Algoritmos em grafos

Prof. Lilian Berton

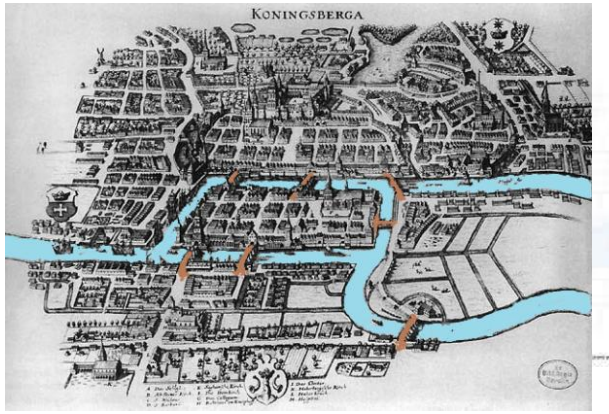
São José dos Campos, 2018

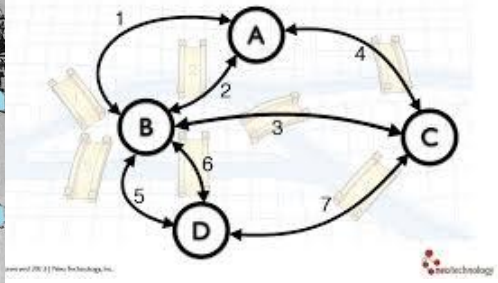
Introdução: motivação

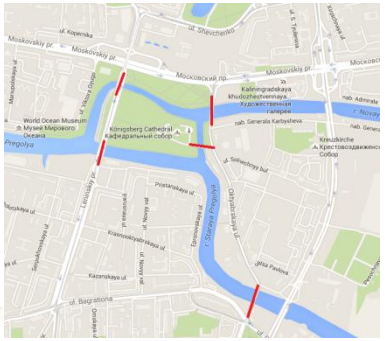
- Muitas aplicações em computação necessitam considerar um conjunto de conexões entre pares de objetos.

- Existe um tipo abstrato chamado **grafo** que é usado para modelar tais situações.

- A teoria dos grafos iniciou-se em com Leonhard Euler, em 1736 com um artigo sobre as 7 pontes de Königsberg (Kaliningrad).

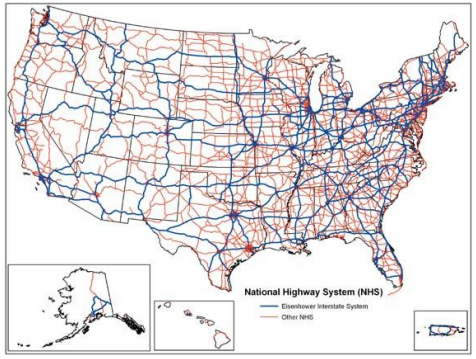






Introdução: aplicações

- **Rede de computadores**: protocolos de roteamento.
- **Vértice central**: instalação de uma loja no local que diminui a distância entre ela e locais estratégicos. Ex: estabelecimentos que prestarão serviços, concentração de público alvo.
- **Mapa rodoviário**: encontrar caminho mais curto entre duas cidades.
- **Máquinas de busca**: localizar informação relevante na Web.

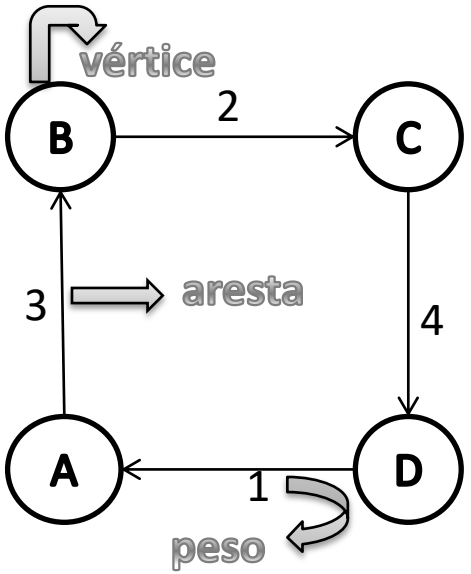


Introdução: aplicações

Grafo	Vértice	Aresta
Comunicação	Centrais telefônicas, Computadores, Satélites	Cabos, Fibra óptica, Enlaces de microondas
Circuitos	Portas lógicas, registradores, processadores	Filamentos
Hidráulico	Reservatórios, estações de bombeamento	Tubulações
Financeiro	Ações, moeda	Transações
Transporte	Cidades, Aeroportos	Rodovias, Vias aéreas
Escalonamento	Tarefas	Restrições de precedência
Arquitetura funcional de um software	Módulos	Interações entre os módulos
Internet	Páginas Web	*Links*
Jogos de tabuleiro	Posições no tabuleiro	Movimentos permitidos
Relações sociais	Pessoas, Atores	Amizades, Trabalho conjunto em filmes

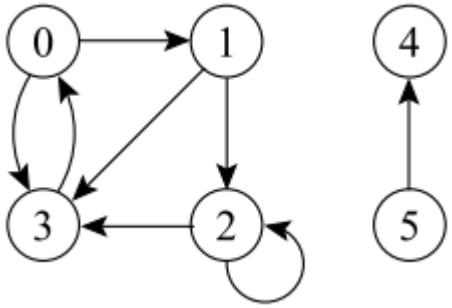
Definição de grafo

- **Grafo (G)**: conjunto de vértices e arestas.
- **Vértice (V)**: representa um objeto.
- **Aresta (E)**: representa uma conexão entre dois vértices.
- **Peso (W)**: representa similaridade/distância/ custo entre dois vértices.
- **Notação**: G = (V,E,W)

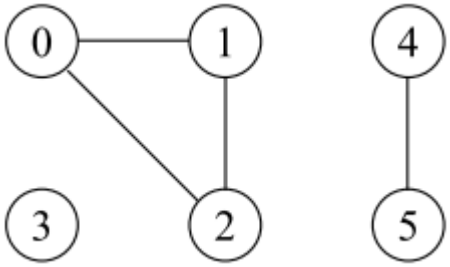


Grafos direcionados x não-direcionados

- **Grafos direcionados**: uma aresta (u, v) sai do vértice u e entra no vértice v. O vértice v é adjacente ao vértice u.
- Podem existir arestas de um vértice para ele mesmo, chamadas de self-loops.

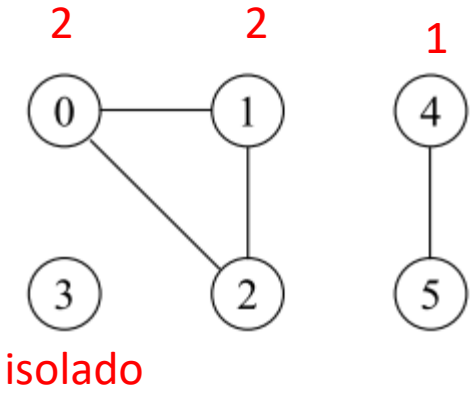


- **Grafos não-direcionados**: as arestas (u, v) e (v, u) são consideradas como uma única aresta. A relação de adjacência é simétrica.
- Self-loops não são permitidos.

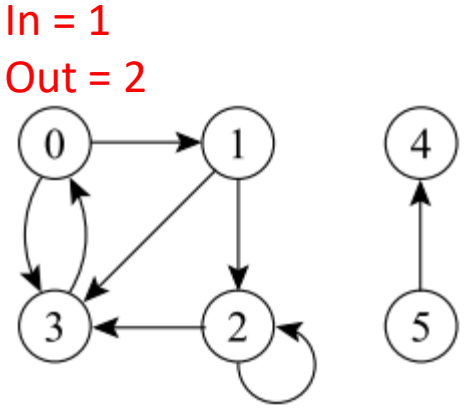


Grau de um vértice

- **Em grafos não direcionados**:
  - O grau de um vértice é o número de arestas que incidem nele.
  - Um vértice de grau zero é dito isolado ou não conectado.



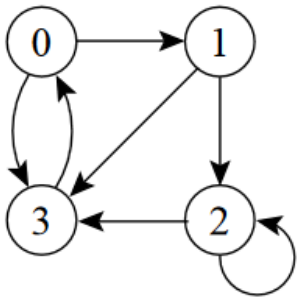
- **Em grafos direcionados**:
  - O grau de um vértice é o número de arestas que saem dele (out-degree) mais o número de arestas que chegam nele (in-degree).



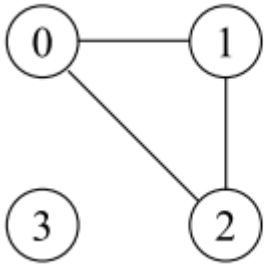
Implementação usando matriz de adjacência

Matriz de adjacência:

• A matriz de adjacência de um grafo G contendo n vértices é uma matriz n x n, onde A[i,j] = 1 se existir uma aresta do vértice i ao vértice j. No caso de grafos ponderados contém o peso da aresta. Caso contrário A[i,j] = 0.



	0	1	2	3
0		1		1
1			1	1
2			1	1
3	1			

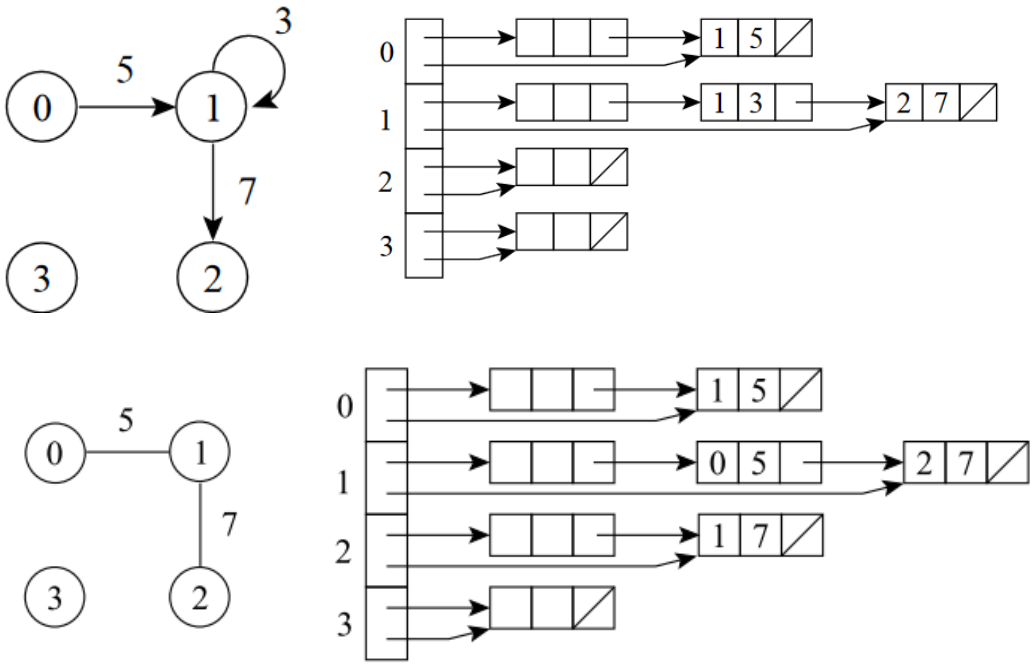


	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						
5						

Implementação usando lista-encadeada

Lista encadeada:

• Utiliza-se uma lista para cada vértice em V e cada lista possui todos os vértices adjacentes a i.



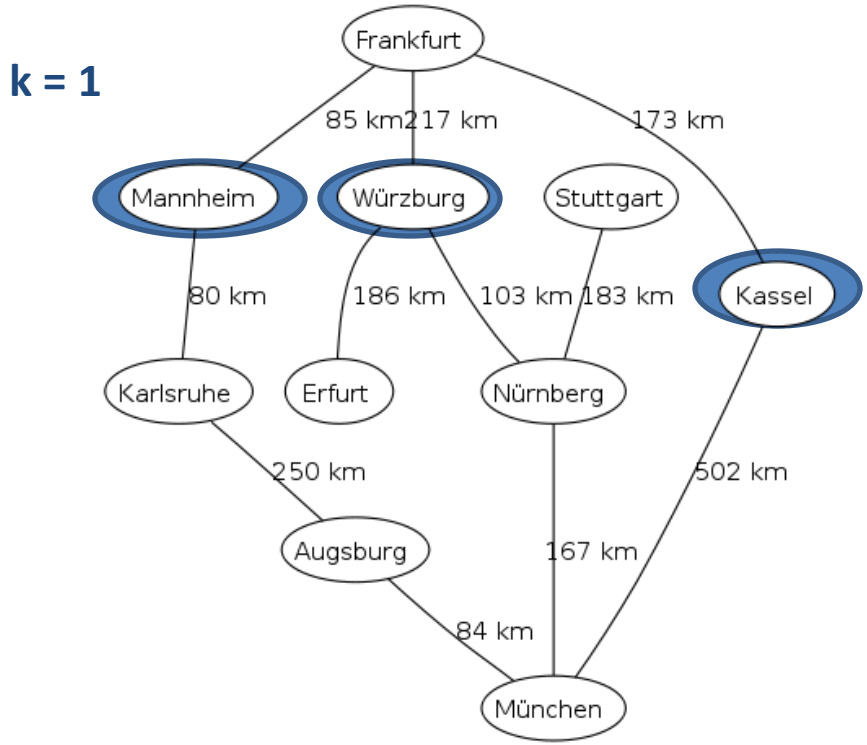
Matriz x Lista

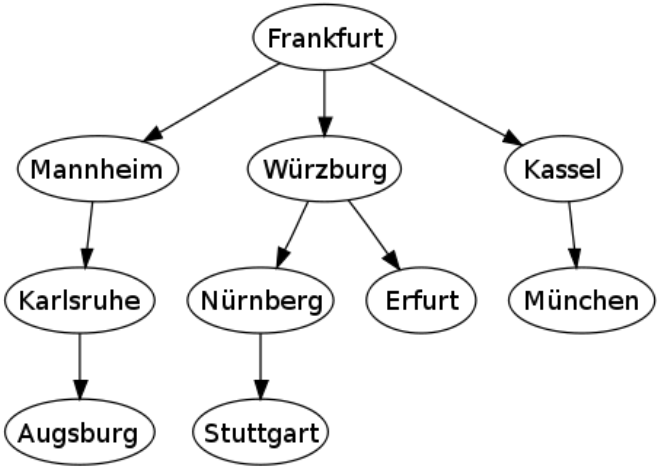
- **Matriz:**
- Deve ser utilizada por **grafos densos**, onde |E| é próximo de $|V|^2$.
- O tempo necessário para acessar um elemento é independente de |V| ou |A|.
- A maior desvantagem é que a matriz necessita **O($|V|^2$) de espaço**.
- Ler ou examinar a matriz tem complexidade de tempo O($|V|^2$).

- **Lista:**
- Indicada para **grafos esparsos**, onde |E| é muito menor que $|V|^2$.
- Possui uma complexidade de **espaço O(|V| + |A|).**
- A principal desvantagem é que ela pode ter tempo O(|V|) para determinar se existe uma aresta entre o vértice i e o vértice j, pois podem existir O(|V|) vértices na lista de adjacentes do vértice i.

Busca em largura

- O algoritmo descobre todos os vértices a uma distância k = 1 do vértice origem antes de descobrir qualquer vértice a uma distância k+1.

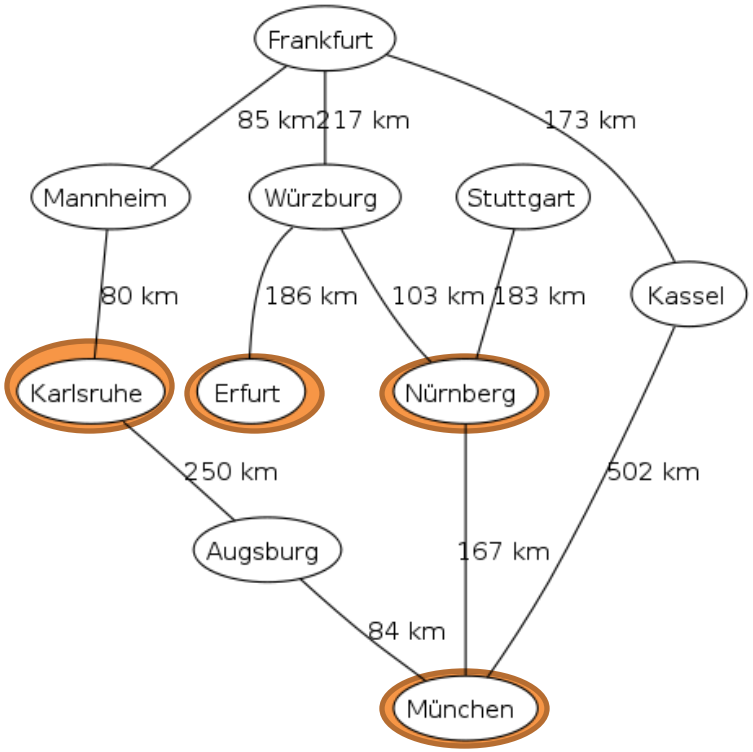


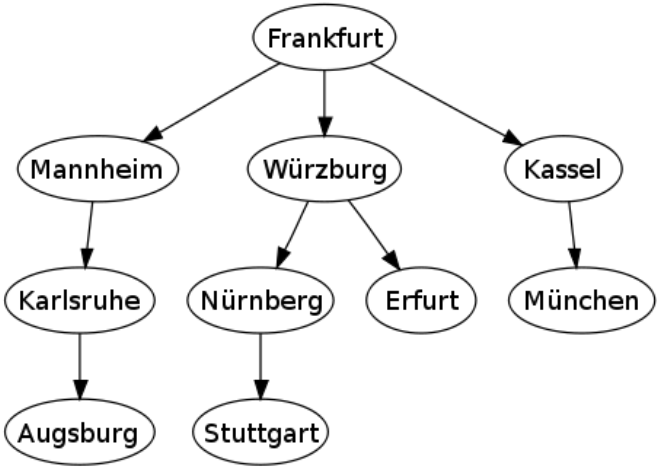


Busca em largura

- O algoritmo descobre todos os vértices a uma distância k = 1 do vértice origem antes de descobrir qualquer vértice a uma distância k+1.

k = 2

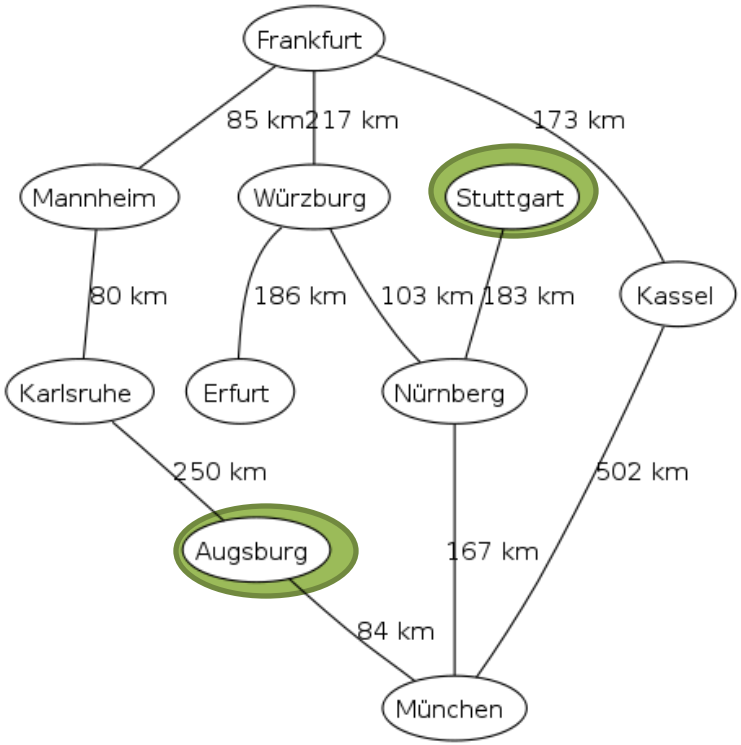


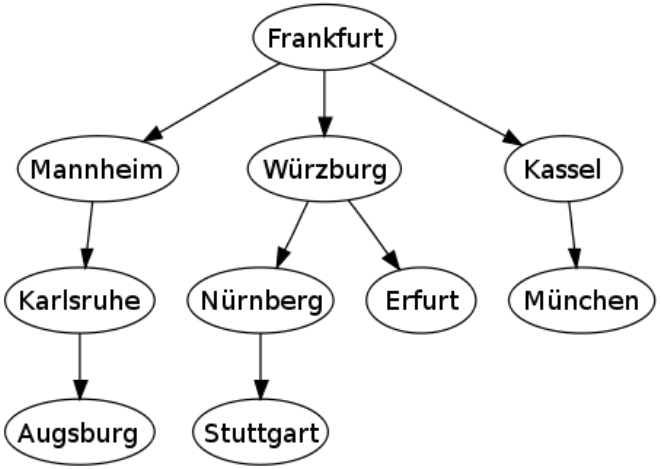


Busca em largura

- O algoritmo descobre todos os vértices a uma distância k = 1 do vértice origem antes de descobrir qualquer vértice a uma distância k+1.

k = 3





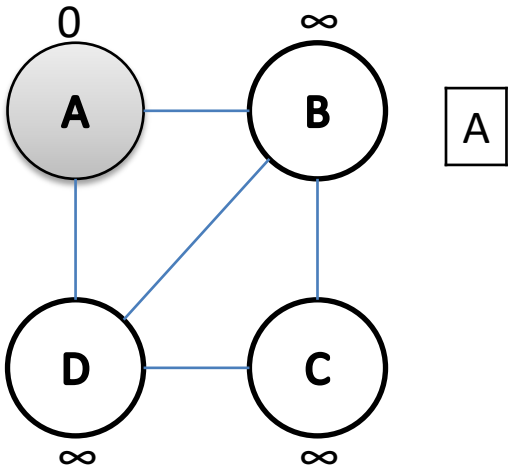
Busca em largura

- O algoritmo calcula a distância (menor número de arestas) desde s até todos os vértices acessíveis a partir de s. **É um algoritmo de caminho mínimo sobre grafos não ponderados.**

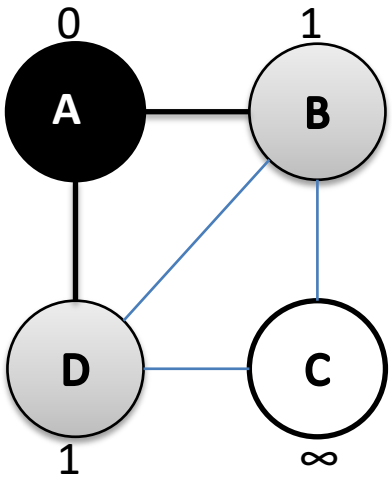
- Para controlar o andamento, a busca pinta cada vértice de BRANCO, CINZA ou **PRETO**.

1. No início todos os vértices são brancos.
2. Quando é descoberto pela primeira vez torna-se cinza. Os vértices de cor cinza podem ter adjacentes brancos e representam a fronteira entre vértices descobertos e não descobertos.
3. Já todos os vértices adjacentes a vértices pretos foram descobertos.

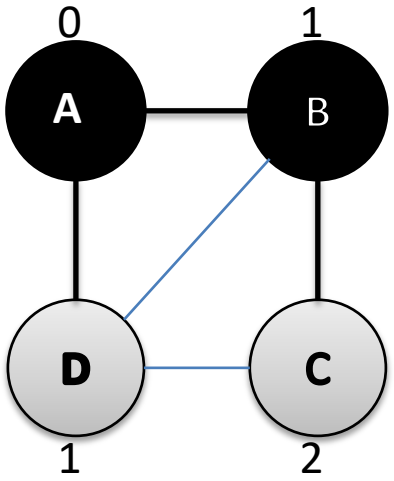
Exemplo



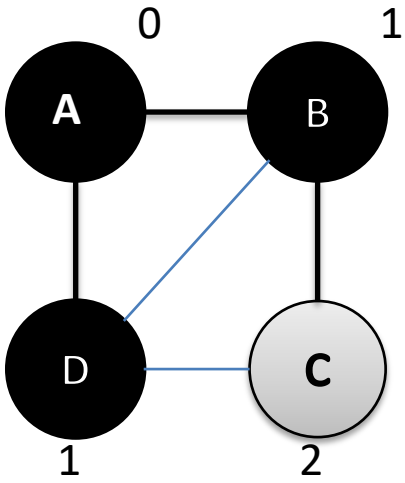
A



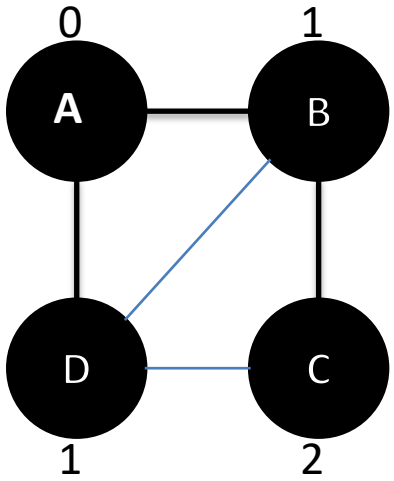
B D

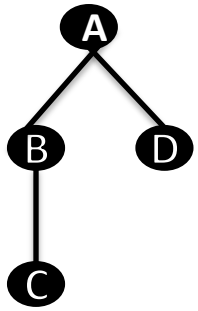


D C



C





Algoritmo

```
BFS(G, s)
 1  for cada vértice u ∈ V[G] – {s}
 2       do cor[u] ← BRANCO
 3          d[u] ← ∞
 4          α[u] ← NIL
 5  cor[s] ← CINZA
 6  d[s] ← 0
 7  α[s] ← NIL
 8  Q ← 0
 9  ENQUEUE(Q, s)
10   while Q ≠ 0
11     do u ← DEQUEUE(Q)
12        for cada v ← Adj[u]
13           do if cor[v] = BRANCO
14                 then cor[v] ← CINZA
15                      d[v] ← d[u] + 1
16                      π[v] ← u
17                      ENQUEUE(Q, v)
18        cor[u] ← PRETO
```

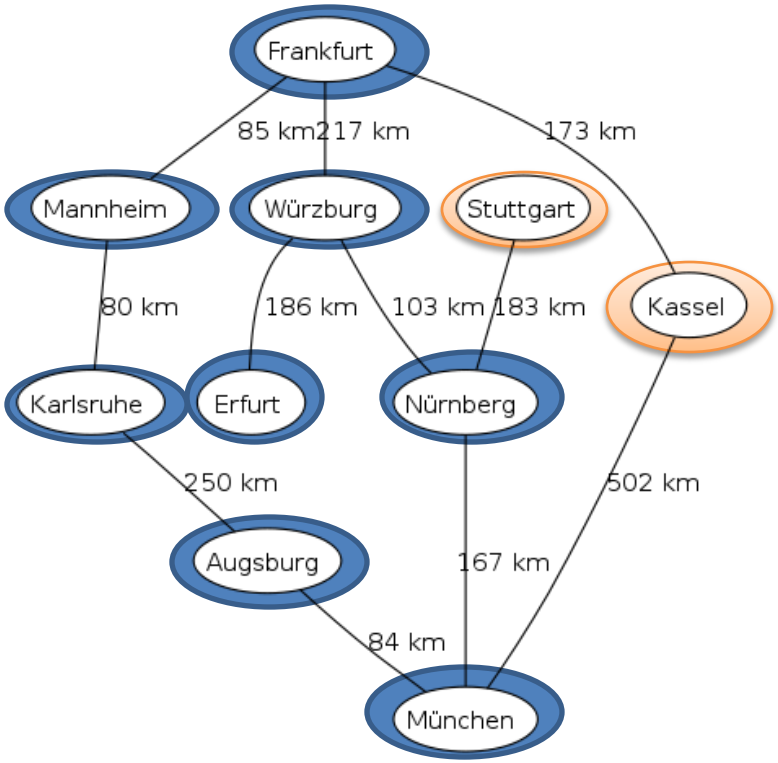
Complexidade

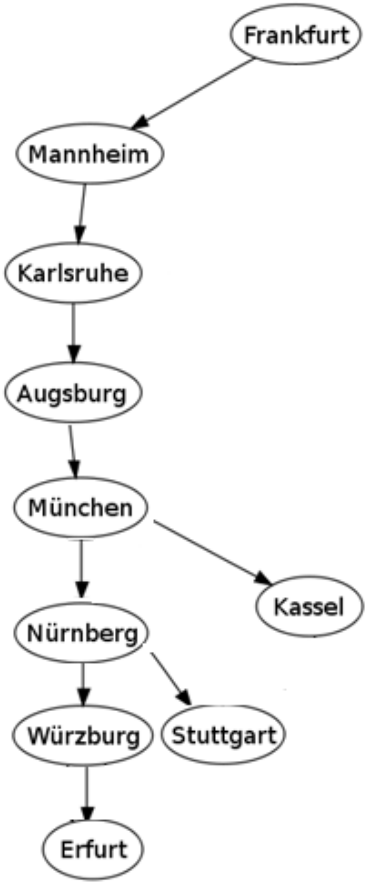
- O custo do primeiro anel é O(V).
- O tempo dedicado as operações de filas é O(V).
- Como a soma de todas as listas de adjacências é O(E), o gasto máximo na varredura das listas de adjacência é O(E).
- O tempo total é **O(V+E).**

```
BFS(G, s)
 1  for cada vértice u ∈ V[G] – {s}
 2       do cor[u] ← BRANCO
 3          d[u] ← ∞
 4          α[u] ← NIL
 5  cor[s] ← CINZA
 6  d[s] ← 0
 7  α[s] ← NIL
 8  Q ← 0
 9  ENQUEUE(Q, s)
10   while Q ≠ 0
11     do u ← DEQUEUE(Q)
12        for cada v ← Adj[u]
13           do if cor[v] = BRANCO
14              then cor[v] ← CINZA
15                   d[v] ← d[u] + 1
16                   π[v] ← u
17                   ENQUEUE(Q, v)
18        cor[u] ← PRETO
```

Busca em profundidade

- Essa estratégia procura mais “fundo” no grafo enquanto for possível.
- A busca explora sempre o vértice recém descoberto e depois “regressa” para explorar os respectivos antecessores.





Busca em profundidade

- O algoritmo registra duas **variáveis de tempo:**

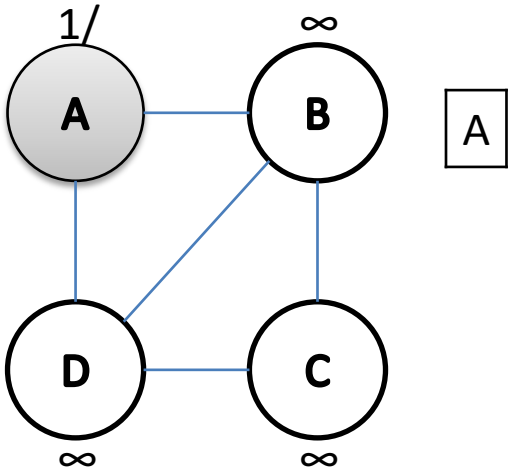
1. d[u] o momento em que o vértice u é descoberto
2. f[v] o momento em que termina o vértice u.

- Esses tempos são inteiros entre 1 e 2|V|, pois existe um evento de descoberta e um evento de término para cada um dos |V| vértices.

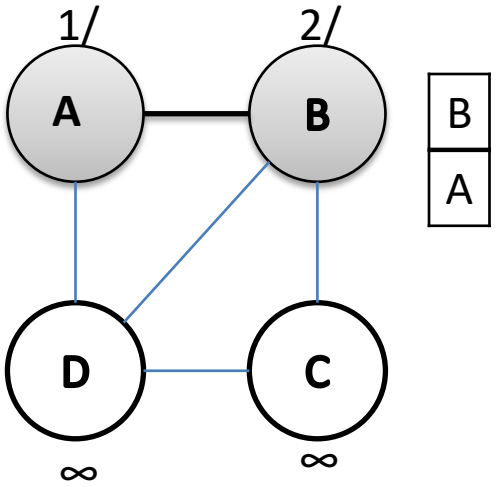
- Para controlar o andamento, a busca pinta cada vértice de BRANCO, CINZA ou **PRETO**.

1. No início todos os vértices são brancos.
2. Quando é descoberto pela primeira vez torna-se cinza.
3. Já todos os vértices adjacentes foram examinados torna-se preto.

Exemplo

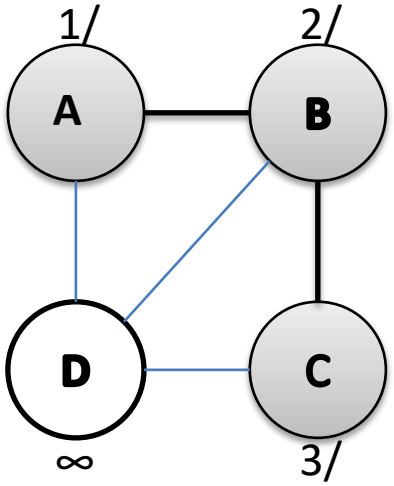


A



B

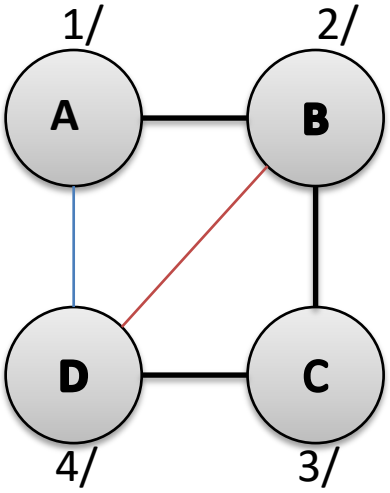
A



C

B

A

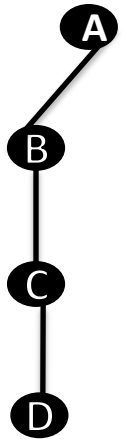


D

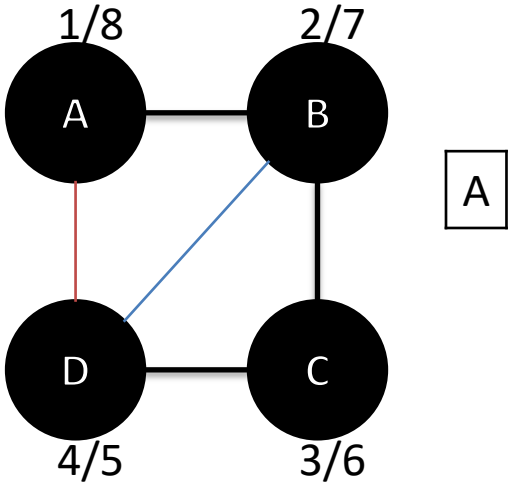
C

B

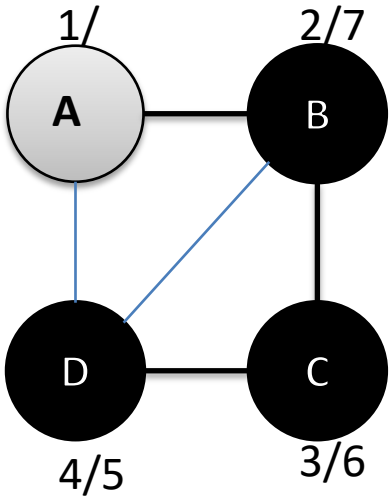
A



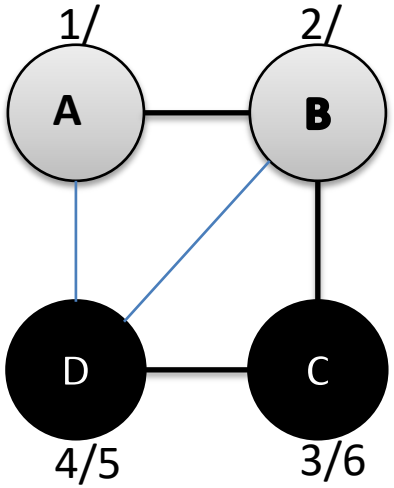
Exemplo



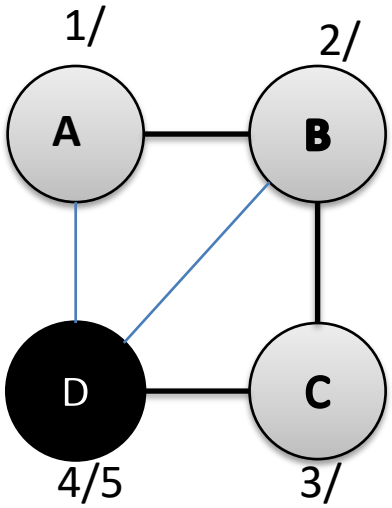
A



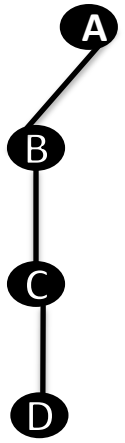
B
A



C
B
A



D
C
B
A



Algoritmo

```
DFS(G)
1 for cada vértice u ← V[G]
2     do cor[u] ← BRANCO
3        π[u] ← NIL
4 tempo ← 0
5 for cada vértice u ∈ V[G]
6     do if cor[u] = BRANCO
7           then DFS-VISIT(u)
```

Pode ser implementado como uma pilha!

```
DFS-VISIT(u)
1 cor[u] ← CINZA          ▷ Branco, o vértice u acabou de ser descoberto.
2 tempo ← tempo + 1
3 d[u] ← tempo
4 for cada v ∈ Adj[u]     ▷ Explora a aresta (u, v).
5     do if cor[u] = BRANCO
6           then π[v] ← u
7                DFS-VISIT(v)
8 cor[u] ← PRETO          ▷ Enegrece u; terminado.
9 f[u] ← tempo ← tempo + 1
```

Complexidade

- O custo dos dois anéis é O(V).

```
DFS(G)
1 for cada vértice u ← V[G]
2     do cor[u] ← BRANCO
3        π[u] ← NIL
4 tempo ← 0
5 for cada vértice u ∈ V[G]
6     do if cor[u] = BRANCO
7           then DFS-VISIT(u)
```

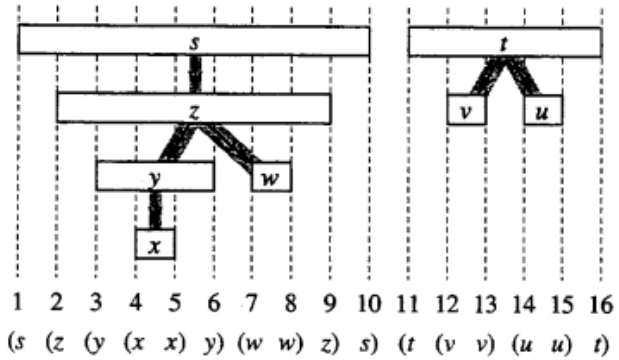
- O tempo dedicado as operações DFS-VISIT é O(V).
- O loop das linhas 4-7 é executado O(E).
- O tempo total é **O(V+E).**

```
DFS-VISIT(u)
1 cor[u] ← CINZA          ▷ Branco, o vértice u acabou de ser descoberto.
2 tempo ← tempo + 1
3 d[u] ← tempo
4 for cada v ∈ Adj[u]     ▷ Explora a aresta (u, v).
5     do if cor[u] = BRANCO
6           then π[v] ← u
7                DFS-VISIT(v)
8 cor[u] ← PRETO          ▷ Enegrece u; terminado.
9 f[u] ← tempo ← tempo + 1
```

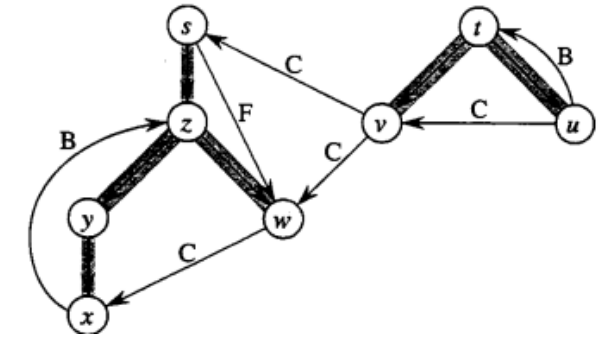
Propriedades busca em profundidade

- **O tempo de descoberta e término tem estrutura de parênteses.** Representando a descoberta de u com um parênteses esquerdo "(u" e seu término por um parênteses direito "u)", a história de descoberta e términos gera uma expressão bem formada.

(b)



(c)

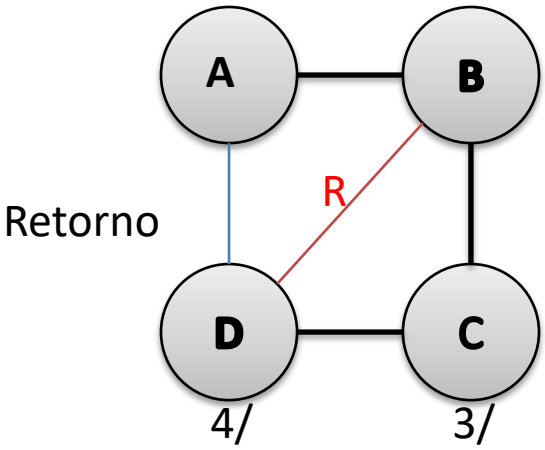


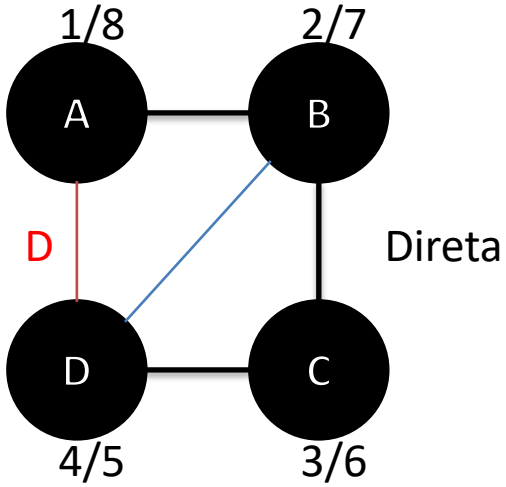
Propriedades busca em profundidade

- A pesquisa pode ser usada para classificar arestas de um grafo, o qual pode ser empregada para reunir informações sobre um grafo.

- Por exemplo, **se um grafo orientado é acíclico**. Nesse caso uma busca em profundidade não produz nenhuma aresta de retorno.

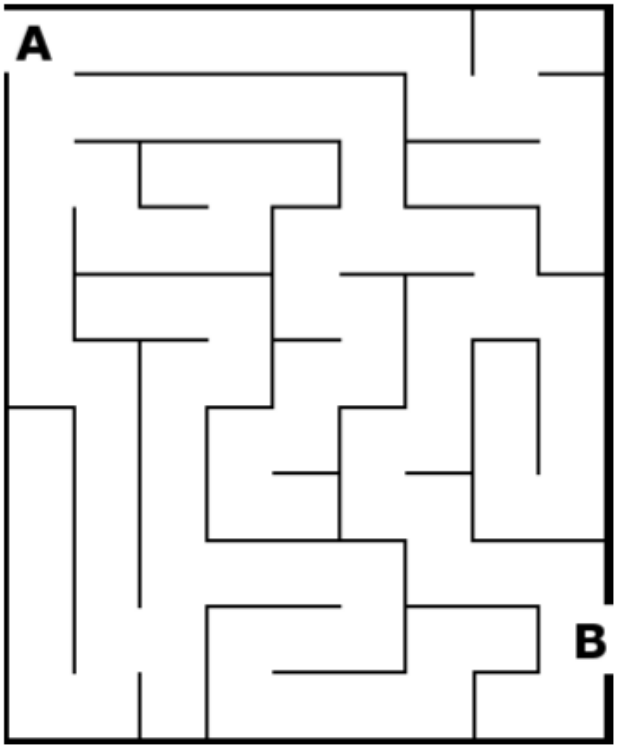
1. Branco indica uma aresta de árvore.
2. Cinza indica uma aresta de retorno.
3. Preto indica uma aresta direta ou cruzada.





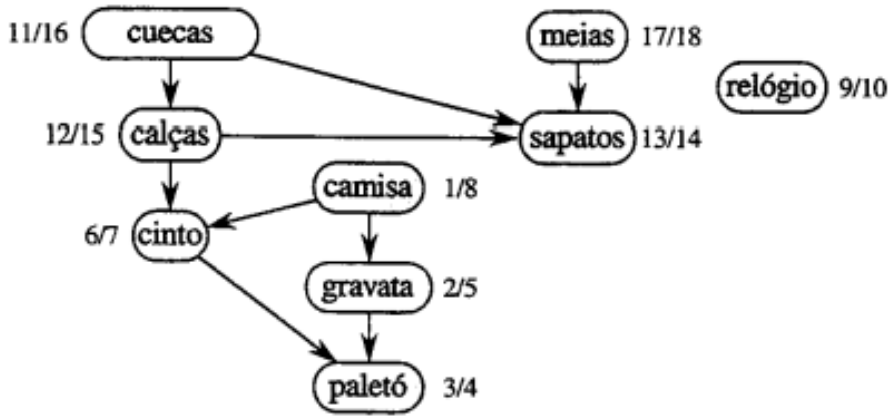
Aplicações

- Percorrer um labirinto

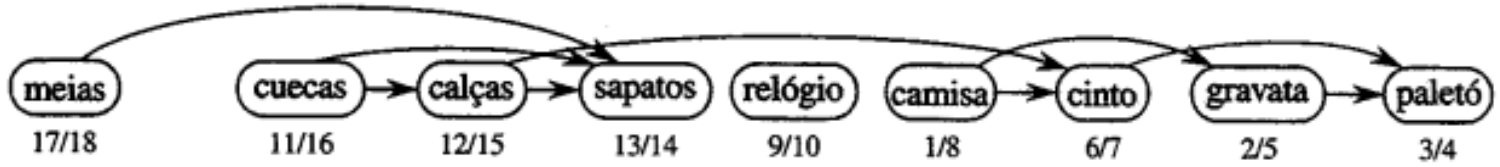


Aplicações

- Ordenação topológica de um grafo acíclico orientado.

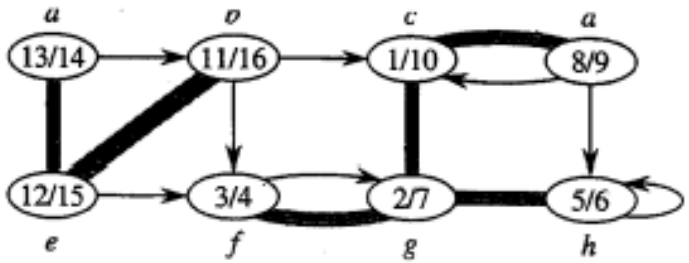


Topological-sort(G)
1 - chamar DFS(G) para calcular o tempo de término f[v] para cada vértice v
2 - à medida que cada vértice é terminado, inserir à frente de uma lista encadeada
3 - retorna a lista

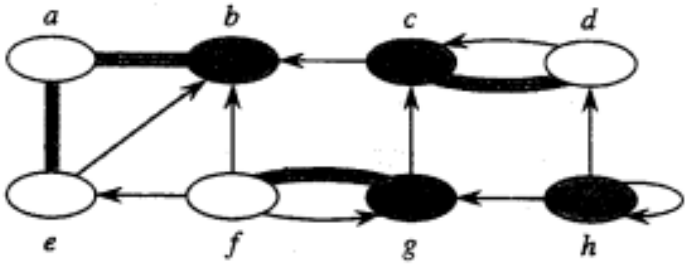


Aplicações

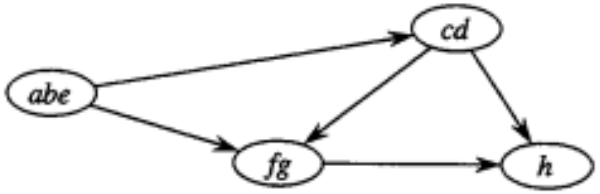
- Componentes fortemente conectados: vértices u e v são acessíveis um a partir do outro.



1 – Chamar DFS(G) para calcular o tempo de término f[u] para cada u.

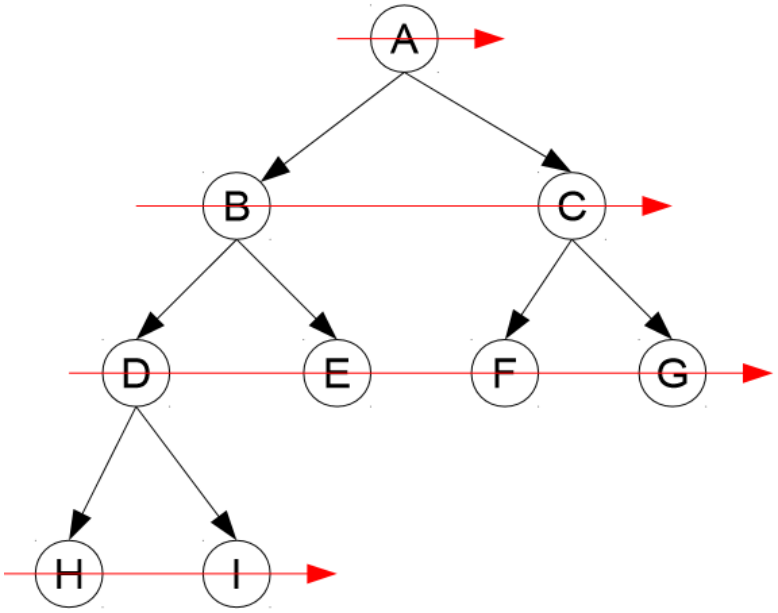


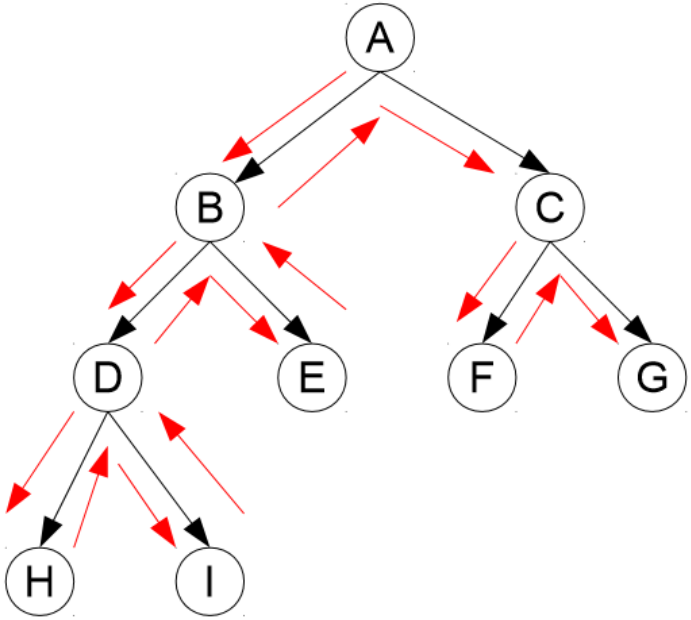
2 – Calcular $G^T = (V, E^T)$
3 – Chamar DFS(GT) e considerar os vértices em ordem decrescente de f[u].



4 - Gera os componentes fortemente conectados.

Busca em largura x busca em profundidade





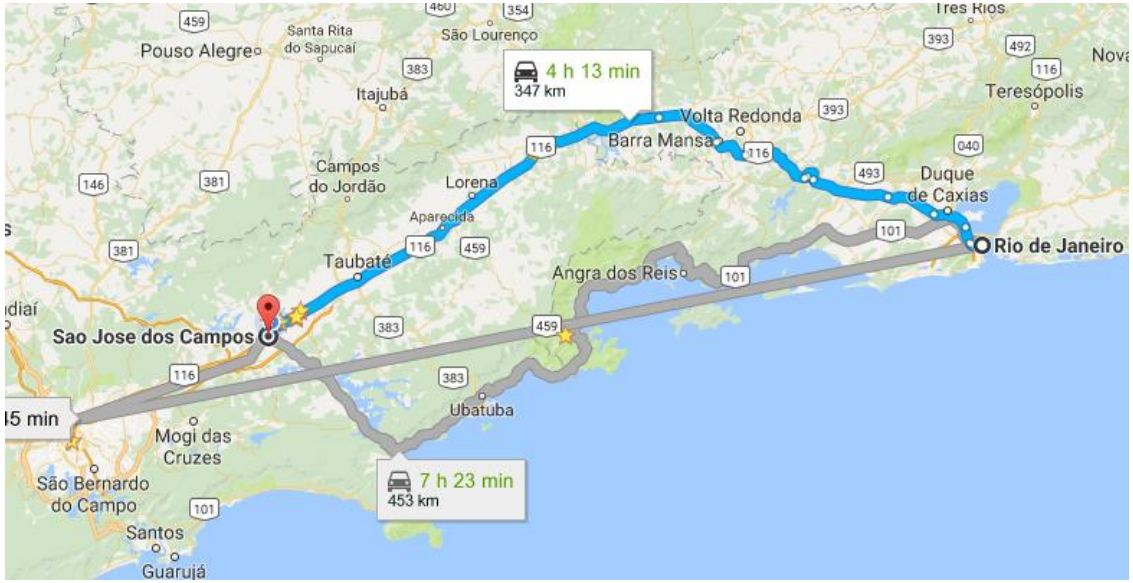
FIFO = First in First out
Complexidade = O (V + E)
Completa = Sim: dado um k limite
Verifica apenas um componente
Grafos não ponderados

LIFO = Last in First out
Complexidade = O (V + E)
Completa = Não: falha em espaços com profundidade infinita
Verifica vários componentes
Grafos não ponderados

O problema do caminho mínimo de única origem

- Se um motorista deseja encontrar a rota mais curta entre São José dos Campos e Rio de Janeiro, como podemos encontrá-la dado um mapa rodoviário do Brasil com as distâncias entre cada cidade?

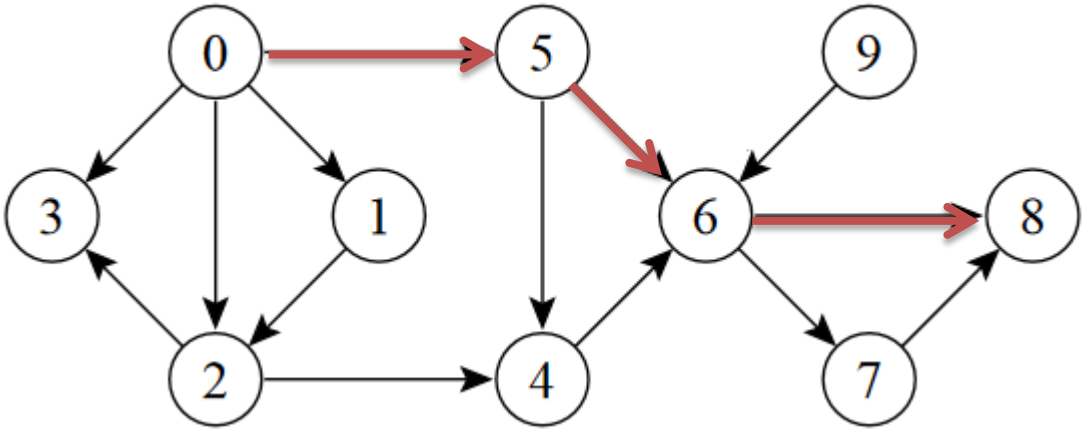
- Podemos modelar o mapa como um grafo: os vértices representam cidades, as arestas estradas e os pesos as distâncias (km).



Introdução: caminho mínimo

- Um caminho de comprimento k de um vértice u até um vértice v em um grafo G é uma sequência s = (v_0, v_1, v_2, ..., v_k) tal que u = v_0 e v = v_k, e (v_{i-1},v_i) ∈ E para i = 1,2, ..., k.

- **Dado um grafo G = (V,E) determine o caminho mínimo (distância) entre dois pares de vértices em G.**



Introdução: caminho mínimo

- **Encontrar o menor caminho de um vértice origem u para todos os vértices que são alcançáveis a partir de u, quando todos os arcos têm comprimento não negativos**.

- O peso do caminho $s = (v_0, v_1, v_2, ..., v_k)$ é o somatório dos pesos de suas arestas constituintes:

$$d(s) = \sum_{i=1}^{k} d(v_{i-1}, v_i)$$

- O peso do caminho mais curto de u até v é dado por:

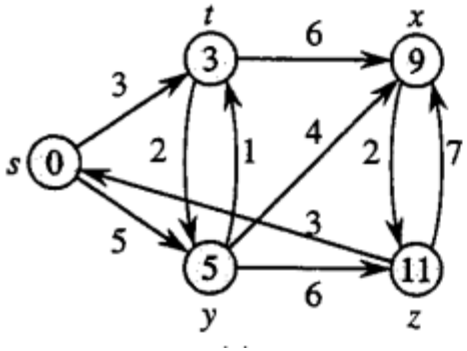
$$\delta(u,v) = \min d(s): u \rightarrow v$$

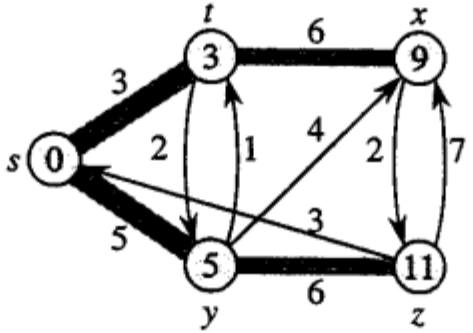
- **Um caminho mínimo em um grafo não pode conter ciclo**, pois a remoção do ciclo produz um caminho com os mesmos vértices origem e destino e um caminho de menor peso.
  - Em um grafo um caminho $(v_0; v_1;...; v_k)$ forma um ciclo se $v_0 = v_k$.

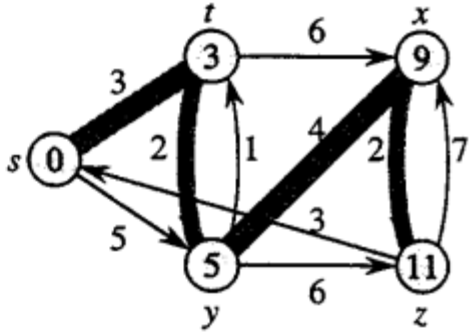
Árvore de caminhos mínimos

- Uma árvore de caminhos mais curtos com raiz em s é um subgrafo G'=(V',E') onde V' C V e E' C V:
  1. V' é o conjunto de vértices acessíveis a partir de s em G.
  2. G' forma uma árvore enraizada com raiz s
  3. Para todo v ∈ V', o único caminho simples de s até v em G' é o caminho mais curto desde s até v em G.

- Caminhos mais curtos não são necessariamente únicos.



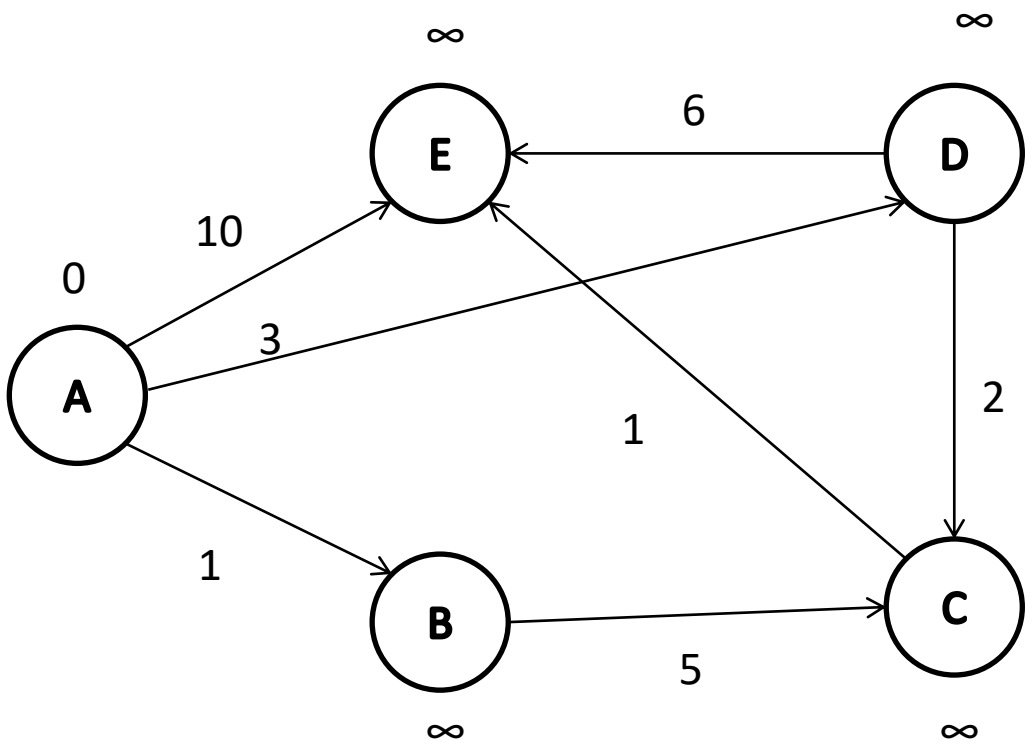




Exemplo Dijkstra

- Caminho mínimo do vértice A ao vértice E.
- 1ª iteração: S = { }

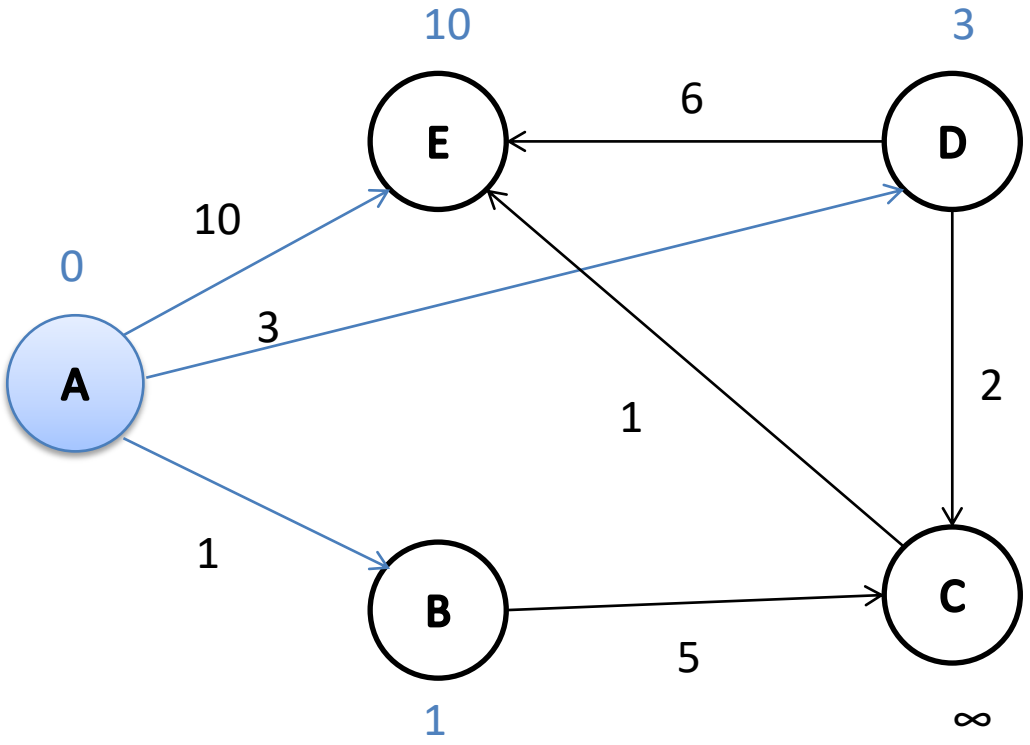
	A	B	C	D	E
1ª	0	∞	∞	∞	∞



Exemplo Dijkstra

- Caminho mínimo do vértice A ao vértice E.
- 2ª iteração: S = {A}

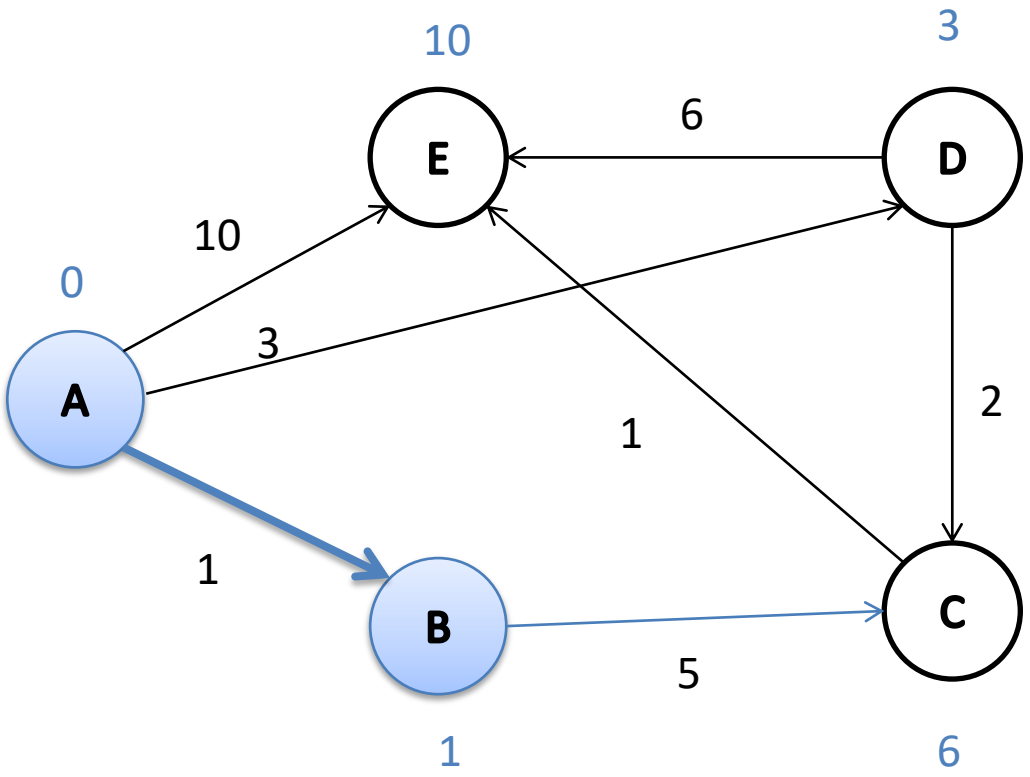
S	A	B	C	D	E
1ª	∞	∞	∞	∞	∞
2ª	0	1	∞	3	10



Exemplo Dijkstra

- Caminho mínimo do vértice A ao vértice E.
- 3ª iteração: S = {A,B}

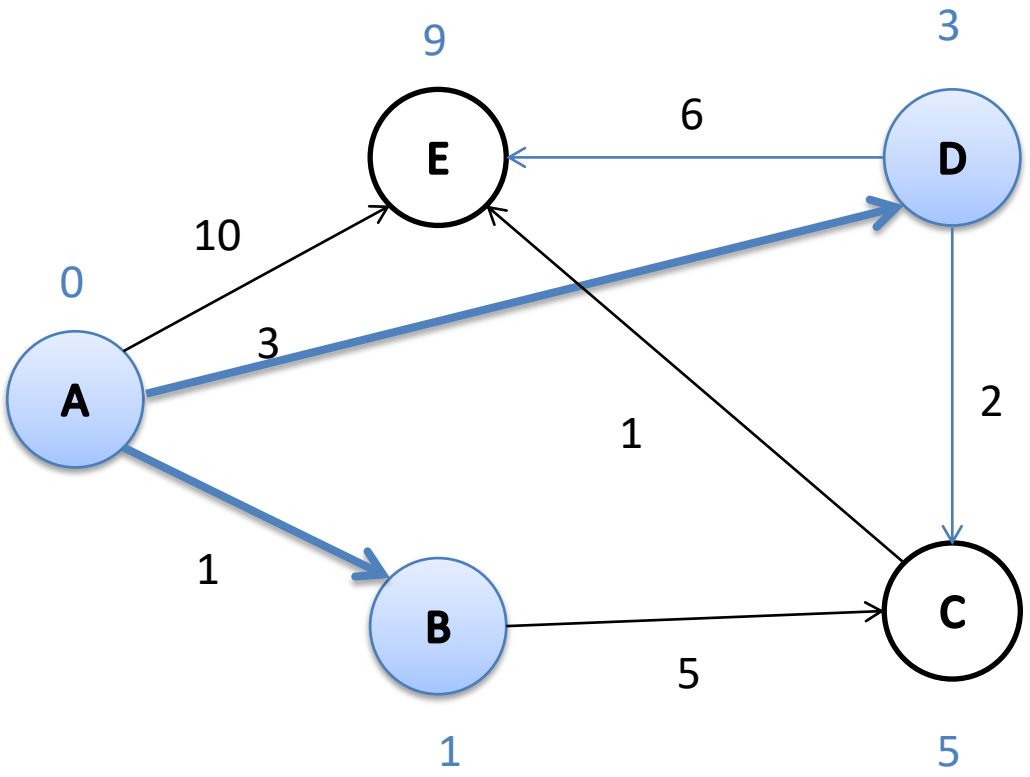
S	A	B	C	D	E
1ª	∞	∞	∞	∞	∞
2ª	0	1	∞	3	10
3ª	0	1	6	3	10



Exemplo Dijkstra

- Caminho mínimo do vértice A ao vértice E.
- 4ª iteração: S = {A,B,D}

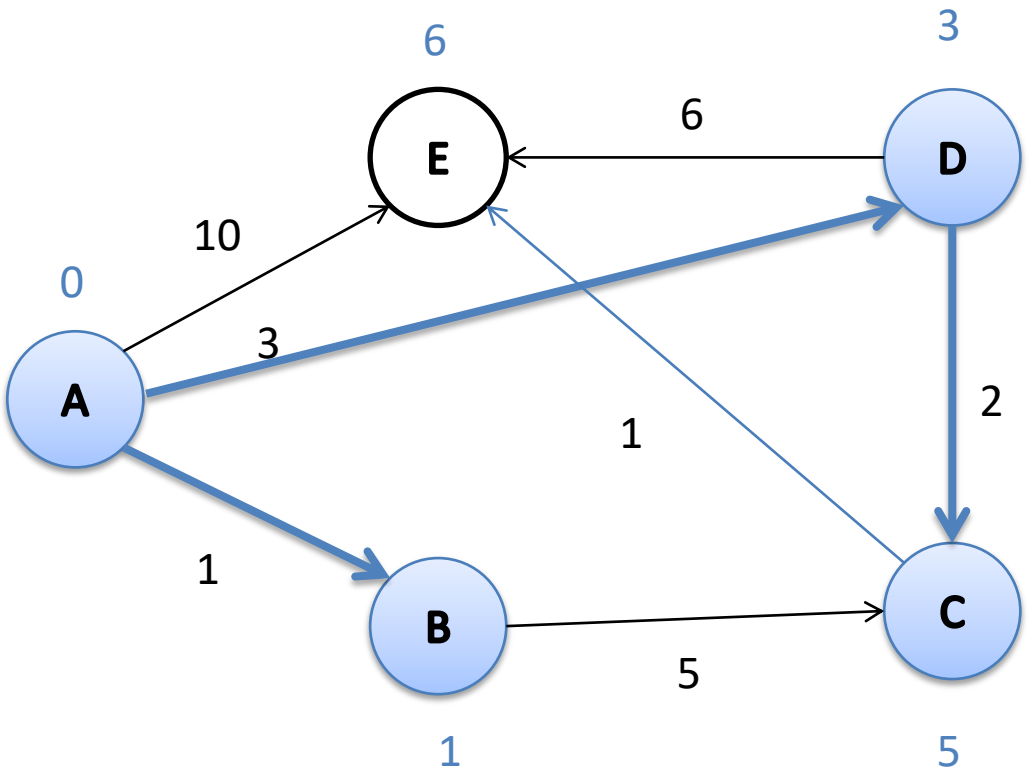
S	A	B	C	D	E
1ª	∞	∞	∞	∞	∞
2ª	0	1	∞	3	10
3ª	0	1	6	3	10
4ª	0	1	5	3	9



Exemplo Dijkstra

- Caminho mínimo do vértice A ao vértice E.
- 4ª iteração: S = {A,B,D,C}

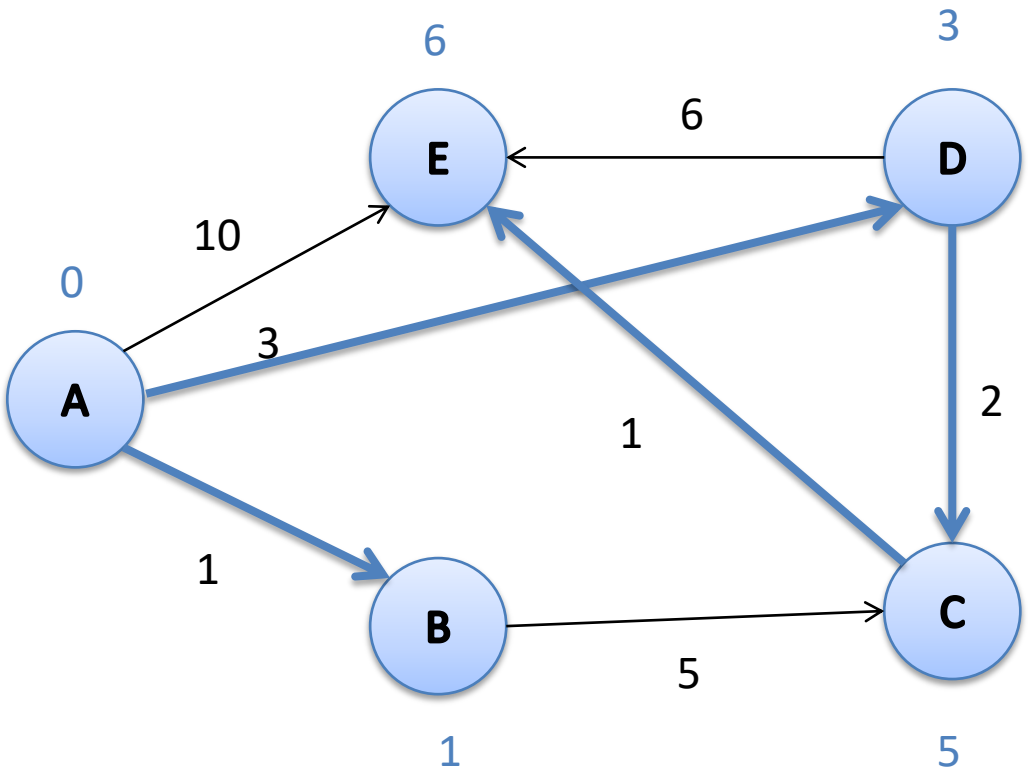
S	A	B	C	D	E
1ª	∞	∞	∞	∞	∞
2ª	0	1	∞	3	10
3ª	0	1	6	3	10
4ª	0	1	5	3	9
5ª	0	1	5	3	6



Exemplo Dijkstra

- Caminho mínimo do vértice A ao vértice E.
- 4ª iteração: S = {A,B,D,C,E}

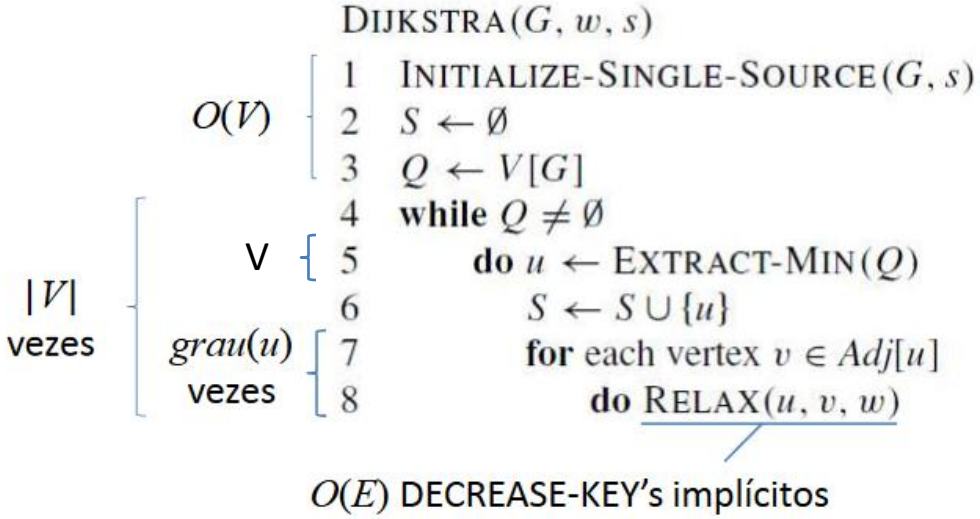
S	A	B	C	D	E
1ª	∞	∞	∞	∞	∞
2ª	0	1	∞	3	10
3ª	0	1	6	3	10
4ª	0	1	5	3	9
5ª	0	1	5	3	6
6ª	0	1	5	3	6



Menor custo A -> E= 6

Complexidade

- Insert (linha 3) e extract-min (linha 5) são invocados uma vez por vértice.
- Cada aresta Adj[v] é examinada no loop for das linhas 7 e 8 uma vez . Como o total de arestas em todas as listas de adjacências é |E|, o total de iterações desse loop será |E|.
- Decrease-key executa |E| vezes.
- No total são O (V^2 + E) = O (V^2).



Algoritmo de Dijkstra

- Utiliza **a técnica de relaxamento**:

  - Para cada vértice u o atributo d[u] é um limite superior do peso de um caminho mais curto do vértice origem até u.

  - O relaxamento de uma aresta (u,v) consiste em verificar se é possível melhorar o caminho até v obtido até o momento se passarmos por u.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ V[G]
2       do d[v] ← ∞
3          π[v] ← NIL
4  d[s] ← 0
```

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          π[v] ← u
```

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5       do u ← EXTRACT-MIN(Q)
6          S ← S ∪ {u}
7          for each vertex v ∈ Adj[u]
8               do RELAX(u, v, w)
```


Ciência da Computação está tão relacionada aos computadores quanto a Astronomia aos telescópios, Biologia aos microscópios, ou Química aos tubos de ensaio. A Ciência não estuda ferramentas. Ela estuda como nós as utilizamos, e o que descobrimos com elas.
(Edsger Dijkstra)