Programação 2 _ T12

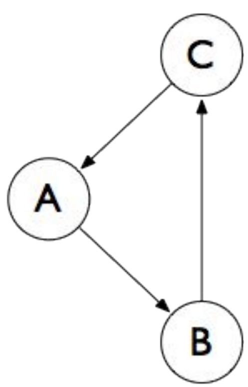
Algoritmos em grafos

(caminho mais curto)

Rui Camacho
(slides por Luís Teixeira)
MIEEC 2020/2021

RELEMBRAR

- Um grafo $G = \langle V, E \rangle$ consiste:
  - ... num conjunto finito de vértices V (*vertex*, *vertices*)
  - ... num conjunto finito de arestas E (*edge, edges*)
  - ... cada aresta é um par (a, b) tal que a, b $\in$ V



$V = \{A, B, C\}$

$E = \{(A, B), (B, C), (C, A)\}$

CAMINHO MAIS CURTO EM GRAFOS

Objetivo: Dado um grafo, qual é o caminho mais curto entre dois vértices?

Motivações

- Muitos problemas podem ser representados por este tipo de grafos
- Os vértices podem representar os possíveis estados, e as arestas entre eles representam o custo de passar de um estado para outro (distância, preço, etc)

Aplicações

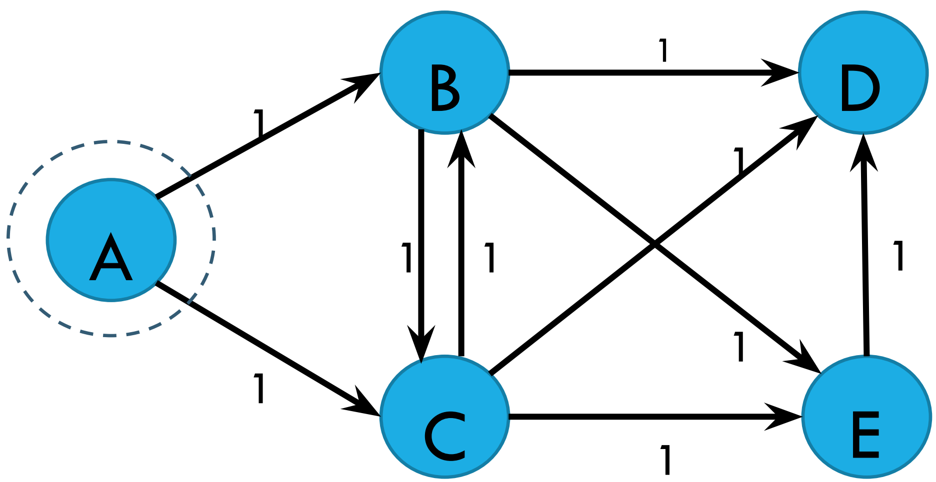
- Encaminhamento de pacotes de dados em redes de computadores entre dois dispositivos
- Sistemas de navegação

ARESTAS UNITÁRIAS

Começando em A, qual é o caminho mais curto para os outros vértices?

B: [A, B] D: [A, B, D] or [A, C, D]

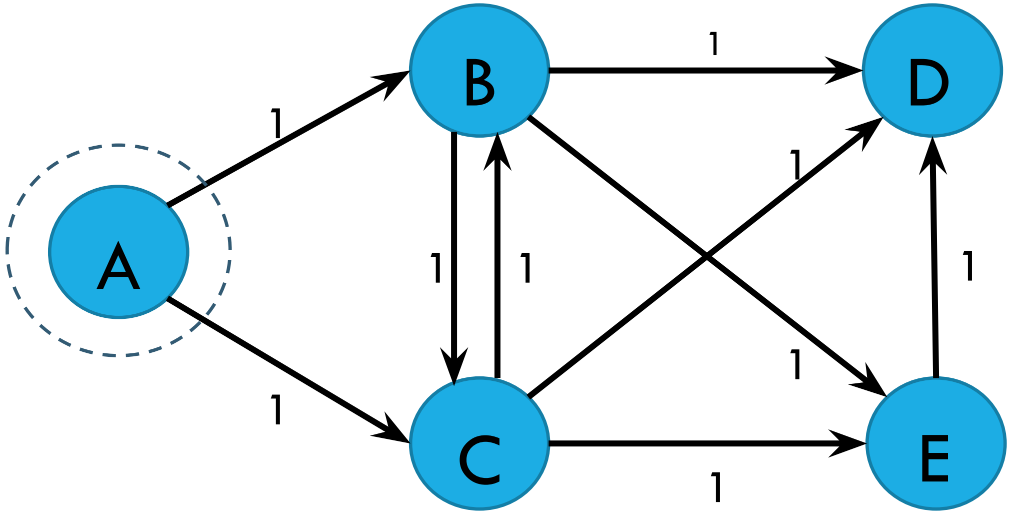
C: [A, C] E: [A, B, E] or [A, C, E]



PESQUISA EM LARGURA

A pesquisa em largura alcança o vértice alvo no menor número de passos possível

Qual é o caminho mais curto entre A e E?

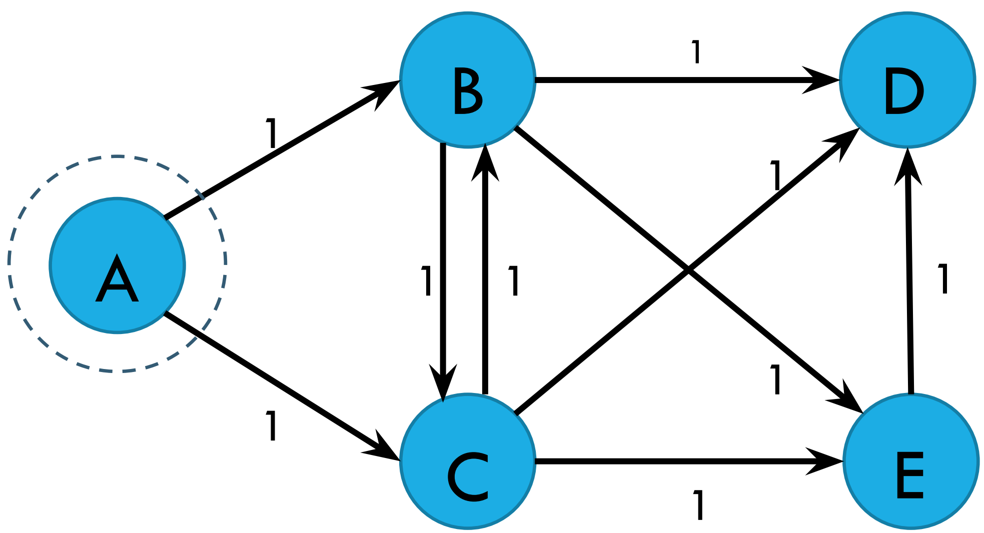


EXEMPLO DE PESQUISA EM LARGURA

Pesquisa em largura usa uma fila auxiliar

O vértice inicial é colocado na fila, e em cada vértice é registado o vértice anterior

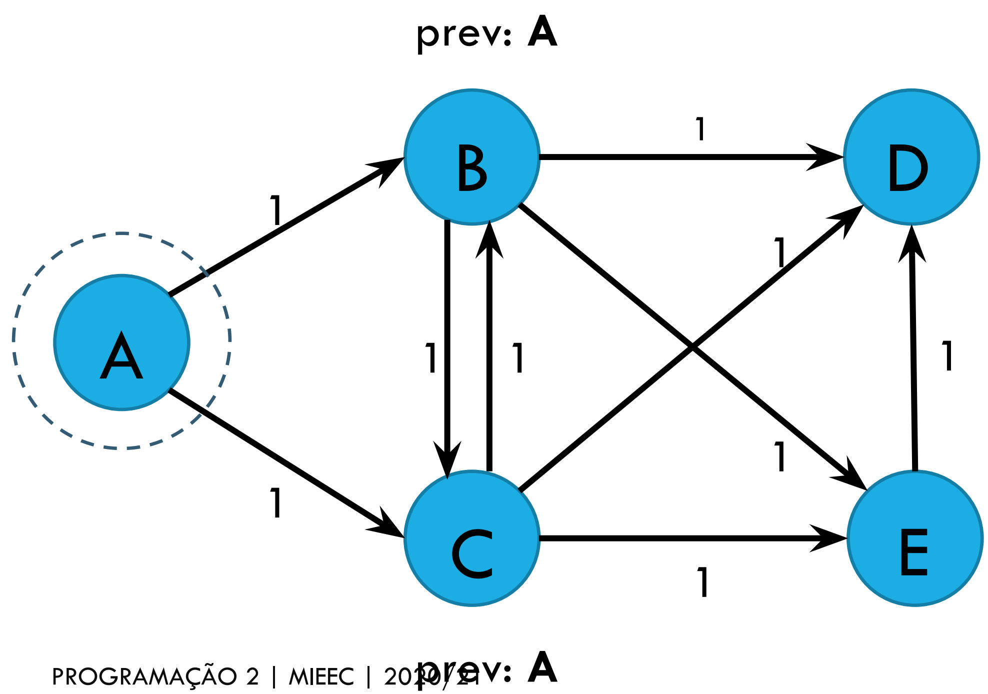
Fila
A



EXEMPLO DE PESQUISA EM LARGURA

Vértice A é retirado da fila, e em todos os seus vizinhos é registado A como vértice anterior; são ainda colocados na fila.

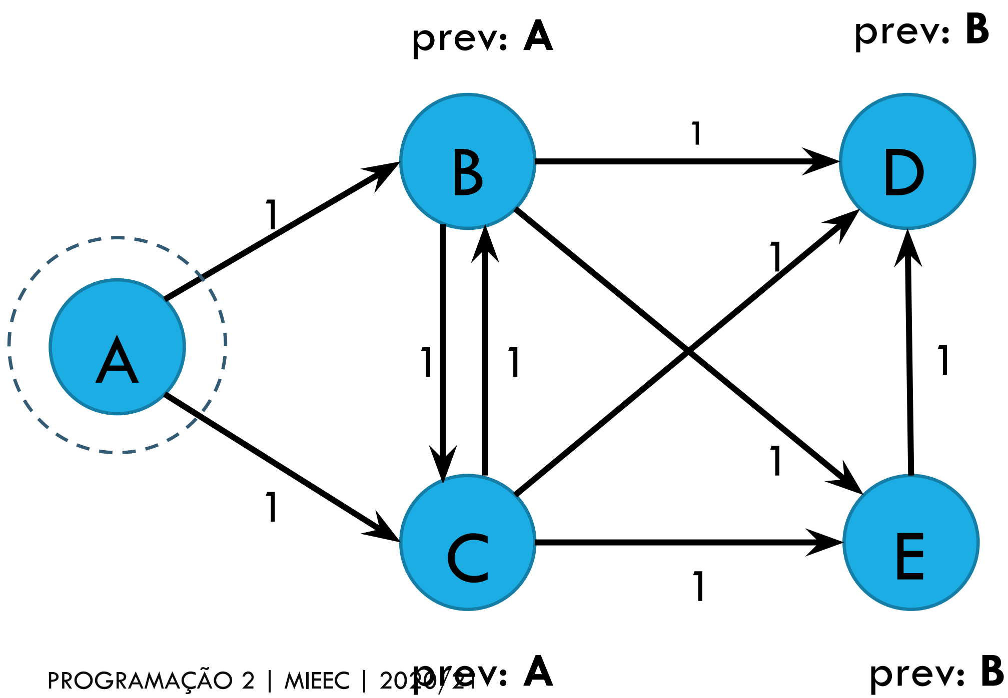
Fila
B
C



EXEMPLO DE PESQUISA EM LARGURA

Vértice B é retirado da fila, e os passos anteriores são repetidos, ignorando os vizinhos que já tenham registado um vértice anterior

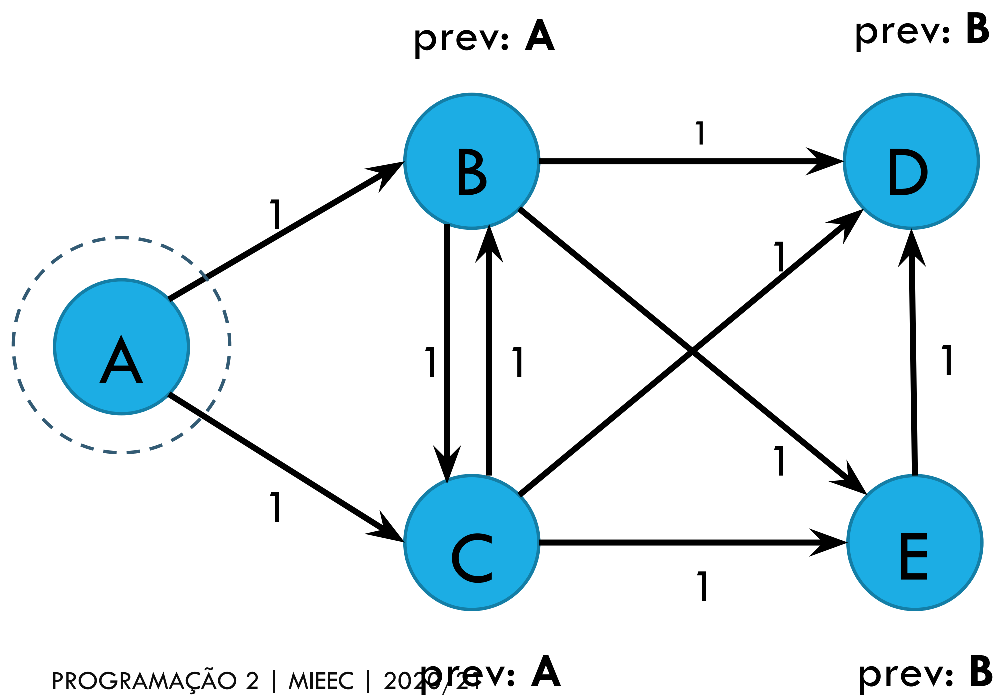
Fila
C
D
E



EXEMPLO DE PESQUISA EM LARGURA

Retirar os vértices C e D da fila não tem qualquer efeito, uma vez que todos os seus vizinhos já foram visitados

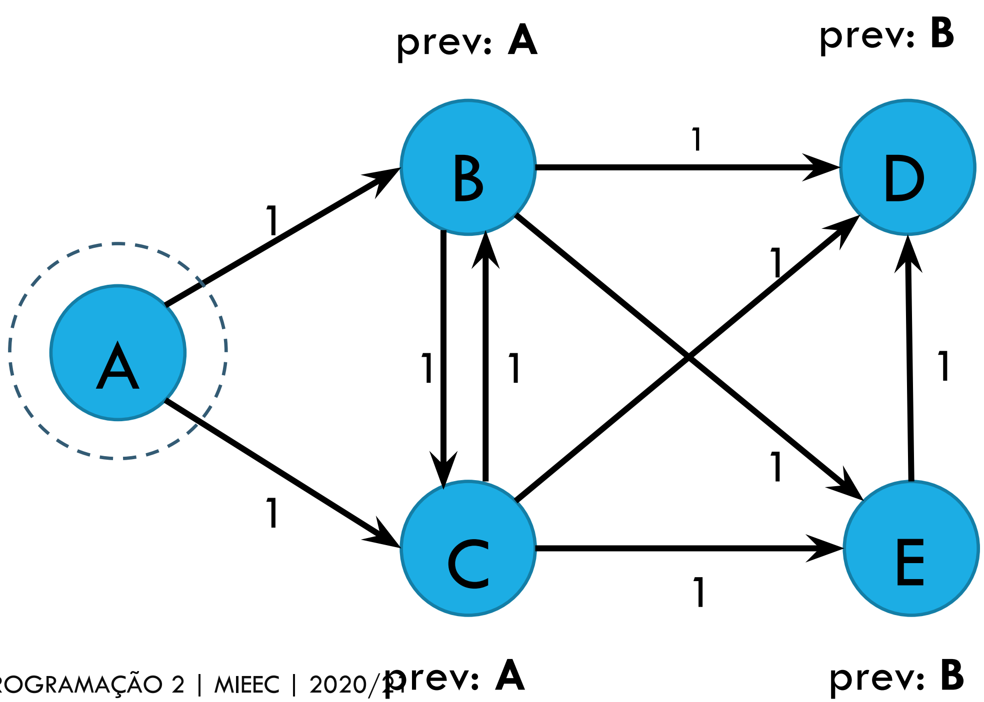
Fila
E



EXEMPLO DE PESQUISA EM LARGURA

Quando finalmente o vértice E é retirado da fila, é possível recorrer à informação dos vértices anteriores para determinar o caminho mais curto

Vértice objetivo	Caminho mais curto
B	[A, B]
C	[A, C]
D	[A, B, D]
E	[A, B, E]



PESQUISA EM LARGURA - COMPLEXIDADE

Uma vez que no pior dos casos cada vértice e aresta são verificados, a complexidade temporal pode ser expressa por:

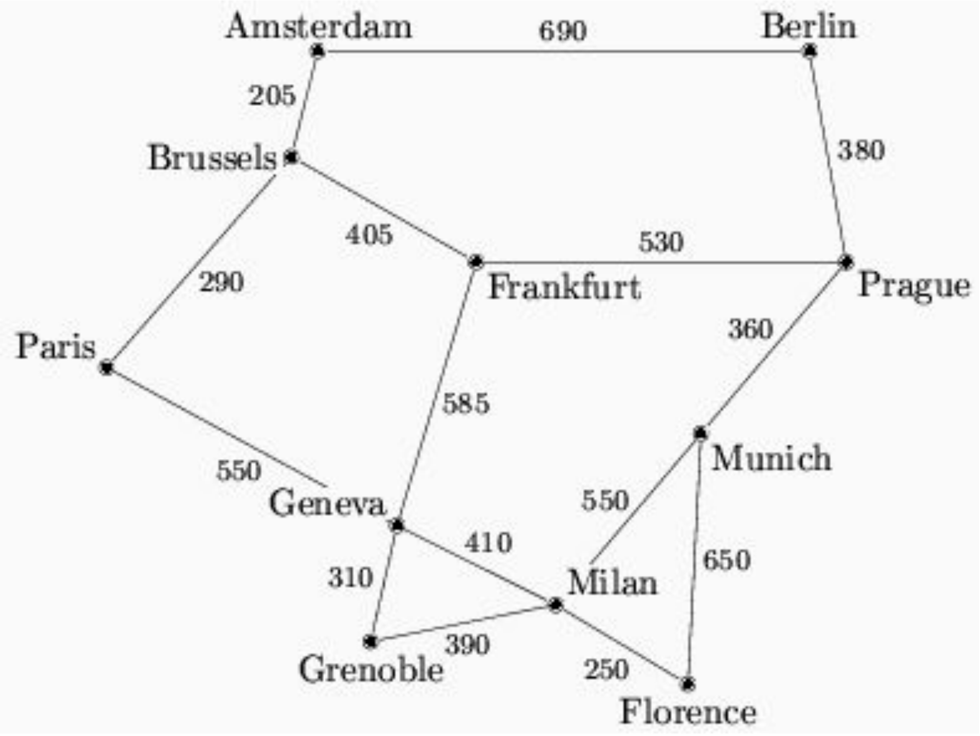
$$O(\mathrm{V}+\mathrm{E})$$

onde V é o número de vértices e E é o número de arestas

Ou seja, a complexidade será dominada pelo maior dos dois números. Por exemplo, um grafo denso (no limite, um grafo completo) terá E > V; por outro lado, um grafo esparso terá V > E.

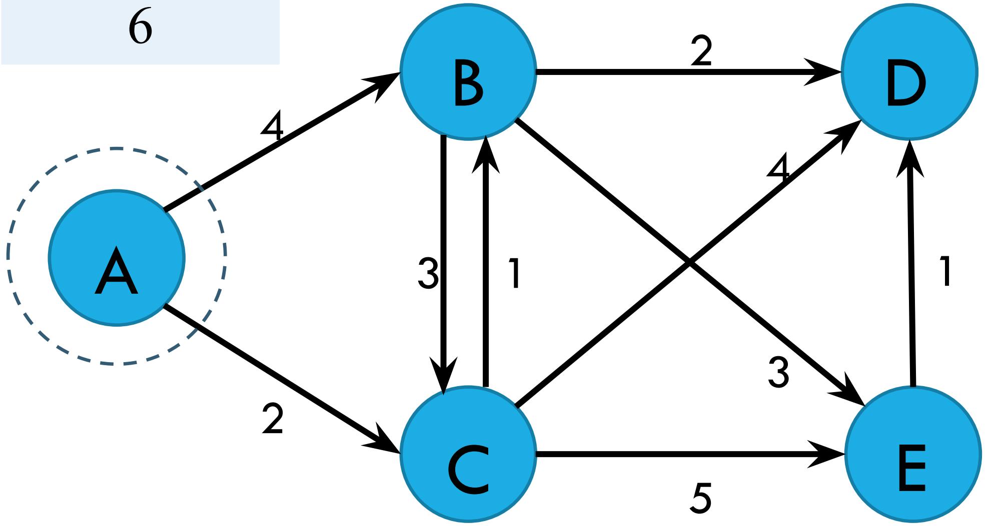
ARESTAS NÃO UNITÁRIAS

E se os pesos das arestas não forem unitários?



ARESTAS NÃO UNITÁRIAS

Vértice objetivo	Caminho mais curto	Distância mais curta
B	[A, C, B]	3
C	[A, C]	2
D	[A, C, B, D]	5
E	[A, C, B, E]	6



ENCONTRAR O CAMINHO MAIS CURTO

Pesquisa em largura funciona quando todas as arestas têm o mesmo peso

- Vértices visitado por ordem de distância total do início

E quando as arestas têm pesos diferentes?

Qual o algoritmo para percorrer os vértices por ordem de distância total do início?

- Vamos recorrer a uma estrutura de dados conhecida .. Qual?

ENCONTRAR O CAMINHO MAIS CURTO (2)

Recorremos a uma fila de prioridade (**priority queue**)

- A prioridade é a distância total do início ao vértice
- Ao visitar os vértices pela ordem que são devolvidos pela função `findMin()`, visitamos os vértices por ordem de distância total do início

Como não exploramos um vértice antes de explorar todos os outros que estão mais próximos do início, garantimos que o caminho mais curto é encontrado

ALGORITMO DE DIJKSTRA

Marcar todos os vértices com valor ∞ (distância), exceto o vértice inicial que tem distância 0

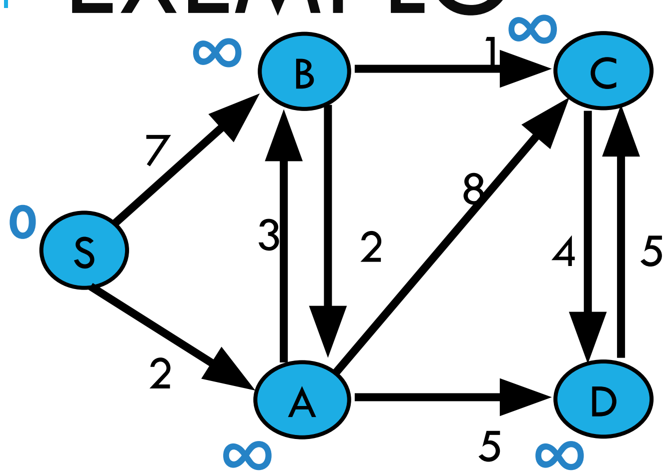
Inserir todos os vértices na fila de prioridade, utilizando a distância como prioridade

Enquanto a fila de prioridade não estiver vazia:

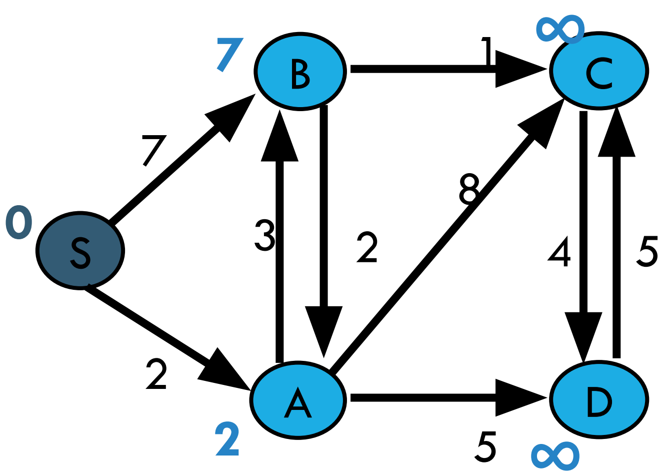
- Remover o vértice com menor prioridade
- Atualizar as distâncias dos vértices vizinhos do que acabou de ser removido, caso a distância tenha sido reduzida

Quanto termina (fila de prioridade vazia) todos os vértices têm atribuído a distância (custo) mínima desde o nó inicial

ALGORITMO DE DIJKSTRA: EXEMPLO

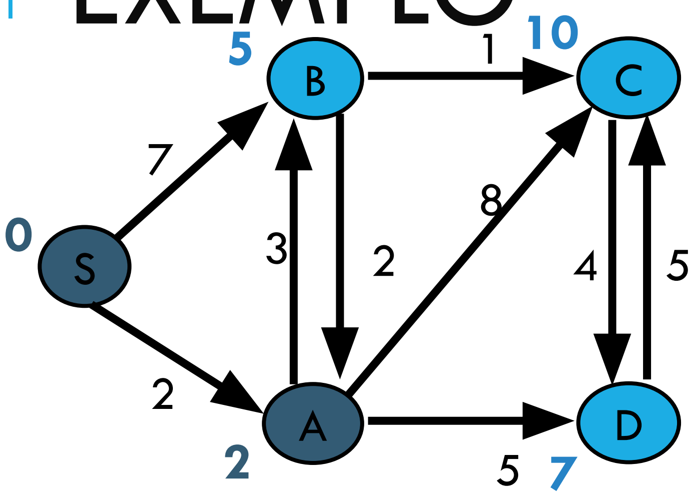


Passo 1: Vértice inicial marcado com a distância 0 e os restantes com distância infinita. Inserir todos os vértices na fila de prioridade.

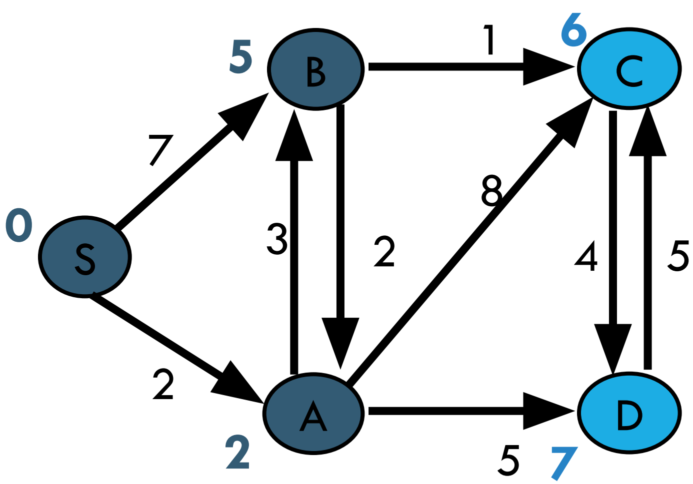


Passo 2: Remover da fila de prioridade o vértice com prioridade mínima (*S* neste exemplo). Calcular a distância desde o início até ao vértices vizinhos do que acabou de ser removido somando ao peso do arco a distância de S.

ALGORITMO DE DIJKSTRA: EXEMPLO

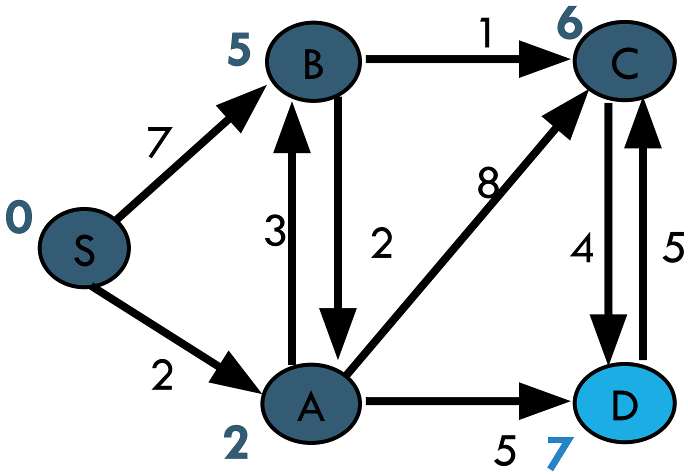


Passo 3: Repetir o passo anterior (repete enquanto a fila de prioridade não estiver vazia) removendo *A*. As prioridades dos vértices na fila de prioridade podem ter de ser atualizadas, como é o caso da prioridade de *B* neste exemplo.

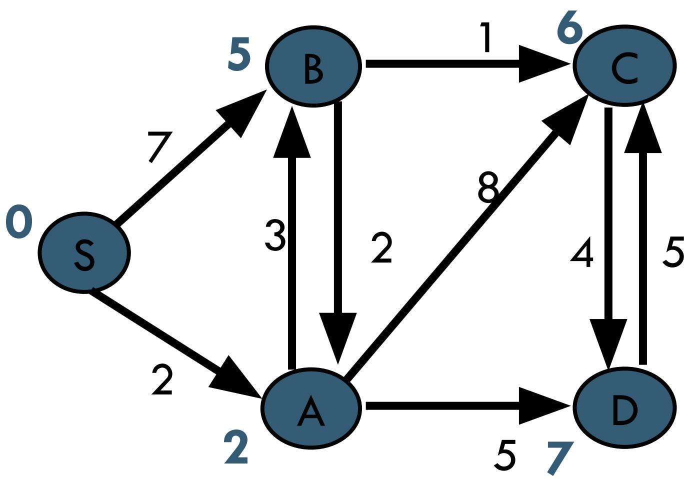


Passo 4: Repetir removendo vértice *B*. Voltar a atualizar as distâncias (prioridades) que se reduzem ao utilizar este novo caminho (por exemplo, C passa a ter distância 6 e não 10).

ALGORITMO DE DIJKSTRA: EXEMPLO

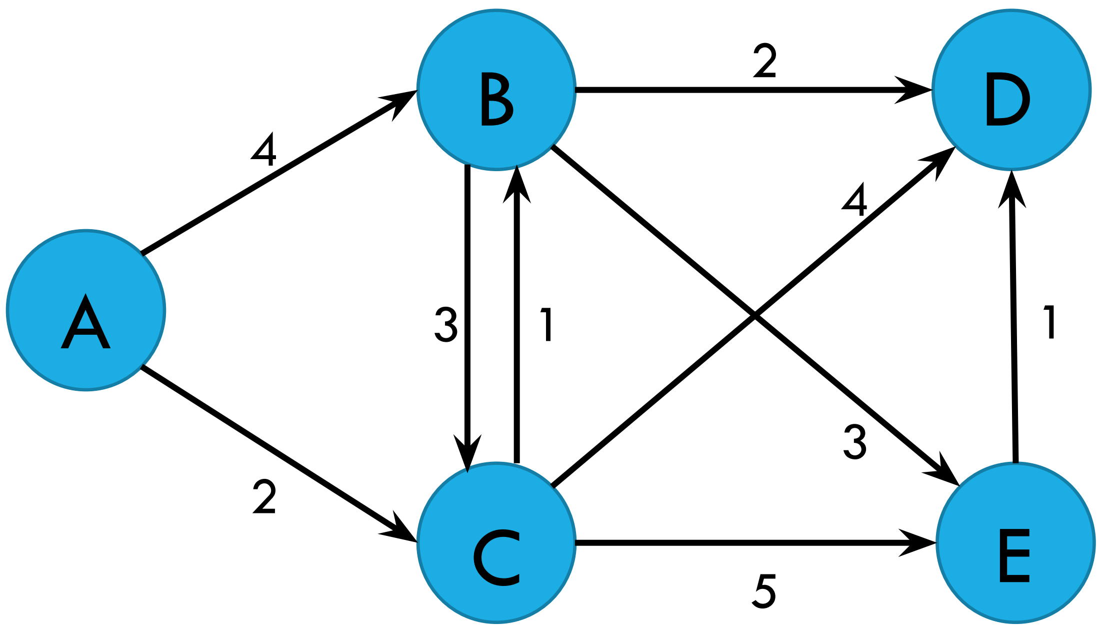


Passo 5: Repetir, removendo C.



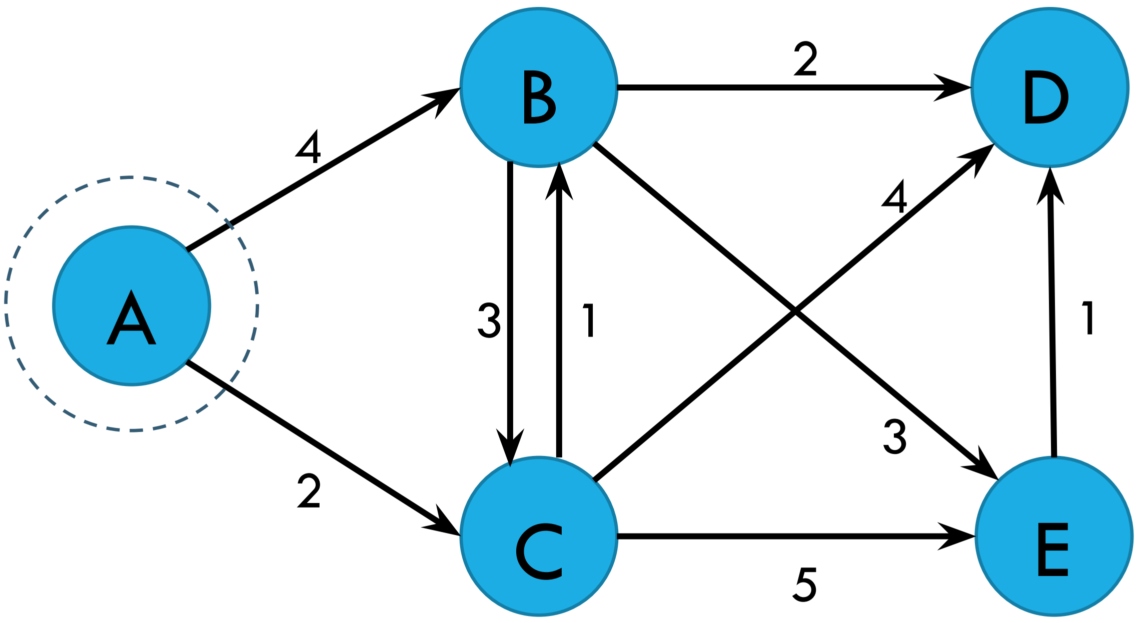
Passo 6: Após remover *D*, todos os vértices foram visitados e marcados com a distância mais curta desde o início.

ALGORITMO DE DIJKSTRA: EXEMPLO 2



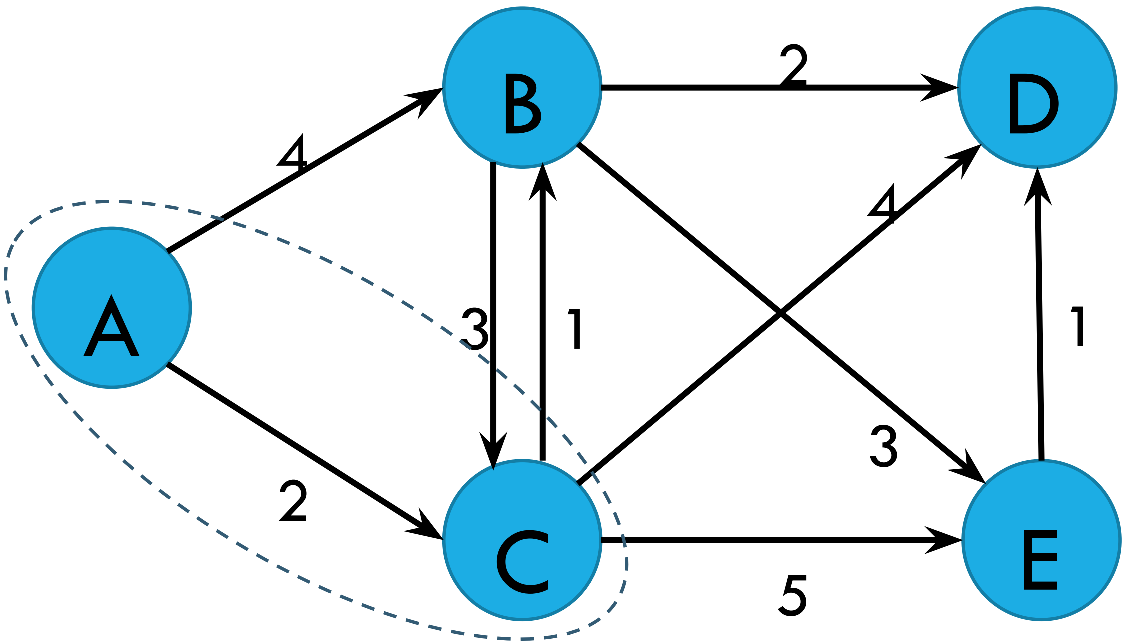
A	B	C	D	E
0	∞	∞	∞	∞

ALGORITMO DE DIJKSTRA: EXEMPLO 2



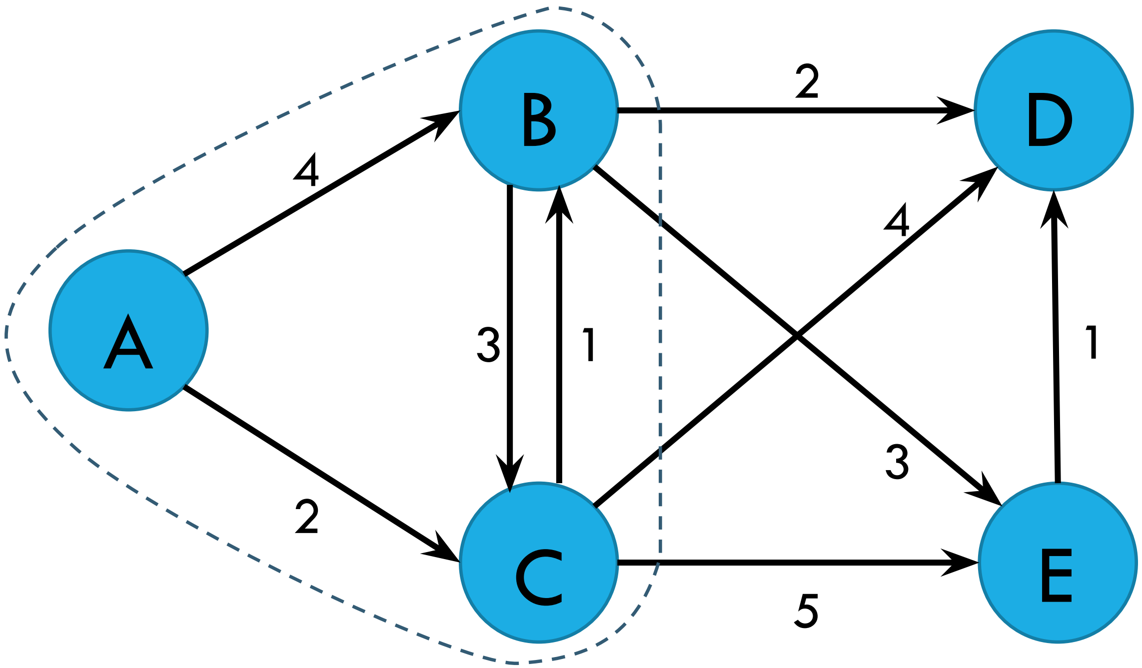
A	B	C	D	E
0	4	2	∞	∞

ALGORITMO DE DIJKSTRA: EXEMPLO 2



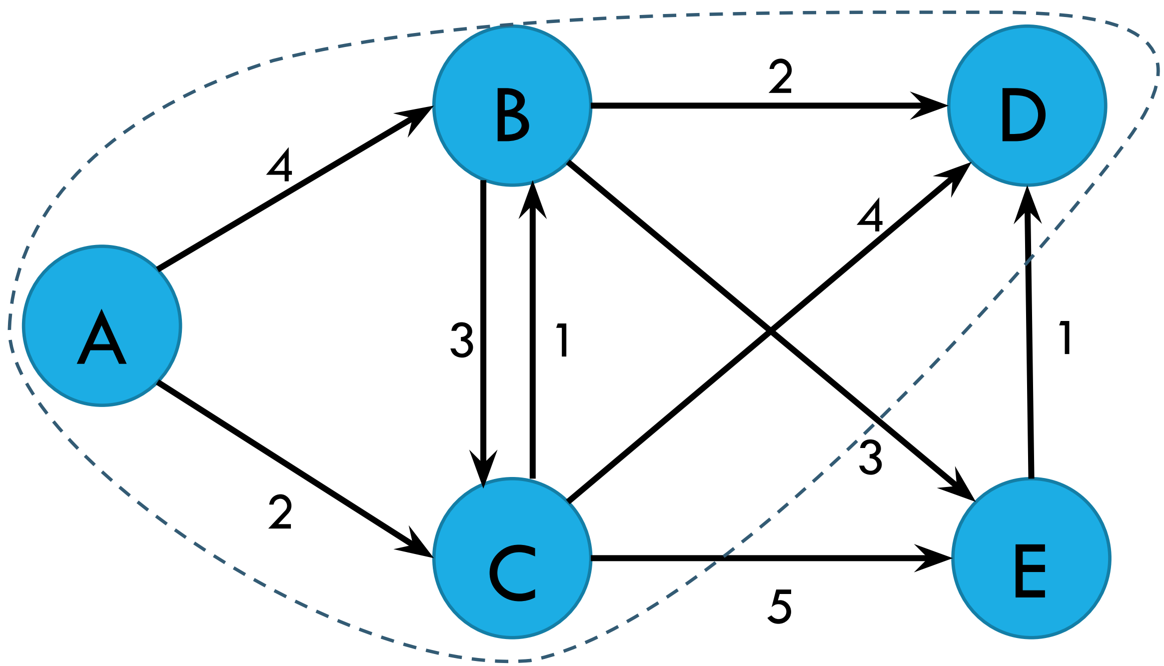
A	B	C	D	E
0	3	2	6	7

ALGORITMO DE DIJKSTRA: EXEMPLO 2



A	B	C	D	E
0	3	2	**5**	**6**

ALGORITMO DE DIJKSTRA: EXEMPLO 2



A	B	C	D	E
0	3	2	5	6

ALGORITMO DE DIJKSTRA: PSEUDO CÓDIGO

```
function dijkstra(G, s):
   // Input: A graph G with vertices V, and a start vertex s
   // Output: Nothing
   // Purpose: Decorate nodes with shortest distance from s
   for v in V:
    v.dist = infinity  // Initialize distance decorations
    v.prev = null      // Initialize previous pointers to null
   s.dist = 0             // Set distance to start to 0

   PQ = PriorityQueue(V)    // Use v.dist as priorities
   while PQ not empty:
      u = PQ.removeMin()
      for all edges (u, v):
         if v.dist > u.dist + cost(u, v): // cost() is weight
            v.dist = u.dist + cost(u,v)   // Replace as necessary
            v.prev = u      // Maintain pointers for path
            PQ.replaceKey(v, v.dist)
```

ALGORITMO DE DIJKSTRA: COMPLEXIDADE

```
function dijkstra(G, s):
   // Input: A graph G with vertices V, and a start vertex s
   // Output: Nothing
   // Purpose: Decorate nodes with shortest distance from s
   for v in V:                    // O(V)
    v.dist = infinity
    v.prev = null
   s.dist = 0

   PQ = PriorityQueue(V)
   while PQ not empty:          // O(V)
      u = PQ.removeMin()        // Depends on PQ implementation!
      for all edges (u, v):     // O(E)
         if v.dist > u.dist + cost(u, v):
            v.dist = u.dist + cost(u,v)
            v.prev = u
            PQ.replaceKey(v, v.dist) // Depends on PQ implementation!
```

ALGORITMO DE DIJKSTRA: COMPLEXIDADE

Depende da implementação da fila de prioridade

Vetor ou Lista ligada

- `removeMin()` $O(\mathrm{V})$ (necessário percorrer para determinar o mínimo)
- `replaceKey()` $O(1)$ (já temos o vértice quando alteramos a chave)
- Tempo de execução é $O(V^2 + E) \rightarrow \boldsymbol{O(N^2)}$

Heap binário

- `removeMin()` $O(logV)$
- `replaceKey()` $O(logV)$ (pode ser necessário downheap)
- Tempo de execução é $\boldsymbol{O(\mathrm{V}\ log\ V + \mathrm{E}\ log\ V)} \rightarrow \boldsymbol{O(N\ log\ N)}$

DIJKSTRA COMPLEXIDADE – VETOR/LISTA LIGADA

```
function dijkstra(G, s):
   // Input: A graph G with vertices V, and a start vertex s
   // Output: Nothing
   // Purpose: Decorate nodes with shortest distance from s
   for v in V:                  // O(V)
    v.dist = infinity
    v.prev = null
   s.dist = 0

   PQ = PriorityQueue(V)
   while PQ not empty:         // O(V)
      u = PQ.removeMin()       // O(V)
      for all edges (u, v):    // O(E)
         if v.dist > u.dist + cost(u, v):
            v.dist = u.dist + cost(u,v)
            v.prev = u
            PQ.replaceKey(v, v.dist) // O(1)
```

DIJKSTRA COMPLEXIDADE – HEAP BINÁRIO

```
function dijkstra(G, s):
   // Input: A graph G with vertices V, and a start vertex s
   // Output: Nothing
   // Purpose: Decorate nodes with shortest distance from s
   for v in V:                    // O(V)
    v.dist = infinity
    v.prev = null
   s.dist = 0

   PQ = PriorityQueue(V)
   while PQ not empty:           // O(V)
      u = PQ.removeMin()         // O(log(V))
      for all edges (u, v):      // O(E)
         if v.dist > u.dist + cost(u, v):
            v.dist = u.dist + cost(u,v)
            v.prev = u
            PQ.replaceKey(v, v.dist) // O(log(V))
```