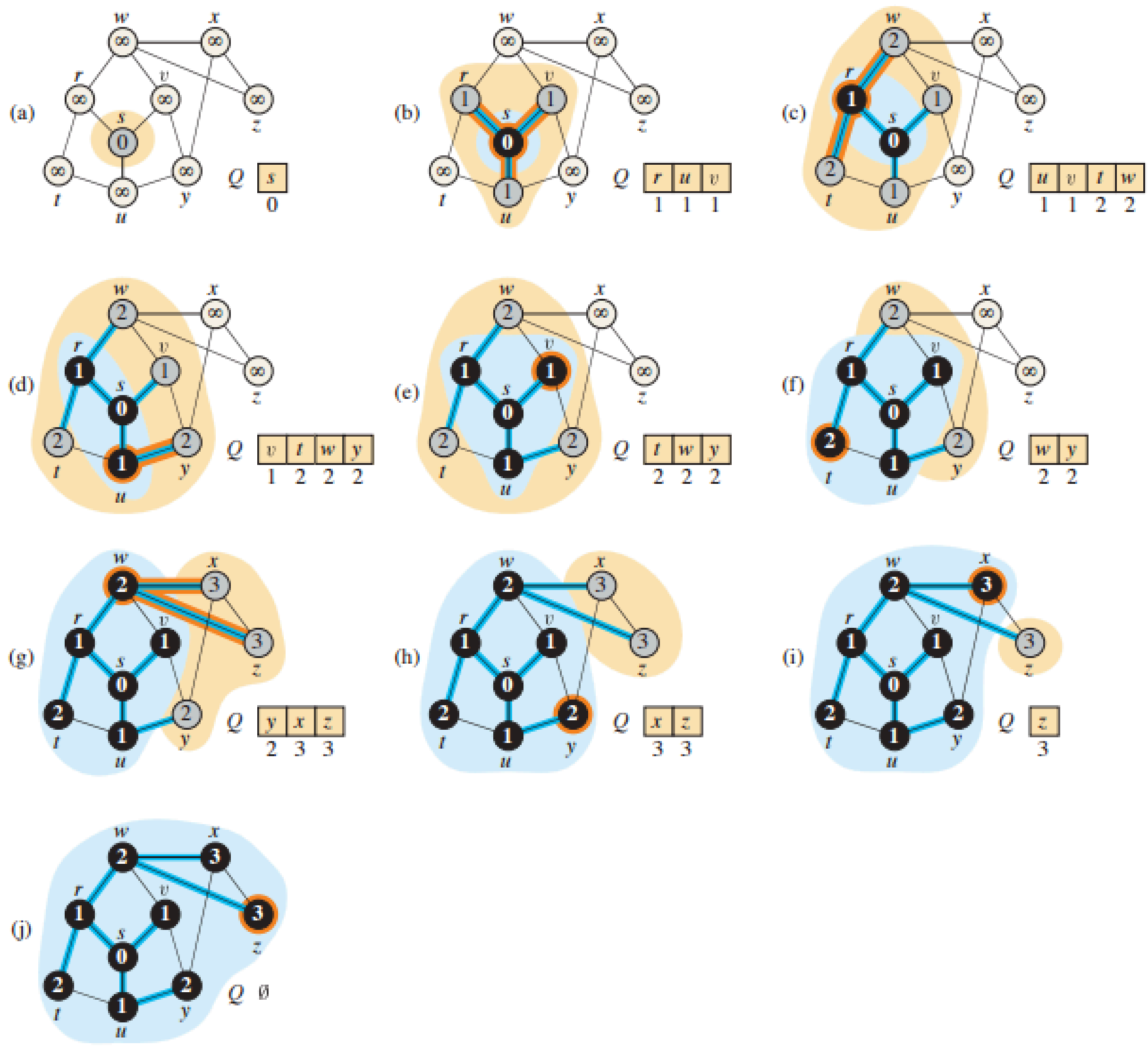
- **Branco**: Vértice ainda não descoberto.
- **Cinza**: Vértice descoberto, mas seus vizinhos ainda não foram todos explorados. Os vértices cinzas formam a fronteira na fila.
- **Preto**: Vértice "finalizado", ou seja, todos os seus vizinhos já foram descobertos

Algoritmo

```
BFS(G, s)   // G = (V, E), s = vértice de origem
    para cada v ∈ V[G] faça
        cor[v] ← branco         // não visitado
        d[v] ← ∞                // distância da origem
        p[v] ← NIL              // pai/predecessor
    cor[s] ← cinza
    d[s] ← 0
    p[s] ← NIL

    Q ← fila vazia
    ENQUEUE(Q, s)

    enquanto Q não estiver vazia faça
        u ← DEQUEUE(Q)
        para cada v ∈ Adj[u] faça
            se cor[v] = branco então
                cor[v] ← cinza
                d[v] ← d[u] + 1
                p[v] ← u
                ENQUEUE(Q, v)
        cor[u] ← preto
```



4. Subgrafos

Árvores

- Grafo **não direcionado**, **conexo** (existe caminho entre quaisquer dois vértices) e **sem ciclos**.
- Propriedades principais:
    a. Se tem n vértices, possui exatamente $n-1$ arestas.
    b. Há um **único caminho simples** entre quaisquer dois vértices.

Floresta

- Conjunto de **árvores disjuntas** (um grafo acíclico, mas não necessariamente conexo).
- Cada componente conexo de uma floresta é uma árvore.

Subgrafo

- Grafo formado a partir de outro grafo $G = (V, E)$, escolhendo um subconjunto de vértices $V' \subseteq V$ e um subconjunto de arestas $E' \subseteq E$ que conectam apenas vértices em V'.
- Pode ser:
  a. **Induzido**: contém todas as arestas entre os vértices de V'.
  b. **Não induzido**: contém apenas algumas dessas arestas.

Subgrafo predecessor

- Subgrafo formado a partir das relações de **predecessores** em uma busca (DFS ou BFS).
- Quando executamos DFS ou BFS, cada vértice visitado guarda um **pai** (predecessor).
- Conectando cada vértice ao seu predecessor, obtemos um **subgrafo em forma de árvore ou floresta**, chamado **árvore de busca** ou **subgrafo predecessor**.

Durante a busca, para cada vértice `v` que é descoberto a partir de um vértice `u`, `u` se torna o predecessor de `v`. O subgrafo predecessor é o grafo G' = (V', E') formado por todos os vértices alcançados durante a busca e pelas arestas
`(u, v)` onde `u` é o predecessor de `v`.
Este subgrafo forma uma árvore (ou uma floresta de árvores) que representa os caminhos descobertos a partir do vértice de origem.
Por exemplo, na busca por caminhos mais curtos, o subgrafo predecessor forma uma **árvore de caminhos mais curtos**

5. Busca em Profundidade (DFS - Depth-First Search)

Estratégia: A DFS explora o mais "profundamente" possível ao longo de cada ramo antes de retroceder (backtracking). Ela vai o mais longe que pode por um caminho, e só volta quando não há mais vértices brancos (não descobertos) para explorar a partir do vértice atual.

A execução constrói uma árvore de profundidade, podendo gerar uma floresta caso o grafo seja desconexo

Estrutura Recursiva: A DFS é naturalmente recursiva. A busca a partir de um vértice `u` é suspensa quando um novo vértice `v` é descoberto, iniciando uma nova busca a partir de `v`. A busca a partir de `u` só recomeça depois que todos os vértices alcançáveis a partir de `v` forem explorados.

Cores e Timestamps:
Para evitar loops, o DFS mantém um atributo de "cor" para cada vértice.

- **Branco**: não visitado
- **Cinza**: descoberto, mas ainda em processamento
- **Preto**: totalmente explorado

Além disso, ela atribui dois "timestamps" a cada vértice:

- **Timestamp de Descoberta (d)**: Registra quando um vértice se torna cinza.
- **Timestamp de Finalização (f)**: Registra quando um vértice se torna preto.

Em vez de apenas marcar vértices visitados, o algoritmo também mantém o controle da árvore gerada pela travessia em profundidade. Ele faz isso marcando o "pai" de cada vértice visitado, ou seja, o vértice que o DFS visitou imediatamente antes de visitar o filho.
O DFS aumentado também marca dois carimbos de tempo auto-incrementais, d e f, para indicar quando um nó foi descoberto pela primeira vez e quando foi finalizado.

DFS serve de base para **ordenamento topológico** e **componentes fortemente conectados**.

Complexidade de Tempo: Como cada aresta é explorada **no máximo uma vez**, o tempo de execução é **Θ(V + E)**.

Algoritmo

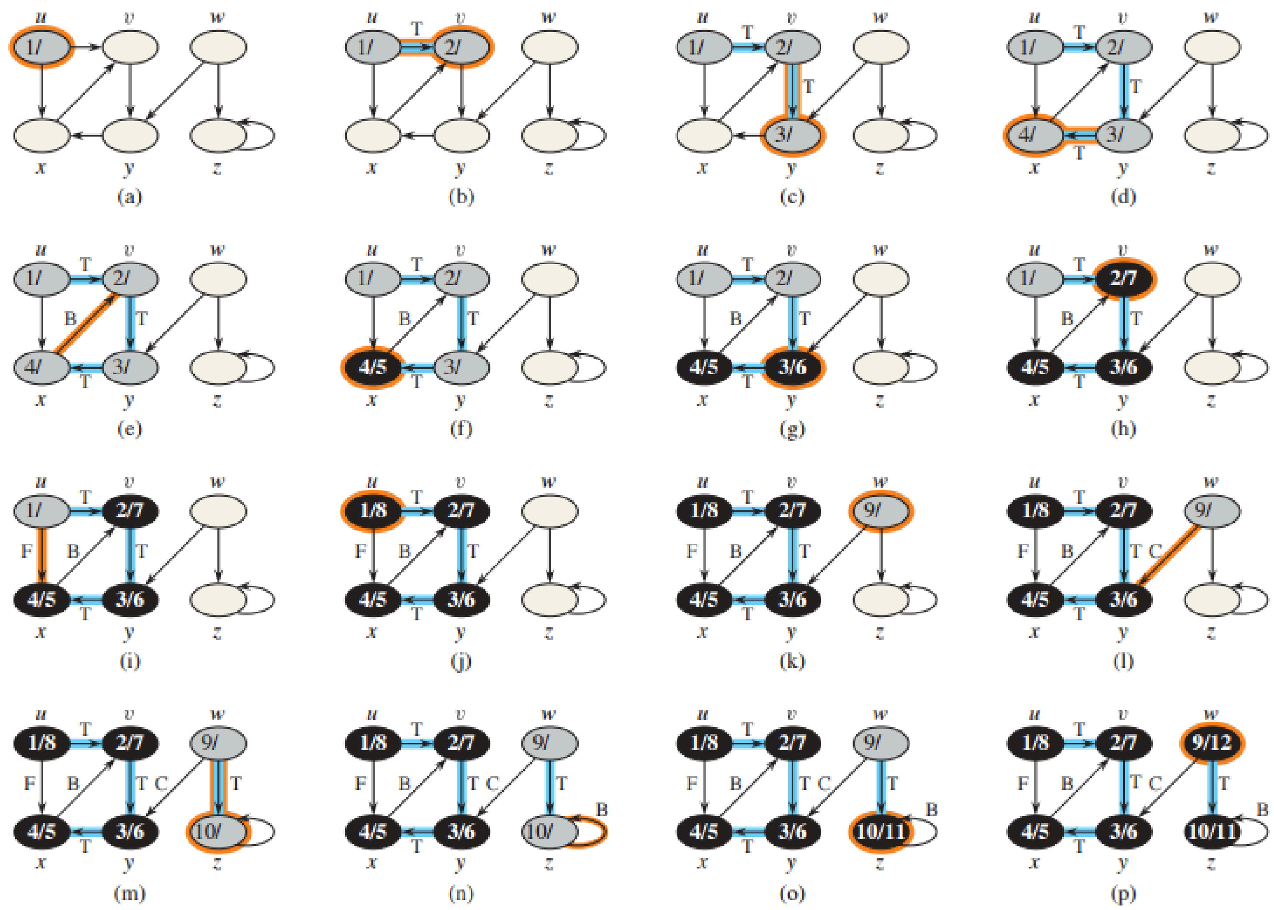
Procedimento principal:

```
DFS(G)
    para cada vértice u ∈ V[G] faça
        cor[u] ← branco
        π[u] ← NIL
    tempo ← 0
    para cada vértice u ∈ V[G] faça
        se cor[u] = branco então
            DFS-Visita(G, u)
```

Procedimento auxiliar:

```
DFS-VISIT(G, u)
    cor[u] ← cinza
    tempo ← tempo + 1
    d[u] ← tempo
    para cada v ∈ Adj[u] faça
        se cor[v] = branco então
            π[v] ← u
            DFS-Visita(G, v)
    cor[u] ← preto
```

```
tempo ← tempo + 1
f[u] ← tempo
```



6. Aplicações da Busca em Profundidade (DFS)

Classificação de Arestas:

Com base nesses timestamps e cores, a DFS classifica as arestas em quatro tipos, o que revela informações importantes sobre a estrutura do grafo:

- **Arestas de Árvore**: Arestas que levam a um vértice branco não descoberto. Elas formam a "floresta de busca em profundidade".
- **Arestas de Retorno (Back edges)**: Arestas que conectam um vértice a um de seus ancestrais na árvore de busca (indicam a presença de ciclos).
- **Arestas de Avanço (Forward edges)**: Arestas que conectam um vértice a um de seus descendentes (que não seja um filho direto).
- **Arestas de Cruzamento (Cross edges)**: Todas as outras arestas.

Ordenação Topológica

Aplicável apenas em **grafos direcionados acíclicos** (DAGs).

Um ordenamento topológico é uma listagem linear dos vértices tal que, para toda aresta (u, v), (u) aparece antes de (v).

Usos:

- Planejamento de tarefas com dependências.
- Compilação (ordem correta de módulos).
- Ordenação de precedências em projetos.

Algoritmo (via DFS):

1. Executa-se DFS no grafo.
2. Cada vértice recebe um tempo de término $(f[v])$.
3. A ordenação topológica é a lista dos vértices em ordem decrescente de $(f[v])$.

Complexidade: $(O(V + E))$.

Observação: só é possível se o grafo **não tiver ciclos**.

Componentes Conectados

Para **grafos não direcionados** um componente conectado de um grafo G é um conjunto máximo de vértices no qual qualquer par de vértices está ligado por algum caminho.

Componentes Fortemente Conectados (CFC ou SCC)

Em grafo direcionado:
É um subconjunto máximo de vértices $(S \subseteq V)$ tal que, para quaisquer $(u, v \in S)$, existe um caminho de `u` até `v` e de `v` até `u` (Ambas as direções).

Propriedades:

- Todo grafo direcionado pode ser decomposto em SCCs disjuntos.
- A relação de pertencimento a um **SCC é uma relação de equivalência** (reflexiva, simétrica, transitiva).
- O conjunto dos SCCs forma um **grafo condensado, que é sempre um DAG**.

Algoritmo de Kosaraju (baseado em DFS):

1. Executa DFS no grafo original e guarda os tempos de término.
2. Inverter todas as arestas (grafo transposto).

3. Executar DFS novamente, explorando vértices em ordem decrescente de tempos de término da 1ª fase.
4. Cada árvore da segunda DFS corresponde a um SCC.

Complexidade: ($O(V + E)$).

7. Problemas de Otimização

Um problema de otimização é aquele em que, entre muitas soluções possíveis (S), o objetivo é encontrar uma solução "ótima"(S^*), seja minimizando ou maximizando um valor específico

Problema de otimização combinatória

- **Conjunto base (ground set)** $E = \{1, 2, \ldots, n\}$.
- **Conjunto de soluções viáveis**: $F \subseteq 2^E$.
- **Função objetivo** $f : 2^E \to \mathbb{R}$.
  Para **minimização**, busca-se $S^* \in F$ tal que:

  $$f(S^*) \leq f(S), \quad \forall S \in F$$

  Exemplos: caminho mínimo, árvore geradora mínima, problema da mochila, TSP.
- Há versões do problema: **otimização, avaliação e decisão**, sendo esta última fundamental na análise de complexidade

Método Guloso (Greedy Method)

O método guloso é uma estratégia para resolver problemas de otimização que constrói uma solução passo a passo. A cada passo, o algoritmo faz uma escolha que parece ser a melhor naquele momento — uma escolha localmente ótima — na esperança de que ela leve a uma solução globalmente ótima

As principais características de um algoritmo guloso são:

- **Propriedade da escolha gulosa**: Uma solução globalmente ótima pode ser alcançada fazendo escolhas localmente ótimas. Diferente da programação dinâmica, uma escolha gulosa pode ser feita sem considerar as soluções de subproblemas futuros.
- **Subestrutura ótima**: Um problema exibe subestrutura ótima se uma solução ótima para o problema contém soluções ótimas para os subproblemas

Ideia: Construir a solução de forma incremental, escolhendo localmente a melhor opção em cada passo.

Estrutura:

1. Construir a solução **passo a passo**.

2. Em cada passo, escolher um **elemento viável** que pareça o melhor segundo uma **função de escolha gulosa**.
3. Repetir até formar uma solução completa.

- Vantagens: simples e rápido.
- Limitação: não garante solução ótima em todos os problemas.

Variantes:

- **Guloso simples** (usa custos fixos, ex.: Kruskal).
- **Guloso adaptativo** (função de escolha depende das escolhas anteriores).
- **Semi-guloso** (introduz aleatoriedade para escapar de soluções ruins).

Algoritmo

```
ALGORITMO-GULOSO(E, F, f)
    // E: conjunto base
    // F: conjunto de soluções viáveis
    // f: função objetivo
    // S: solução viável

    S ← ∅                       // solução parcial
    f(S) ← 0
    F ← {i ∈ E : S ∪ {i} é viável}
    enquanto F ≠ ∅ faça
        i* ← argmin{ci : i ∈ F};
        S ← S ∪ {i*};
        f(S) ← f(S) + ci*;
        F ← {i ∈ F \ {i*} : S ∪ {i} não é inviável};
    retorne S, f(S);
```

Problema da árvore geradora mínima

Este é um exemplo clássico de um problema que pode ser resolvido eficientemente com um método guloso

Definição:
Dado um grafo conectado $G = (V, E)$, não direcionado e com pesos $w(e)$ nas arestas, o objetivo é encontrar um **subconjunto** de arestas que **conecte todos os vértices** (uma árvore geradora $T \subseteq E$) e cujo **peso total** (a soma dos pesos das arestas) seja o **menor possível**.

Propriedades:

- A MST sempre tem $|V| - 1$ arestas.
- $w(u, v) =$ distância
- Substituir uma aresta cara por uma mais barata que mantenha a conectividade sempre melhora a solução.

Complexidade: $O(E \log V)$

Árvore Geradora

Uma **árvore geradora** (spanning tree) de um grafo `G = (V, E)` que é conectado e não direcionado, é um subgrafo $T = (V, E')$ que é uma árvore e conecta todos os vértices em `V`.

Em outras palavras, uma árvore geradora deve satisfazer duas condições principais:

1. **Conectividade**: Ela deve incluir todos os vértices do grafo original G.
2. **Acíclica**: Não pode conter ciclos e ser conexa, ou seja, **ser uma árvore**.
3. Possuir exatamente $|V| - 1$ arestas.

Intuição: é uma forma de “conectar todos os vértices” com o **mínimo de arestas possível**, sem deixar o grafo desconexo e sem redundâncias (ciclos).

8. Árvore Geradora Mínima (MST)

Definição: dado um grafo não direcionado e conexo $G = (V, E)$ com pesos $w(e)$, uma **árvore geradora mínima** é um subconjunto de arestas $T \subseteq E$ que:

- conecta todos os vértices,
- forma uma árvore ($|V| - 1$ arestas, sem ciclos),
- tem peso total mínimo $w(T) = \sum_{e \in T} w(e)$. (Minimiza o custo)

Aplicações: projeto de redes (energia, computadores, estradas), compressão de dados, agrupamento em aprendizado de máquina.

- É um problema clássico resolvido de forma eficiente por algoritmos gulosos (ex.: **Prim** e **Kruskal**).

Algoritmo Genérico

1. Inicialize um conjunto de arestas `A` como vazio.
2. Enquanto `A` não formar uma árvore geradora, encontre e adicione uma aresta "segura" para `A`. Uma aresta é segura se, ao ser adicionada a `A`, o novo conjunto `A'` ainda for um subconjunto de alguma MST.
3. Retorne `A`.

```
GENERIC-MST(G)
    A ← ∅
    enquanto A não forma uma árvore geradora faça
        encontre uma aresta (u, v) que seja segura para A
        A ← A ∪ {(u, v)}
    retornar A
```

Aresta Segura

A parte crucial do algoritmo genérico é encontrar uma aresta segura. Para isso, o conceito de **corte** é introduzido

Cortes e Arestas Leves

- Um **corte** de G é uma partição dos vértices em dois conjuntos disjuntos $(S, V - S)$.
- Uma aresta **cruza o corte** se conecta um vértice de S a um de $V - S$.
- Diz-se que o corte **respeita** A se nenhuma aresta de A cruza o corte.
- A **aresta leve** de um corte é a de **menor peso** entre as que cruzam o corte.

Teorema da Aresta Segura

> Se (u, v) é uma aresta leve que cruza um corte que respeita A, então (u, v) é **segura** para A

Intuição da prova:

- Seja T uma MST que contém A.
- Adicionar (u, v) cria um ciclo.
- Esse ciclo tem outra aresta (x, y) que também cruza o corte.
- Como (u, v) é a mais leve, $w(u, v) \leq w(x, y)$.
- Substituindo (x, y) por (u, v), obtemos outra MST que contém $A \cup \{(u, v)\}$.

Por consequência o algoritmo genérico **sempre escolhe arestas seguras** → garante correção

Tanto **Prim** quanto **Kruskal** são instâncias desse algoritmo:

- **Prim:** mantém uma árvore única; escolhe a aresta leve que conecta a árvore a um novo vértice.
- **Kruskal:** mantém uma floresta; escolhe a menor aresta que conecta componentes distintos.

Algoritmo Kruskal

Ideia: construir a MST adicionando as arestas em ordem de peso crescente (ir adicionado arestas de menor peso), sem formar ciclos

Estratégia: O algoritmo examina as arestas em ordem crescente de peso. A cada passo, ele adiciona a próxima aresta de menor peso que não forma um ciclo com as arestas já selecionadas.

Passos:

1. Ordenar todas as arestas de E por peso crescente.
2. Inicializar uma floresta com cada vértice isolado.
3. Para cada aresta (u, v), em ordem:
   a. Se u e v estão em componentes diferentes → adicionar a aresta.
   b. Caso contrário, descartar (pois formaria ciclo).
4. Repetir até obter $|V| - 1$ arestas.

Estrutura auxiliar: Union-Find (Disjoint Set Union – DSU) para verificar componentes.

Complexidade:

- Ordenação: $O(E \log E)$.
- Union-Find quase constante por operação ($\alpha(V)$).
- Total: $O(E \log V)$.

Vantagem: simples, eficiente em grafos **esparsos**.

```
KRUSKAL-MST(G, w)
    A ← Ø
    para cada v ∈ V[G] faça
        MAKE-SET(v)
    ordenar as arestas E[G] em ordem crescente de peso
    para cada aresta (u, v) ∈ E[G], na ordem crescente faça
        se FIND-SET(u) ≠ FIND-SET(v) então
            A ← A ∪ {(u, v)}
            UNION(u, v)
    retornar A
```

Algoritmo Prim

Ideia: crescer a MST a partir de um vértice, sempre escolhendo a aresta de menor peso que conecta a árvore a um novo vértice

Estratégia: O algoritmo mantém um único conjunto de vértices que já fazem parte da MST. A cada passo, ele encontra a aresta de menor peso (a aresta "leve") que conecta um vértice dentro da árvore a um vértice fora da árvore e a adiciona à MST.

Passos:

1. Escolher vértice inicial s.
2. Inicializar conjunto $A = \emptyset$.
3. Usar uma **fila de prioridade** para manter arestas de menor peso que conectam a árvore a vértices ainda fora dela.
4. Repetir até incluir todos os vértices:
   a. Escolher a aresta (u, v) de menor peso que conecta um vértice dentro da árvore a um fora dela.
   b. Adicionar (u,v)(u,v)(u,v) a AAA.

Complexidade:

- Com min-heap binário: $O(E \log V)$.
- Com Fibonacci heap: $O(E + V \log V)$.

Vantagem: eficiente em grafos **densos**

```
PRIM-MST(G, w, r)
    para cada u ∈ V[G] faça
        chave[u] ← ∞
        π[u] ← NIL
    chave[r] ← 0

    Q ← V[G]     // Q é uma fila de prioridade mínima

    enquanto Q ≠ ∅ faça
        u ← EXTRACT-MIN(Q)
        para cada v ∈ Adj[u] faça
            se v ∈ Q e w(u, v) < chave[v] então
                π[v] ← u
                chave[v] ← w(u, v)
```

`chave[v]` : custo mínimo conhecido para conectar `v` à árvore parcial.
`π[v]` : predecessor de `v` na MST.
`EXTRACT-MIN(Q)` : remove da fila o vértice com menor `chave` .

9. Complexidade do Algoritmo de Prim

A complexidade de tempo do algoritmo de Prim depende crucialmente da implementação da fila de prioridade, que é usada para armazenar os vértices que ainda não estão na árvore.

1. **Matriz de adjacência + busca linear**
   a. Para cada extração e atualização: $O(V)$.
   b. Complexidade total: $O(V^2)$.
   c. Bom para grafos **densos** ($E \approx V^2$).
2. **Min-Heap binário**
   a. Cada `EXTRACT-MIN` : $O(\log V)$), repetido V vezes.
   b. Cada atualização (`DECREASE-KEY`): $O(\log V)$, feito até E vezes.
   c. Complexidade total: $O(E \log V)$.
3. **Heap de Fibonacci**
   a. `EXTRACT-MIN` : $O(\log V)$.
   b. `DECREASE-KEY` : $O(1)$ amortizado.
   c. Complexidade total: $O(E + V \log V)$.
   d. Melhor escolha em grafos **muito densos**.

Kruskal vs. Prim

Ambos têm desempenho semelhante em grafos médios, a escolha depende do tipo de grafo (esparso ou denso)

Aspecto	Kruskal	Prim
Estratégia	Ordena arestas globais e escolhe em ordem crescente (Union-Find)	Cresce uma única árvore a partir de um vértice inicial
Estrutura de dados	Union-Find (disjoint set)	Fila de prioridade (heap)
Complexidade	$O(E \log V)$	$O(E \log V)$ (heap binário) - $O(E + V \log V)$ (heap Fibonacci) - $O(V^2)$ (matriz)
Melhor para	**Grafos esparsos** (poucas arestas)	**Grafos densos** (muitas arestas)
Natureza da construção	Floresta → conecta componentes	Uma árvore única que cresce passo a passo
Facilidade de implementação	Mais simples	Mais complexo (manipulação de heap)

Característica	Algoritmo de Kruskal	Algoritmo de Prim
Estratégia	Constrói a MST como uma floresta, unindo árvores componentes ao adicionar a aresta de menor peso que não forma um ciclo.	Constrói a MST a partir de um único vértice, expandindo a árvore ao adicionar a aresta mais barata que conecta um vértice da árvore a um vértice fora dela.
Foco	Foca nas arestas do grafo inteiro, processando-as em ordem de peso.	Foca nos vértices, crescendo uma única árvore a partir de uma raiz arbitrária.
Complexidade	**O(E log E)**, dominada pela ordenação das arestas.	**O(E log V)** com heap binário ou **O(E + V log V)** com heap de Fibonacci.
Quando Usar	Geralmente mais rápido para **grafos esparsos** (onde o número de arestas E é significativamente menor que V^2), pois `log E` é próximo a `log V`.	Geralmente mais rápido para **grafos densos** (onde E está próximo de V^2), especialmente com a implementação de heap de Fibonacci, pois sua complexidade se aproxima de O(E).
Estrutura de Dados	Utiliza a estrutura de dados de **conjuntos disjuntos (disjoint-set)** para detectar ciclos eficientemente.	Utiliza uma **fila de prioridade** para encontrar eficientemente a aresta mais leve que cruza o corte entre a árvore e o resto do grafo.