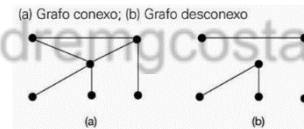


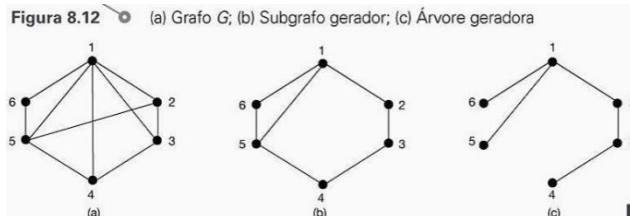
Grafos

Grafo $G = (V, E)$

V – conjunto dos nós (ou vértices)
 $E \subseteq V^2$ – conjunto dos arcos (ou arestas)



- Dois vértices ligados por uma aresta são denominados **adjacentes**.
- Grafo **simples** se não possui laços ou arestas múltiplas.
- Grafo **completo** quando existe uma aresta para cada par de vértices distintos.
- Grafo **regular** quando todos os vértices de um grafo possuem o mesmo grau. Grau: o grau de um vértice é o número de arestas que incidem nele.
- O número de arestas do caminho é denominado **comprimento do caminho**. Quando todos os vértices do caminho são distintos, a sequência recebe o nome de caminho simples ou elementar. Se as arestas forem diferentes recebe o nome de trajeto.
- Quando um grafo não possui ciclos é chamado **acíclico**.
- Grafo **conexo** se existe um caminho para cada par de vértices de G . Caso contrário é chamado **desconexo**. G é **fortemente conexo** se existe algum caminho de qualquer nó para qualquer nó.
- Dois grafos $G_1(V_1, E_1)$ e $G_2(V_2, E_2)$, com $|V_1| = |V_2| = n$, são **isomorfos** entre si, quando existe uma função unívoca $f: V_1 \rightarrow V_2$ tal que (u, v) pertence a E_1 se e somente se $(f(u), f(v))$ pertencer a E_2 .
- Grafo **pesado** é aquele com peso nas arestas.
- Os grafos que não possuem orientação nas suas arestas são chamados de **não orientado**. Um grafo **orientado** também chamado de **dígrafo**, possui um conjunto de arestas, tal que para cada aresta existe uma única direção entre u e v .
- Um grafo que não possui ciclos e é conexo é chamado de **árvore**. Uma árvore T com n vértices possui $n-1$ arestas.
- Percurso em largura (BFS)**: Dado um grafo G e um vértice de origem s , a procura em largura explora as arestas de G até explorar todos os vértices alcançáveis a partir de s . Além disso também calcula a menor distância em número de arestas de s até todos os vértices acessíveis a ele.
- Percurso em profundidade (DFS)**: Nós são visitados pela ordem por que são encontrados.
- Ordenação topológica**: Podemos usar busca em profundidade para executar uma ordenação topológica de um grafo acíclico dirigido. Se contém um ciclo, nenhuma ordenação é possível.
- Árvore de cobertura mínima**: Seja G um grafo pesado não orientado conexo, uma árvore de cobertura (ou geradora) é um grafo não orientado conexo acíclico. Uma árvore de cobertura mínima de G é uma árvore de cobertura de peso mínimo.
- Algoritmo de Prim**: começa com um vértice e cresce passo a passo, adicionando sempre a aresta de menor peso. Faz uso da fila de prioridade. Garante que a MST resultante seja conectada e tenha o menor peso possível. **O Algoritmo de Kruskal**: Ordena todas as arestas em ordem crescente de peso e, em seguida constrói a MST adicionando arestas em ordem até que todos os vértices estejam conectados.



Percurso em largura (a partir do vértice s)

```
BFS(G, s)
1 for each vertex u in G.V - {s} do
2   u.color <- WHITE
3   u.d <- INFINITY // distância para s
4   u.p <- NIL // antecessor no caminho
5 s.color <- GREY
6 s.d <- 0
7 s.p <- NIL
8 Q <- EMPTY // fila (FIFO)
9 ENQUEUE(Q, s)
10 while Q != EMPTY do
11   u <- DEQUEUE(Q) // próximo nó a explorar
12   for each vertex v in G.adj[u] do
13     if v.color = WHITE then
14       v.color <- GREY
15       v.d <- u.d + 1
16       v.p <- u
17       ENQUEUE(Q, v)
18   u.color <- BLACK // u foi explorado
```

Percurso em profundidade

```
DFS(G)
1 for each vertex u in G.V do
2   u.color <- WHITE
3   u.p <- NIL // variável global
4 time <- 0 // explora todos os nós
5 for each vertex u in G.V do
6   if u.color = WHITE then
7     DFS-VISIT(G, u)

DFS-VISIT(G, u)
1 time <- time + 1 // instante da descoberta do
2 u.d <- time // vértice u
3 u.color <- GREY
4 for each vertex v in G.adj[u] do // explora arco (u, v)
5   if v.color = WHITE then
6     v.p <- u
7     DFS-VISIT(G, v)
8 u.color <- BLACK // u foi explorado
9 time <- time + 1 // instante em que termina
10 u.f <- time // a exploração de u
```

Árvore de cobertura mínima

Algoritmo de Prim

$G = (V, E)$ – grafo pesado não orientado conexo

MST-PRIM(G, w, s)

```
1 for each vertex u in G.V do
2   u.key <- INFINITY // custo de juntar u à MST
3   u.p <- NIL // nó a que u é ligado
4 s.key <- 0
5 Q <- G.V // fila com prioridade, por
// mínimos, chave é u.key
6 while Q != EMPTY do
7   u <- EXTRACT-MIN(Q)
8   for each vertex v in G.adj[u] do
9     if v in Q and w(u, v) < v.key then
10      v.p <- u // arco (u, v) é candidato
11      v.key <- w(u, v) // pode alterar Q!
```

Ordenação topológica

Adaptação de DFS

$G = (V, E)$ – grafo orientado acíclico (DAG)

TOPOLOGICAL-SORT(G)

```
1 for each vertex u in G.V do
2   u.color <- WHITE
3 L <- EMPTY // lista, global
4 for each vertex u in G.V do
5   if u.color = WHITE then
6     DFS-VISIT'(G, u)
7 return L
```

DFS-VISIT'(G, u)

```
1 u.color <- GREY
2 for each vertex v in G.adj[u] do
3   if v.color = WHITE then
4     DFS-VISIT'(G, v)
5 u.color <- BLACK
6 LIST-INSERT-HEAD(L, u) // insere
```

Partição

Implementação em vector

MAKE-SETS(n)

```
1 let P[1..n] be a new array
2 for i <- 1 to n do
3   P[i] <- -1 // i is the representative for set {i}
4 return P
```

FIND-SET-BASIC(P, i)

```
1 while P[i] > 0 do
2   i <- P[i]
3 return i
```

UNION-BASIC(P, i, j)

```
1 P[FIND-SET(P, j)] <- FIND-SET(P, i)
```

Árvore de cobertura mínima

Algoritmo de Kruskal

$G = (V, E)$ – grafo pesado não orientado conexo

MST-KRUSKAL(G, w)

```
1 v <- |G.V|
2 A <- EMPTY // conjunto dos arcos da MST
3 P <- MAKE-SETS(G.V) // partição de G.V, floresta
4 Q <- G.E // fila com prioridade, por
// mínimos, chave é w(u, v)
// número de arcos na MST
5 e <- 0
6 while e < v - 1 do
7   (u, v) <- EXTRACT-MIN(Q)
8   if FIND-SET(P, u) != FIND-SET(P, v) then
9     A <- A + {(u, v)} // novo arco da MST
10    UNION(P, u, v)
11    e <- e + 1
12 return A
```

Abstracção da implementação de conjuntos disjuntos com os elementos do conjunto $\{1, 2, \dots, n\}$

Operações suportadas

MAKE-SETS(n)

Cria conjuntos singulares com os elementos $\{1, 2, \dots, n\}$

FIND-SET(i)

Devolve o representante do conjunto que contém o elemento i

UNION(i, j)

Reúne os conjuntos a que pertencem os elementos i e j

Reunião por tamanho (union by size/rank): Em vez de simplesmente unir um conjunto ao outro sem considerar seus tamanhos, a reunião por tamanho escolhe a raiz com base no tamanho do conjunto. O conjunto menor é unido ao conjunto maior, para que a estrutura permaneça mais balanceada e a profundidade das árvores seja reduzida.

Compressão de caminho (path compression): Durante a busca pelo representante (raiz) de um elemento, a compressão de caminho é aplicada para atualizar os ponteiros ao longo do caminho percorrido, fazendo com que eles apontem diretamente para a raiz. Essa compressão de caminho reduz a altura da árvore, tornando as futuras buscas mais eficientes.

- **Algoritmo de Dijkstra:** só aplicável a grafos sem pesos negativos. Começa num ponto e atualiza-se a distância aos adjacentes (a distância da fonte a ela mesma é 0). Pego sempre na distância mais curta para ver os adjacentes. Vértice v - $[p,d]$ p -pai d -distancia.
- **Algoritmo de Bellman-Ford:** aplicável a qualquer grafo pesado.
- **Algoritmo de Floyd-Warshall:** Arestas de peso negativo podem estar presentes, valores diferentes de 0 na diagonal principal indica que há ciclos negativos no grafo. O algoritmo assume que não existem ciclos negativos no grafo.
- **Algoritmo de Kruskal:** Colocar os vértices, ordenar as arestas pelos tamanhos da menor para a maior. Ir adicionando sem criar ciclos até estarem todos os vértices ligados.

FLOYD-WARSHALL(w)

```

1 n <- w.rows
2 d <- w
3 for k <- 1 to n do
4   for i <- 1 to n do
5     for j <- 1 to n do
6       if d[i,k] + d[k,j] < d[i,j] then
7         d[i,j] <- d[i,k] + d[k,j]
8 return d

```

BELLMAN-FORD(G, w, s)

```

1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i <- 1 to |G.V| - 1 do
3   for each edge (u,v) in G.E do
4     RELAX(u, v, w)
5 for each edge (u,v) in G.E do
6   if u.d + w(u,v) < v.d then
7     return FALSE
8 return TRUE

```

DIJKSTRA(G, w, s)

```

1 INITIALIZE-SINGLE-SOURCE(G, s)
2 Q <- G.V // fila com prioridade, chave u.d
3 while Q != EMPTY do
4   u <- EXTRACT-MIN(Q)
5   for each vertex v in G.adj[u] do
6     RELAX(u, v, w) // pode alterar Q!

```

O peso do caminho mais curto de s a qualquer outro nó é inicializado com ∞

INITIALIZE-SINGLE-SOURCE(G, s)

```

1 for each vertex v in G.V do
2   v.d <- INFINITY // peso do caminho mais curto de s a v
3   v.p <- NIL // predecessor de v nesse caminho
4 s.d <- 0

```

Se o caminho de s a v , que passa por u e pelo arco (u, v) , tem menor peso do que o mais curto anteriormente encontrado, encontrámos um caminho mais curto

RELAX(u, v, w)

```

1 if u.d + w(u,v) < v.d then
2   v.d <- u.d + w(u,v)
3   v.p <- u

```

Em cada *rede de fluxos* existem dois pontos especiais:

- **Fonte (source)** Origem de tudo o que flui na rede
- **Dreno (sink)** Destino final de tudo o que flui na rede

Rede de fluxos (Flow network)

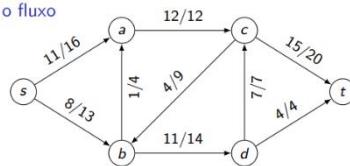
- Modelada através de um **grafo orientado** $G = (V, E)$
- $c(u, v) > 0$ é a **capacidade** do arco (u, v)
- $s \in V$ é a **fonte (source)** da rede
- $t \in V$ é o **dreno (sink)** da rede ($s \neq t$)
- Se $(u, v) \in E$, então $(v, u) \notin E$
- Assume-se que, qualquer que seja o vértice $v \in V$, existe um caminho $s \dots v \dots t$
(Logo, $|E| \geq |V| - 1$)

Numa *rede residual*

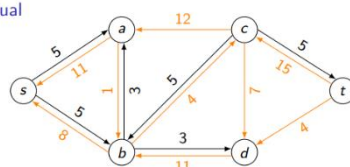
- A capacidade dos arcos comuns à rede original corresponde à capacidade não utilizada **peleto fluxo**
- A capacidade dos arcos com **orientação oposta** à dos da rede original corresponde à quantidade de fluxo que pode ser cancelada

Uma rede residual indica os limites das **alterações** que podem ser feitas a um fluxo em cada ligação

Rede com o fluxo



Rede residual



Majorante para o valor do fluxo na rede:
Soma da saída de fluxo da fonte

Método de Ford-Fulkerson

1. Inicializar $f(u, v) = 0$, para todo $(u, v) \in E$
2. Enquanto houver um caminho simples $p = s \dots t$ na rede residual
 - a. Seja $c_f(p) = \min \{ c_f(u, v) \mid (u, v) \text{ está em } p \}$
 - b. Para cada arco (u, v) em p
 - Se $(u, v) \in E$, o fluxo no arco (u, v) é **aumentado** em $c_f(p)$ unidades
 $f(u, v) = f(u, v) + c_f(p)$
 - Senão, então $(v, u) \in E$ e são **canceladas** $c_f(p)$ unidades de fluxo no arco (v, u)
 $f(v, u) = f(v, u) - c_f(p)$

Algoritmo de Edmonds-Karp

EDMONDS-KARP(G, s, t)

```

1 for each edge (u,v) in G.E do
2   (u,v).f <- 0 // fluxo f(u,v) = 0
3 Gf <- RESIDUAL-NET(G)
4 while (cf <- BFS-FIND-PATH(Gf, s, t)) > 0 do
5   v <- t
6   while v.p != NIL do
7     if edge (v.p,v) is in G.E then
8       (v.p,v).f = (v.p,v).f + cf // aumenta
9     else // edge (v,v.p) is in G.E
10      (v,v.p).f = (v,v.p).f - cf // cancela
11   v <- v.p
12 UPDATE(Gf, G) // atualiza rede residual

```

Percurso em largura (BFS): É usado para explorar ou pesquisar todos os nós de um grafo de maneira sistemática, visitando primeiro os nós vizinhos antes de avançar para os próximos níveis. **Percurso em profundidade (DFS):** Permite explorar ou pesquisar um grafo, visitando os nós o mais a fundo possível antes de retroceder. É útil para encontrar componentes conexas, ciclos e outras propriedades dos grafos. **Ordenação topológica:** É um algoritmo aplicado a grafos direcionados acíclicos (DAGs) para ordenar os nós de forma que todas as arestas apontem para nós anteriores na ordem. **Grafo transposto:** É obtido revertendo as direções de todas as arestas de um grafo direcionado. É útil para resolver problemas relacionados a grafos direcionados. **Cálculo das componentes fortemente conexas:** Permite identificar e agrupar os nós de um grafo direcionado em componentes, onde cada nó de uma componente é alcançável por qualquer outro nó dessa mesma componente. **Algoritmo de Prim e de Kruskal:** Ambos são algoritmos de árvore geradora mínima, usados para encontrar a árvore que conecta todos os nós de um grafo ponderado com o custo total mínimo. O algoritmo de Prim começa de um nó inicial e expande a árvore adicionando a aresta de menor peso, enquanto o algoritmo de Kruskal constrói a árvore adicionando arestas em ordem crescente de peso. Caminhos mais curtos num DAG: DAG significa "Directed Acyclic Graph" (Grafo Direcionado Acíclico). Os algoritmos para caminhos mais curtos em um DAG, como o algoritmo de Bellman-Ford ou o algoritmo de Dijkstra, permitem encontrar o

Complexidade dos algoritmos

$G = (V, E)$

Compl. Temporal

Percurso em largura	$O(V + E)$
Percurso em profundidade	$\Theta(V + E)$
Ordenação topológica (ambos os algoritmos)	$\Theta(V + E)$
Grafo transposto	$\Theta(V + E)$
Cálculo das componentes fortemente conexas	$\Theta(V + E)$
Algoritmos de Prim e de Kruskal	$O(E \log V)$
Caminhos mais curtos num DAG	$\Theta(V + E)$
Algoritmo de Dijkstra	$O((V + E) \log V)$
Algoritmo de Bellman-Ford	$O(VE)$
Algoritmo de Floyd-Warshall	$\Theta(V^3)$

Pressupostos

Grafo representado através de listas de adjacências (excepto algoritmos de Kruskal, de Bellman-Ford e de Floyd-Warshall)

Algoritmos de Prim e de Dijkstra recorrem a uma fila tipo *heap* binário com atualização (EXTRACT-MIN e DECREASE-KEY com complexidade temporal logarítmica no número de elementos da fila)

Algoritmo de Kruskal usa Partição com compressão de caminho