

Algoritmos elementares em Grafos

Ana Paula Tomás

Desenho e Análise de Algoritmos 2018/19

Outubro 2018

Problema: Obter os nós acessíveis de s num grafo G

Estratégia: **pesquisa em largura** a partir do nó s em $G = (V, A)$

BFS_Visit(s, G) // **Breadth-First Search**

```

Para cada  $v \in G.V$  fazer
     $visitado[v] \leftarrow \text{false};$ 
     $pai[v] \leftarrow \text{NULL};$ 
 $visitado[s] \leftarrow \text{true};$ 
 $Q \leftarrow \text{MKEMPTYQUEUE}();$ 
 $\text{ENQUEUE}(s, Q);$ 
Repita
     $v \leftarrow \text{DEQUEUE}(Q);$ 
    Para cada  $w \in G.Adjs[v]$  fazer
        Se  $visitado[w] = \text{false}$  então
             $\text{ENQUEUE}(w, Q);$ 
             $visitado[w] \leftarrow \text{true};$ 
             $pai[w] \leftarrow v;$       //  $v$  precede  $w$  no caminho de  $s$  para  $w$ 
até ( $\text{QUEUEISEMPTY}(Q) = \text{true}$ );
  
```

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $\text{pai}[\cdot]$ define uma árvore com raiz s , que se chama **árvore de pesquisa em largura a partir de s** . Os ramos da árvore são os pares $(\text{pai}[v], v)$ tais que $\text{pai}[v] \neq \text{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, **mínimo** significa que tem o **menor número de ramos possível**). Os nós são visitados por **ordem crescente de distância a s** , nesse sentido.
- Se G for **não dirigido**, os vértices visitados na chamada $\text{BFS_VISIT}(s, G)$ são os que definem a **componente conexa** a que s pertence.

Complexidade de $\text{BFS_VISIT}(s, G)$:

Sendo G dado por **listas de adjacências** e as operações MkEMPTYQUEUE , ENQUEUE , QUEUEISEMPTY e DEQUEUE suportadas em $O(1)$:

- a complexidade temporal de $\text{BFS_VISIT}(s, G)$ é $O(|V| + |A|)$;
- a complexidade espacial é $O(|V|)$, se a fila for suportada por um vetor (ou lista ligada com acesso ao primeiro e ao último elemento), além de $\Theta(|V| + |A|)$ para G .

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raiz s , que se chama **árvore de pesquisa em largura a partir de s** . Os ramos da árvore são os pares $(pai[v], v)$ tais que $pai[v] \neq \text{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, **mínimo** significa que tem o **menor número de ramos possível**). Os nós são visitados por **ordem crescente de distância a s** , nesse sentido.
- Se G for **não dirigido**, os vértices visitados na chamada $\text{BFS_VISIT}(s, G)$ são os que definem a **componente conexa** a que s pertence.

Complexidade de $\text{BFS_VISIT}(s, G)$:

Sendo G dado por **listas de adjacências** e as operações MkEMPTYQUEUE , ENQUEUE , QUEUEISEMPTY e DEQUEUE suportadas em $O(1)$:

- a complexidade temporal de $\text{BFS_VISIT}(s, G)$ é $O(|V| + |A|)$;
- a complexidade espacial é $O(|V|)$, se a fila for suportada por um vetor (ou lista ligada com acesso ao primeiro e ao último elemento), além de $\Theta(|V| + |A|)$ para G .

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raiz s , que se chama **árvore de pesquisa em largura a partir de s** . Os ramos da árvore são os pares $(pai[v], v)$ tais que $pai[v] \neq \text{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, **mínimo** significa que tem o **menor número de ramos possível**). Os nós são visitados por **ordem crescente de distância a s** , nesse sentido.
- Se G for **não dirigido**, os vértices visitados na chamada $\text{BFS_VISIT}(s, G)$ são os que definem a **componente conexa** a que s pertence.

Complexidade de $\text{BFS_VISIT}(s, G)$:

Sendo G dado por **listas de adjacências** e as operações MkEMPTYQUEUE , ENQUEUE , QUEUEISEMPTY e DEQUEUE suportadas em $O(1)$:

- a complexidade temporal de $\text{BFS_VISIT}(s, G)$ é $O(|V| + |A|)$;
- a complexidade espacial é $O(|V|)$, se a fila for suportada por um vetor (ou lista ligada com acesso ao primeiro e ao último elemento), além de $\Theta(|V| + |A|)$ para G .

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raiz s , que se chama **árvore de pesquisa em largura a partir de s** . Os ramos da árvore são os pares $(pai[v], v)$ tais que $pai[v] \neq \text{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, **mínimo** significa que tem o **menor número de ramos possível**). Os nós são visitados por **ordem crescente de distância a s** , nesse sentido.
- Se G for **não dirigido**, os vértices visitados na chamada $\text{BFS_VISIT}(s, G)$ são os que definem a **componente conexa** a que s pertence.

Complexidade de $\text{BFS_VISIT}(s, G)$:

Sendo G dado por **listas de adjacências** e as operações MKEMPTYQUEUE , ENQUEUE , QUEUEISEMPTY e DEQUEUE suportadas em $O(1)$:

- a complexidade temporal de $\text{BFS_VISIT}(s, G)$ é $O(|V| + |A|)$;
- a complexidade espacial é $O(|V|)$, se a fila for suportada por um vetor (ou lista ligada com acesso ao primeiro e ao último elemento), além de $\Theta(|V| + |A|)$ para G .

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $\text{pai}[\cdot]$ define uma árvore com raiz s , que se chama **árvore de pesquisa em largura a partir de s** . Os ramos da árvore são os pares $(\text{pai}[v], v)$ tais que $\text{pai}[v] \neq \text{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, **mínimo** significa que tem o **menor número de ramos possível**). Os nós são visitados por **ordem crescente de distância a s** , nesse sentido.
- Se G for **não dirigido**, os vértices visitados na chamada $\text{BFS_VISIT}(s, G)$ são os que definem a **componente conexa** a que s pertence.

Complexidade de $\text{BFS_Visit}(s, G)$:

Sendo G dado por **listas de adjacências** e as operações MkEMPTYQUEUE , ENQUEUE , QUEUEISEMPTY e DEQUEUE suportadas em $O(1)$:

- a complexidade temporal de $\text{BFS_VISIT}(s, G)$ é $O(|V| + |A|)$;
- a complexidade espacial é $O(|V|)$, se a fila for suportada por um vetor (ou lista ligada com acesso ao primeiro e ao último elemento), além de $\Theta(|V| + |A|)$ para G .

Visita o grafo $G = (V, A)$ em largura a partir de s

Propriedades

- $pai[\cdot]$ define uma árvore com raiz s , que se chama **árvore de pesquisa em largura a partir de s** . Os ramos da árvore são os pares $(pai[v], v)$ tais que $pai[v] \neq \text{NULL}$.
- O caminho de s até v na árvore é um **caminho mínimo de s até v** no grafo (aqui, **mínimo** significa que tem o **menor número de ramos possível**). Os nós são visitados por **ordem crescente de distância a s** , nesse sentido.
- Se G for **não dirigido**, os vértices visitados na chamada $\text{BFS_VISIT}(s, G)$ são os que definem a **componente conexa** a que s pertence.

Complexidade de $\text{BFS_VISIT}(s, G)$:

Sendo G dado por **listas de adjacências** e as operações MKEMPTYQUEUE , ENQUEUE , QUEUEISEMPTY e DEQUEUE suportadas em $O(1)$:

- a complexidade temporal de $\text{BFS_VISIT}(s, G)$ é $O(|V| + |A|)$;
- a complexidade espacial é $O(|V|)$, se a fila for suportada por um vetor (ou lista ligada com acesso ao primeiro e ao último elemento), além de $\Theta(|V| + |A|)$ para G .

Escrita do caminho de s para v (se existir), $s \neq v$

Assumindo que $pai[v] \neq \text{NULL}$, a função seguinte determina a sequência de vértices no caminho de s para v

EscreveCaminho(v, pai)

Se $pai[v] \neq \text{NULL}$ então

 ESCREVECAMINHO($pai[v], pai$);

escrever(v);

Filas FIFO (**para inteiros**) – queue.h (no arquivo)

Tipo QUEUE para suporte de filas FIFO (*first in first out*) para inteiros

```
typedef enum {FALSE,TRUE} BOOL;

typedef struct fila {
    int inicio, fim, nmax;
    int *queue;    // apontador para array (a reservar dinamicamente)
} QUEUE;
```

Operações disponíveis, com **complexidade temporal $O(1)$**

// criar fila para n inteiros	// libertar fila
QUEUE *mk_empty_queue(int n);	void free_queue(QUEUE *f);
// colocar valor na fila	// verificar se está vazia
void enqueue(int v,QUEUE *f);	BOOL queue_is_empty(QUEUE *f);
// retirar valor da fila	// verificar se está cheia
int dequeue(QUEUE *f);	BOOL queue_is_full(QUEUE *f);

BFS_VISIT em C (usando DAA1819_EstruturasDados)

```
#define UNDEF 0

void BFS_Visit(int s, GRAFO *g, int pai[]) {
    int v, w, n = NUM_VERTICES(g);
    QUEUE *q;
    ARCO *adjs;
    BOOL visitado[n+1];

    for(v=1; v <= n; v++) { // nós numerados de 1 a n
        visitado[v] = FALSE;
        pai[v] = UNDEF; // pai não encontrado
    }
    visitado[s] = TRUE;
    q = mk_empty_queue(n);
    enqueue(s,q);
    do {
        v = dequeue(q);
        adjs = ADJS_NO(v,g);
        while (adjs != NULL) {
            w = EXTREMO_FINAL(adjs);
            if (visitado[w] == FALSE) {
                enqueue(w,q);
                visitado[w] = TRUE;
                pai[w] = v;
            }
            adjs = PROX_ADJ(adjs);
        }
    } while (queue_is_empty(q) == FALSE);
    free_queue(q);
}
```

Visitar o grafo $G = (V, A)$ em largura

BFS(G)

```

Para cada  $v \in G.V$  fazer
     $visitado[v] \leftarrow \text{false};$ 
     $pai[v] \leftarrow \text{NULL};$ 
 $Q \leftarrow \text{MKEMPTYQUEUE}();$ 
Para cada  $v \in G.V$  fazer
    Se  $visitado[v] = \text{false}$  então
        BFS_VISIT( $v, G, Q$ );
  
```

BFS_Visit(s, G, Q)

```

 $visitado[s] \leftarrow \text{true};$ 
ENQUEUE( $s, Q$ );
Repita
     $v \leftarrow \text{DEQUEUE}(Q);$ 
    Para cada  $w \in G.Adjs[v]$  fazer
        Se  $visitado[w] = \text{false}$  então
            ENQUEUE( $w, Q$ );
             $visitado[w] \leftarrow \text{true};$ 
             $pai[w] \leftarrow v$ ;
até ( $\text{QUEUEISEMPTY}(Q) = \text{true}$ );
  
```

Assume que as variáveis $pai[\cdot]$ e $visitado[\cdot]$ são *globais*.

Distância mínima do nó s a cada nó (minimizar número de ramos)

BFS_Visit_Distancia(s, G)

```

Para cada  $v \in G.V$  fazer
     $visitado[v] \leftarrow \text{false};$ 
     $pai[v] \leftarrow \text{NULL};$ 
     $dist[v] \leftarrow \infty;$ 
 $visitado[s] \leftarrow \text{true};$ 
 $dist[s] \leftarrow 0;$ 
 $Q \leftarrow \text{MkEMPTYQUEUE}();$ 
 $\text{ENQUEUE}(s, Q);$ 
Repita
     $v \leftarrow \text{DEQUEUE}(Q);$ 
    Para cada  $w \in G.Adjs[v]$  fazer
        Se  $visitado[w] = \text{false}$  então
             $dist[w] \leftarrow dist[v] + 1;$ 
             $\text{ENQUEUE}(w, Q);$ 
             $visitado[w] \leftarrow \text{true};$ 
             $pai[w] \leftarrow v;$ 
até ( $\text{QUEUEISEMPTY}(Q) = \text{true}$  );
  
```

BFS visita vértices por ordem crescente de distância

Lema: Se $Q = w_1, w_2, \dots, w_k$ então $\text{dist}[w_1] \leq \text{dist}[w_2] \leq \dots \leq \text{dist}[w_k]$ e $\text{dist}[w_k] \leq \text{dist}[w_1] + 1$.

Proposição: Se v é acessível de s em $G = (V, E)$ então $\text{dist}[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v , para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $\text{dist}[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $\text{dist}[u] = \delta(s, u)$, para todo o u tal que $\text{dist}[u] < d$.

Seja v tal que $\text{dist}[v] = d$. De todos os caminhos mínimos de s para v , tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. Quando v' sai da fila, v está por visitar. Se não, pelo lema e hipótese, $\text{dist}[v] \leq \text{dist}[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v' , pois $s \rightsquigarrow \text{pai}[v] \rightarrow v$ seria um caminho mínimo também.

Logo, v' visita v e, portanto, $\text{dist}[v] = \text{dist}[v'] + 1$. Como $\text{dist}[v'] = d - 1 < d$, tem-se, pela hipótese, $\text{dist}[v'] = \delta(s, v')$. Logo, $\text{dist}[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Lema: Se $Q = w_1, w_2, \dots, w_k$ então $\text{dist}[w_1] \leq \text{dist}[w_2] \leq \dots \leq \text{dist}[w_k]$ e $\text{dist}[w_k] \leq \text{dist}[w_1] + 1$.

Proposição: Se v é acessível de s em $G = (V, E)$ então $\text{dist}[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v , para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $\text{dist}[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $\text{dist}[u] = \delta(s, u)$, para todo o u tal que $\text{dist}[u] < d$.

Seja v tal que $\text{dist}[v] = d$. De todos os caminhos mínimos de s para v , tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. Quando v' sai da fila, v está por visitar. Se não, pelo lema e hipótese, $\text{dist}[v] \leq \text{dist}[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v' , pois $s \rightsquigarrow \text{pai}[v] \rightarrow v$ seria um caminho mínimo também.

Logo, v' visita v e, portanto, $\text{dist}[v] = \text{dist}[v'] + 1$. Como $\text{dist}[v'] = d - 1 < d$, tem-se, pela hipótese, $\text{dist}[v'] = \delta(s, v')$. Logo, $\text{dist}[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Lema: Se $Q = w_1, w_2, \dots, w_k$ então $\text{dist}[w_1] \leq \text{dist}[w_2] \leq \dots \leq \text{dist}[w_k]$ e $\text{dist}[w_k] \leq \text{dist}[w_1] + 1$.

Proposição: Se v é acessível de s em $G = (V, E)$ então $\text{dist}[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v , para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $\text{dist}[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como hipótese de indução que $\text{dist}[u] = \delta(s, u)$, para todo o u tal que $\text{dist}[u] < d$.

Seja v tal que $\text{dist}[v] = d$. De todos os caminhos mínimos de s para v , tomamos aquele em que o nó v' que precede imediatamente v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. Quando v' sai da fila, v está por visitar. Se não, pelo lema e hipótese, $\text{dist}[v] \leq \text{dist}[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v' , pois $s \rightsquigarrow \text{pai}[v] \rightarrow v$ seria um caminho mínimo também.

Logo, v' visita v e, portanto, $\text{dist}[v] = \text{dist}[v'] + 1$. Como $\text{dist}[v'] = d - 1 < d$, tem-se, pela hipótese, $\text{dist}[v'] = \delta(s, v')$. Logo, $\text{dist}[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Lema: Se $Q = w_1, w_2, \dots, w_k$ então $\text{dist}[w_1] \leq \text{dist}[w_2] \leq \dots \leq \text{dist}[w_k]$ e $\text{dist}[w_k] \leq \text{dist}[w_1] + 1$.

Proposição: Se v é acessível de s em $G = (V, E)$ então $\text{dist}[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v , para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $\text{dist}[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $\text{dist}[u] = \delta(s, u)$, para todo o u tal que $\text{dist}[u] < d$.

Seja v tal que $\text{dist}[v] = d$. De todos os caminhos mínimos de s para v , tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. Quando v' sai da fila, v está por visitar. Se não, pelo lema e hipótese, $\text{dist}[v] \leq \text{dist}[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v' , pois $s \rightsquigarrow \text{pai}[v] \rightarrow v$ seria um caminho mínimo também.

Logo, v' visita v e, portanto, $\text{dist}[v] = \text{dist}[v'] + 1$. Como $\text{dist}[v'] = d - 1 < d$, tem-se, pela hipótese, $\text{dist}[v'] = \delta(s, v')$. Logo, $\text{dist}[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Lema: Se $Q = w_1, w_2, \dots, w_k$ então $\text{dist}[w_1] \leq \text{dist}[w_2] \leq \dots \leq \text{dist}[w_k]$ e $\text{dist}[w_k] \leq \text{dist}[w_1] + 1$.

Proposição: Se v é acessível de s em $G = (V, E)$ então $\text{dist}[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v , para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $\text{dist}[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $\text{dist}[u] = \delta(s, u)$, para todo o u tal que $\text{dist}[u] < d$.

Seja v tal que $\text{dist}[v] = d$. De todos os caminhos mínimos de s para v , tomamos aquele em que o nó v' que precede imediatamente v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. Quando v' sai da fila, v está por visitar. Se não, pelo lema e hipótese, $\text{dist}[v] \leq \text{dist}[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v' , pois $s \rightsquigarrow \text{pai}[v] \rightarrow v$ seria um caminho mínimo também.

Logo, v' visita v e, portanto, $\text{dist}[v] = \text{dist}[v'] + 1$. Como $\text{dist}[v'] = d - 1 < d$, tem-se, pela hipótese, $\text{dist}[v'] = \delta(s, v')$. Logo, $\text{dist}[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Lema: Se $Q = w_1, w_2, \dots, w_k$ então $\text{dist}[w_1] \leq \text{dist}[w_2] \leq \dots \leq \text{dist}[w_k]$ e $\text{dist}[w_k] \leq \text{dist}[w_1] + 1$.

Proposição: Se v é acessível de s em $G = (V, E)$ então $\text{dist}[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v , para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $\text{dist}[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $\text{dist}[u] = \delta(s, u)$, para todo o u tal que $\text{dist}[u] < d$.

Seja v tal que $\text{dist}[v] = d$. De todos os caminhos mínimos de s para v , tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. Quando v' sai da fila, v está por visitar. Se não, pelo lema e hipótese, $\text{dist}[v] \leq \text{dist}[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v' , pois $s \rightsquigarrow \text{pai}[v] \rightarrow v$ seria um caminho mínimo também.

Logo, v' visita v e, portanto, $\text{dist}[v] = \text{dist}[v'] + 1$. Como $\text{dist}[v'] = d - 1 < d$, tem-se, pela hipótese, $\text{dist}[v'] = \delta(s, v')$. Logo, $\text{dist}[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Lema: Se $Q = w_1, w_2, \dots, w_k$ então $\text{dist}[w_1] \leq \text{dist}[w_2] \leq \dots \leq \text{dist}[w_k]$ e $\text{dist}[w_k] \leq \text{dist}[w_1] + 1$.

Proposição: Se v é acessível de s em $G = (V, E)$ então $\text{dist}[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v , para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $\text{dist}[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $\text{dist}[u] = \delta(s, u)$, para todo o u tal que $\text{dist}[u] < d$.

Seja v tal que $\text{dist}[v] = d$. De todos os caminhos mínimos de s para v , tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. Quando v' sai da fila, v está por visitar. Se não, pelo lema e hipótese, $\text{dist}[v] \leq \text{dist}[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v' , pois $s \rightsquigarrow \text{pai}[v] \rightarrow v$ seria um caminho mínimo também.

Logo, v' visita v e, portanto, $\text{dist}[v] = \text{dist}[v'] + 1$. Como $\text{dist}[v'] = d - 1 < d$, tem-se, pela hipótese, $\text{dist}[v'] = \delta(s, v')$. Logo, $\text{dist}[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Lema: Se $Q = w_1, w_2, \dots, w_k$ então $\text{dist}[w_1] \leq \text{dist}[w_2] \leq \dots \leq \text{dist}[w_k]$ e $\text{dist}[w_k] \leq \text{dist}[w_1] + 1$.

Proposição: Se v é acessível de s em $G = (V, E)$ então $\text{dist}[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v , para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $\text{dist}[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $\text{dist}[u] = \delta(s, u)$, para todo o u tal que $\text{dist}[u] < d$.

Seja v tal que $\text{dist}[v] = d$. De todos os caminhos mínimos de s para v , tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. **Quando v' sai da fila, v está por visitar.** Se não, pelo lema e hipótese, $\text{dist}[v] \leq \text{dist}[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v' , pois $s \rightsquigarrow \text{pai}[v] \rightarrow v$ seria um caminho mínimo também.

Logo, v' visita v e, portanto, $\text{dist}[v] = \text{dist}[v'] + 1$. Como $\text{dist}[v'] = d - 1 < d$, tem-se, pela hipótese, $\text{dist}[v'] = \delta(s, v')$. Logo, $\text{dist}[v] = \delta(s, v') + 1 = \delta(s, v)$.

BFS visita vértices por ordem crescente de distância

Lema: Se $Q = w_1, w_2, \dots, w_k$ então $\text{dist}[w_1] \leq \text{dist}[w_2] \leq \dots \leq \text{dist}[w_k]$ e $\text{dist}[w_k] \leq \text{dist}[w_1] + 1$.

Proposição: Se v é acessível de s em $G = (V, E)$ então $\text{dist}[v] = \delta(s, v)$, onde $\delta(s, v)$ é o número de ramos do caminho mais curto s para v , para todo $v \neq s$.

Prova (por indução forte sobre a distância d):

- **Base:** Se $\delta(s, v) = 1$ então $(s, v) \in E$ e $\text{dist}[v] = 1$, pois s visita os adjacentes.
- **Hereditariedade:** Assumimos como **hipótese de indução** que $\text{dist}[u] = \delta(s, u)$, para todo o u tal que $\text{dist}[u] < d$.

Seja v tal que $\text{dist}[v] = d$. De todos os caminhos mínimos de s para v , tomamos aquele em que o nó v' que **precede imediatamente** v foi o primeiro a ser visitado no algoritmo. Seja $\gamma = s \rightsquigarrow v' \rightarrow v$ tal caminho. Tem-se $(v', v) \in E$ e $\delta(s, v) = \delta(s, v') + 1$. **Quando v' sai da fila, v está por visitar.** Se não, pelo lema e hipótese, $\text{dist}[v] \leq \text{dist}[v'] + 1 = \delta(s, v)$, o que é absurdo pela escolha de v' , pois $s \rightsquigarrow \text{pai}[v] \rightarrow v$ seria um caminho mínimo também.

Logo, v' visita v e, portanto, $\text{dist}[v] = \text{dist}[v'] + 1$. Como $\text{dist}[v'] = d - 1 < d$, tem-se, pela hipótese, $\text{dist}[v'] = \delta(s, v')$. Logo, $\text{dist}[v] = \delta(s, v') + 1 = \delta(s, v)$.

Obter as componentes conexas de um grafo não dirigido

- Em **BFS(G)**, o valor de $pai[v]$ identifica o primeiro nó que descobriu v durante a procura.
- No fim, o vetor $pai[.]$ define uma **floresta** de árvores pesquisa em largura.
- Por **análise para trás a partir de v** , podemos localizar a raiz da árvore de pesquisa em largura a que v pertence (podendo esta ser v se $pai[v] = \text{NULL}$).
- Para grafos **não dirigidos**, essa árvore define a **componente conexa** a que v pertence. Se G for **conexo**, a floresta reduz-se a uma árvore, à qual pertencem todos os vértices de G .

Uma **componente conexa de um grafo não dirigido** $G = (V, E)$ é um subgrafo $\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ tal que $V_{\mathcal{C}}$ é um conjunto *máximo* de nós *acessíveis uns dos outros*. Por definição, **u é acessível de v** se $u = v$ ou existe um **percurso** de v para u . Exemplo de grafo com quatro componentes conexas:



Obter as componentes conexas de um grafo não dirigido

- Em **BFS(G)**, o valor de $pai[v]$ identifica o primeiro nó que descobriu v durante a procura.
- No fim, o vetor $pai[.]$ define uma **floresta** de árvores pesquisa em largura.
- Por **análise para trás a partir de v** , podemos localizar a raiz da árvore de pesquisa em largura a que v pertence (podendo esta ser v se $pai[v] = \text{NULL}$).
- Para grafos **não dirigidos**, essa árvore define a **componente conexa** a que v pertence. Se G for **conexo**, a floresta reduz-se a uma árvore, à qual pertencem todos os vértices de G .

Uma **componente conexa de um grafo não dirigido** $G = (V, E)$ é um subgrafo $\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ tal que $V_{\mathcal{C}}$ é um conjunto *máximo* de nós *acessíveis uns dos outros*. Por definição, **u é acessível de v** se $u = v$ ou existe um **percurso** de v para u . Exemplo de grafo com quatro componentes conexas:



Obter as componentes conexas de um grafo não dirigido

- Em **BFS(G)**, o valor de $pai[v]$ identifica o primeiro nó que descobriu v durante a procura.
- No fim, o vetor $pai[.]$ define uma **floresta** de árvores pesquisa em largura.
- Por **análise para trás a partir de v** , podemos localizar a raiz da árvore de pesquisa em largura a que v pertence (podendo esta ser v se $pai[v] = \text{NULL}$).
- Para grafos não dirigidos, essa árvore define a **componente conexa** a que v pertence. Se G for **conexo**, a floresta reduz-se a uma árvore, à qual pertencem todos os vértices de G .

Uma **componente conexa de um grafo não dirigido** $G = (V, E)$ é um subgrafo $\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ tal que $V_{\mathcal{C}}$ é um conjunto *máximo* de nós *acessíveis uns dos outros*. Por definição, **u é acessível de v** se $u = v$ ou existe um **percurso** de v para u . Exemplo de grafo com quatro componentes conexas:



Obter as componentes conexas de um grafo não dirigido

- Em **BFS(G)**, o valor de $pai[v]$ identifica o primeiro nó que descobriu v durante a procura.
- No fim, o vetor $pai[.]$ define uma **floresta** de árvores pesquisa em largura.
- Por **análise para trás a partir de v** , podemos localizar a raiz da árvore de pesquisa em largura a que v pertence (podendo esta ser v se $pai[v] = \text{NULL}$).
- Para grafos **não dirigidos**, essa árvore define a **componente conexa** a que v pertence. Se G for **conexo**, a floresta reduz-se a uma árvore, à qual pertencem todos os vértices de G .

Uma **componente conexa de um grafo não dirigido** $G = (V, E)$ é um subgrafo $\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ tal que $V_{\mathcal{C}}$ é um conjunto *máximo* de nós *acessíveis uns dos outros*. Por definição, **u é acessível de v** se $u = v$ ou existe um **percurso** de v para u . Exemplo de grafo com quatro componentes conexas:



Obter as componentes conexas de um grafo não dirigido

Proposição: Se G for um grafo não dirigido, cada árvore da floresta obtida por $\text{BFS}(G)$ identifica uma componente conexa do grafo.

Ideia da prova:

- G pode ser representado por um **grafo dirigido simétrico** G' (que designamos por **adjunto** de G)
- Os vértices que constituem a árvore a que w pertence não dependem do nó raiz (embora a estrutura da árvore possa ser diferente os nós são os mesmos).
- Na chamada de $\text{BFS}(v, G, Q)$ no segundo ciclo de $\text{BFS}(G)$, serão visitados todos os nós acessíveis de v em G .
- Como o grafo adjunto de G é simétrico, se algum w acessível de v tivesse sido visitado numa chamada anterior, então também v teria de ter sido marcado como visitado por algum dos descendentes de w .

Visita grafo $G = (V, A)$ em profundidade

DFS(G) Complexidade $\Theta(|V| + |A|)$

```

instante  $\leftarrow 0$ ;
Para cada  $v \in G.V$  fazer  $cor[v] \leftarrow$  branco;  $Pai[v] \leftarrow$  NULL;
Para cada  $v \in G.V$  fazer
    Se  $cor[v] =$  branco então DFS_VISIT( $v, G$ );
  
```

DFS_VISIT(v, G) Complexidade $O(|V| + |A|)$

```

/* Vértices por visitar ainda a branco */
instante  $\leftarrow$  instante + 1;
t_inicial[ $v$ ]  $\leftarrow$  instante;
cor[ $v$ ]  $\leftarrow$  cinzento;      // cor útil para detetar ciclos
Para cada  $w \in G.Adjs[v]$  fazer
    Se  $cor[w] =$  branco então      // se fosse cinzento, G teria um ciclo
        Pai[ $w$ ]  $\leftarrow v$ ;
        DFS_VISIT( $w, G$ );
cor[ $v$ ]  $\leftarrow$  preto;
instante  $\leftarrow$  instante + 1;
t_final[ $v$ ]  $\leftarrow$  instante;      // tempo de finalização para  $v$ 
  
```

Assume que as variáveis $cor[\cdot]$, $Pai[\cdot]$, $t_inicial[\cdot]$, $t_final[\cdot]$ e $instante$ são globais.

Construção de pilha na visita em profundidade

Versão em que DFS produz **stack** (pilha) com os **nós ordenados por ordem decrescente de tempo de finalização**.

DFS(G)

```

  stack  $\leftarrow$  MK_EMPTY_STACK();
  Para cada  $v \in G.V$  fazer
    visitado[ $v$ ]  $\leftarrow$  false;

  Para cada  $v \in G.V$  fazer
    Se visitado[ $v$ ] = false então
      DFS_VISIT( $v, G, visitado, stack$ );
  return stack;
```

DFS_VISIT($v, G, visitado, stack$)

```

  visitado[ $v$ ]  $\leftarrow$  true;
  Para cada  $w \in G.Adjs[v]$  fazer
    Se visitado[ $w$ ] = false então
      DFS_VISIT( $w, G, visitado, stack$ );
  PUSH( $v, stack$ );
```

Exemplo: Planeamento de tarefas

Problema de escalonamento de tarefas

Um projeto é constituído por um conjunto de tarefas, sendo conhecida a duração de cada tarefa e as restrições de precedência entre tarefas. Não se pode dar início a uma tarefa sem concluir as que a precedem. Pretende-se agendar as tarefas de modo a concluir o projeto o mais cedo possível.

- **Cenário 1:** Há apenas uma pessoa para realizar o projeto e não pode realizar várias tarefas ao mesmo tempo.

Problema: Determinar um plano – ordenação das tarefas – que seja compatível com as precedências definidas.

- **Cenário 2:** As tarefas não partilham recursos. Podem ser realizadas várias tarefas simultaneamente, devendo satisfazer as precedências definidas.

Problema: Determinar quando é que o projeto pode ficar concluído, se:

- (i) todas as tarefas tiverem duração unitária;
- (ii) as tarefas puderem ter durações distintas.

Exemplo: Planeamento de tarefas

Problema de escalonamento de tarefas

Um projeto é constituído por um conjunto de tarefas, sendo conhecida a duração de cada tarefa e as restrições de precedência entre tarefas. Não se pode dar início a uma tarefa sem concluir as que a precedem. Pretende-se agendar as tarefas de modo a concluir o projeto o mais cedo possível.

- **Cenário 1:** Há apenas uma pessoa para realizar o projeto e não pode realizar várias tarefas ao mesmo tempo.

Problema: Determinar um plano – ordenação das tarefas – que seja compatível com as precedências definidas.

- **Cenário 2:** As tarefas não partilham recursos. Podem ser realizadas várias tarefas simultaneamente, devendo satisfazer as precedências definidas.

Problema: Determinar quando é que o projeto pode ficar concluído, se:

- (i) todas as tarefas tiverem duração unitária;
- (ii) as tarefas puderem ter durações distintas.

Exemplo: Planeamento de tarefas

Problema de escalonamento de tarefas

Um projeto é constituído por um conjunto de tarefas, sendo conhecida a duração de cada tarefa e as restrições de precedência entre tarefas. Não se pode dar início a uma tarefa sem concluir as que a precedem. Pretende-se agendar as tarefas de modo a concluir o projeto o mais cedo possível.

- **Cenário 1:** Há apenas uma pessoa para realizar o projeto e não pode realizar várias tarefas ao mesmo tempo.

Problema: Determinar um plano – ordenação das tarefas – que seja compatível com as precedências definidas.

- **Cenário 2:** As tarefas não partilham recursos. Podem ser realizadas várias tarefas simultaneamente, devendo satisfazer as precedências definidas.

Problema: Determinar quando é que o projeto pode ficar concluído, se:

- (i) todas as tarefas tiverem duração unitária;
- (ii) as tarefas puderem ter durações distintas.

Exemplo: Planeamento de tarefas

Problema de escalonamento de tarefas

Um projeto é constituído por um conjunto de tarefas, sendo conhecida a duração de cada tarefa e as restrições de precedência entre tarefas. Não se pode dar início a uma tarefa sem concluir as que a precedem. Pretende-se agendar as tarefas de modo a concluir o projeto o mais cedo possível.

- **Cenário 1:** Há apenas uma pessoa para realizar o projeto e não pode realizar várias tarefas ao mesmo tempo.

Problema: Determinar um plano – ordenação das tarefas – que seja compatível com as precedências definidas.

- **Cenário 2:** As tarefas não partilham recursos. Podem ser realizadas várias tarefas simultaneamente, devendo satisfazer as precedências definidas.

Problema: Determinar quando é que o projeto pode ficar concluído, se:

- (i) todas as tarefas tiverem duração unitária;
- (ii) as tarefas puderem ter durações distintas.

Exemplo: Planeamento de tarefas

Problema de escalonamento de tarefas

Um projeto é constituído por um conjunto de tarefas, sendo conhecida a duração de cada tarefa e as restrições de precedência entre tarefas. Não se pode dar início a uma tarefa sem concluir as que a precedem. Pretende-se agendar as tarefas de modo a concluir o projeto o mais cedo possível.

- **Cenário 1:** Há apenas uma pessoa para realizar o projeto e não pode realizar várias tarefas ao mesmo tempo.

Problema: Determinar um plano – ordenação das tarefas – que seja compatível com as precedências definidas.

- **Cenário 2:** As tarefas não partilham recursos. Podem ser realizadas várias tarefas simultaneamente, devendo satisfazer as precedências definidas.

Problema: Determinar quando é que o projeto pode ficar concluído, se:

- (i) todas as tarefas tiverem duração unitária;
- (ii) as tarefas puderem ter durações distintas.

Exemplo: Planeamento de tarefas

Problema de escalonamento de tarefas

Um projeto é constituído por um conjunto de tarefas, sendo conhecida a duração de cada tarefa e as restrições de precedência entre tarefas. Não se pode dar início a uma tarefa sem concluir as que a precedem. Pretende-se agendar as tarefas de modo a concluir o projeto o mais cedo possível.

- **Cenário 1:** Há apenas uma pessoa para realizar o projeto e não pode realizar várias tarefas ao mesmo tempo.

Problema: Determinar um plano – ordenação das tarefas – que seja compatível com as precedências definidas.

- **Cenário 2:** As tarefas não partilham recursos. Podem ser realizadas várias tarefas simultaneamente, devendo satisfazer as precedências definidas.

Problema: Determinar quando é que o projeto pode ficar concluído, se:

- (i) todas as tarefas tiverem duração unitária;
- (ii) as tarefas puderem ter durações distintas.

Exemplo: Planeamento de tarefas

A **relação de precedência** é dada por um **DAG (grafo dirigido acíclico)**. Os nós do grafo definem as tarefas e um ramo (x, y) representa o facto de a tarefa x preceder a tarefa y . Se tem os ramos (x, y) e (y, z) , também x precede z mesmo que não tenha o ramo (x, z) .

- **Cenário 1: Nenhum par de tarefas a decorrer simultaneamente.**

Problema: Ordenação topológica dos nós de um DAG

- **Cenário 2: Algumas tarefas podem decorrer em simultâneo.** Calcular a duração mínima do projeto se:

- (i) todas as tarefas têm **duração unitária**;

Problema: Encontrar o **caminho mais longo num DAG**, sendo o comprimento é dado pelo **número de ramos** no caminho.

- (ii) as tarefas puderem ter **durações distintas**.

Problema: Encontrar o **caminho mais longo num DAG**, sendo o comprimento dado pela **soma dos valores nos nós** do caminho.

Ordenação topológica dos nós de um DAG

Ordenação topológica de um grafo dirigido acíclico, $G = (V, A)$ finito, é uma função bijetiva σ de V em $\{0 \dots, |V| - 1\}$ tal que $\sigma(v) < \sigma(w)$, para todo $(v, w) \in A$.

Ou seja, σ indica uma ordenação dos nós que é compatível com a relação de precedência definida por G .

TOPSORTDAG(G)

```

Para todo  $v \in G.V$  fazer  $GrauE[v] \leftarrow 0$ ;
Para todo  $(v, w) \in G.A$  fazer  $GrauE[w] \leftarrow GrauE[v] + 1$ ;
 $S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$ ; /*  $S$  deve ser suportado por uma fila ou uma pilha. */
 $i \leftarrow 0$ ;
Enquanto  $(S \neq \emptyset)$  fazer
     $v \leftarrow$  um qualquer elemento de  $S$ ;   $S \leftarrow S \setminus \{v\}$ ;
     $\sigma[v] \leftarrow i$ ;   $i \leftarrow i + 1$ ;
    Para todo  $w \in G.Adjs[v]$  fazer
         $GrauE[w] \leftarrow GrauE[w] - 1$ ;
        Se  $GrauE[w] = 0$  então  $S \leftarrow S \cup \{w\}$ ;
  
```

Propriedade que suporta a correção: Qualquer DAG tem algum nó com grau de entrada zero. Se retirar um ramo a um DAG, obtém um DAG.

Observar também que: se G não for um DAG, o ciclo "Enquanto" termina sempre mas com $i < n$ em vez de $i = n$.

Ordenação topológica dos nós de um DAG

Ordenação topológica de um grafo dirigido acíclico, $G = (V, A)$ finito, é uma função bijetiva σ de V em $\{0 \dots, |V| - 1\}$ tal que $\sigma(v) < \sigma(w)$, para todo $(v, w) \in A$.

Ou seja, σ indica uma ordenação dos nós que é compatível com a relação de precedência definida por G .

TOPSORTDAG(G)

```

Para todo  $v \in G.V$  fazer  $GrauE[v] \leftarrow 0$ ;
Para todo  $(v, w) \in G.A$  fazer  $GrauE[w] \leftarrow GrauE[v] + 1$ ;
 $S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$ ; /*  $S$  deve ser suportado por uma fila ou uma pilha. */
 $i \leftarrow 0$ ;
Enquanto  $(S \neq \emptyset)$  fazer
     $v \leftarrow$  um qualquer elemento de  $S$ ;  $S \leftarrow S \setminus \{v\}$ ;
     $\sigma[v] \leftarrow i$ ;  $i \leftarrow i + 1$ ;
    Para todo  $w \in G.Adjs[v]$  fazer
         $GrauE[w] \leftarrow GrauE[w] - 1$ ;
        Se  $GrauE[w] = 0$  então  $S \leftarrow S \cup \{w\}$ ;
  
```

Propriedade que suporta a correção: Qualquer DAG tem algum nó com **grau de entrada zero**. Se retirar um ramo a um DAG, obtém um DAG.

Observar também que: se G não for um DAG, o ciclo "Enquanto" termina sempre mas com $i < n$ em vez de $i = n$.

Ordenação topológica dos nós de um DAG

Ordenação topológica de um grafo dirigido acíclico, $G = (V, A)$ finito, é uma função bijetiva σ de V em $\{0 \dots, |V| - 1\}$ tal que $\sigma(v) < \sigma(w)$, para todo $(v, w) \in A$.

Ou seja, σ indica uma ordenação dos nós que é compatível com a relação de precedência definida por G .

TOPSORTDAG(G)

```

Para todo  $v \in G.V$  fazer  $GrauE[v] \leftarrow 0$ ;
Para todo  $(v, w) \in G.A$  fazer  $GrauE[w] \leftarrow GrauE[v] + 1$ ;
 $S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$ ; /*  $S$  deve ser suportado por uma fila ou uma pilha. */
 $i \leftarrow 0$ ;
Enquanto  $(S \neq \emptyset)$  fazer
     $v \leftarrow$  um qualquer elemento de  $S$ ;  $S \leftarrow S \setminus \{v\}$ ;
     $\sigma[v] \leftarrow i$ ;  $i \leftarrow i + 1$ ;
    Para todo  $w \in G.Adjs[v]$  fazer
         $GrauE[w] \leftarrow GrauE[w] - 1$ ;
        Se  $GrauE[w] = 0$  então  $S \leftarrow S \cup \{w\}$ ;
  
```

Propriedade que suporta a correção: Qualquer DAG tem algum nó com **grau de entrada zero**. Se retirar um ramo a um DAG, obtém um DAG.

Observar também que: se G não for um DAG, o ciclo "Enquanto" termina sempre mas com $i < n$ em vez de $i = n$.

Ordenação topológica por pesquisa em profundidade

TOPSORT_DFS(G)

```

 $S \leftarrow \{\}$ ;      // definir stack vazia
Para cada  $v \in G.V$  fazer  $cor[v] \leftarrow \text{branco}$ ;
Para cada  $v \in G.V$  fazer
    Se  $cor[v] = \text{branco}$  então TOPSORT_DFSVISIT( $v, G, S$ );
/* Enquanto ( $S \neq \{\}$ ) fazer escrever(Pop( $S$ )); */
 $i \leftarrow 0$ ;
Enquanto ( $S \neq \{\}$ ) fazer
     $\sigma[\text{POP}(S)] \leftarrow i$ ;  /* como anteriormente */
     $i \leftarrow i + 1$ ;

```

TOPSORT_DFSVISIT(v, G, S)

```

 $cor[v] \leftarrow \text{cinzento}$ ;
Para cada  $w \in G.Adjs[v]$  fazer
    Se  $cor[w] = \text{branco}$  então
        TOPSORT_DFSVISIT( $w, G, S$ );
    senão se  $cor[w] = \text{cinzento}$  então // retirar se sabe que  $G$  é DAG
        Termina com indicação de erro ( $G$  não é DAG); // retirar se sabe que  $G$  é DAG
 $cor[v] \leftarrow \text{preto}$ ;
PUSH( $v, S$ );

```

Ordenação topológica de um DAG por DFS (retorna stack)

TOPSORT_DFS(G)

```

 $S \leftarrow \{\}$ ;    // definir stack vazia
Para cada  $v \in G.V$  fazer  $visitado[v] \leftarrow \mathbf{false}$ ;
Para cada  $v \in G.V$  fazer
    Se  $visitado[v] = \mathbf{false}$  então TOPSORT_DFSVISIT( $v, G, visitado, S$ );
retorna  $S$ ;    // ordem topológica se efetuar POPs sucessivos de  $S$ 

```

TOPSORT_DFSVISIT($v, G, visitado, S$)

```

 $visitado[v] \leftarrow \mathbf{true}$ ;
Para cada  $w \in G.Adjs[v]$  fazer
    Se  $visitado[w] = \mathbf{false}$  então
        TOPSORT_DFSVISIT( $w, G, visitado, S$ );
PUSH( $v, S$ );

```

Pilhas – stack_v1.h (no arquivo)

Tipo STACK para suporte de pilhas para inteiros; PSTACK equivale a STACK *

```
typedef struct {
    int top, maxstack;
    int *stack;
} STACK, *PSTACK;
```

Operações disponíveis, com **complexidade temporal** $O(1)$

```
// criar pilha para n inteiros          // libertar pilha
PSTACK st_make_empty(int nmax);         void st_destroy(PSTACK pstack);

// colocar valor no topo                // verificar se está vazia
void st_push(PSTACK pstack, int x);     BOOL st_is_empty(PSTACK pstack);

// retirar valor do topo                // verificar se está cheia
int st_pop(PSTACK pstack);              BOOL st_is_full(PSTACK pstack);
```

Caminho máximo num DAG (distâncias unitárias)

Problema: determinar um caminho máximo num grafo dirigido acíclico $G = (V, A)$, sendo o comprimento dado pelo número de ramos do caminho.

MaxPathDAG(G)

```

Para todo  $v \in G.V$  fazer  $ES[v] \leftarrow 0$ ;  $Pai[v] \leftarrow \text{NULL}$ ;  $GrauE[v] \leftarrow 0$ ;
Para todo  $(v, w) \in G.A$  fazer  $GrauE[w] \leftarrow GrauE[v] + 1$ ;
 $S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$ ; /*  $S$  deve ser suportado por uma fila ou uma pilha. */
 $Max \leftarrow -1$ ;  $v_f \leftarrow \text{NULL}$ ; //  $v_f$  indica o vértice final no caminho mais longo obtido
Enquanto  $(S \neq \emptyset)$  fazer
     $v \leftarrow$  um qualquer elemento de  $S$ ;
     $S \leftarrow S \setminus \{v\}$ ;
    Se  $Max < ES[v]$  então  $Max \leftarrow ES[v]$ ;  $v_f \leftarrow v$ ; /*  $ES[v]$  o número de ramos do caminho */
    Para todo  $w \in G.Adjs[v]$  fazer
        Se  $ES[w] < ES[v] + 1$  então
             $ES[w] \leftarrow ES[v] + 1$ ;  $Pai[w] \leftarrow v$ ;
             $GrauE[w] \leftarrow GrauE[v] - 1$ ;
        Se  $GrauE[w] = 0$  então  $S \leftarrow S \cup \{w\}$ ;
    ESCREVECAMINHO( $v_f, Pai$ ); escrever( $Max$ );
  
```

Caminho máximo num DAG com pesos associados aos nós

Dado $G = (V, A, D)$ em que $D(v) \in \mathbb{R}_0^+$ é a duração da tarefa $v \in V$, determinar $ES[v]$, o instante mais próximo em que pode dar início a v ("earliest start").

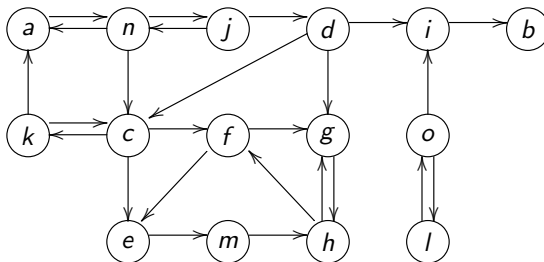
MaxPathWeightedDAG(G)

```

Para todo  $v \in G.V$  fazer  $ES[v] \leftarrow 0$ ;  $Pai[v] \leftarrow \text{Nenhum}$ ;  $GrauE[v] \leftarrow 0$ ;
Para todo  $(v, w) \in G.A$  fazer  $GrauE[w] \leftarrow GrauE[v] + 1$ ;
 $S \leftarrow \{v \in G.V \mid GrauE[v] = 0\}$ ; /*  $S$  deve ser suportado por uma fila ou uma pilha. */
 $Max \leftarrow -1$ ;  $v_f \leftarrow \text{NULL}$ ; /*  $ES[v]$  o número de ramos do caminho */
Enquanto ( $S \neq \emptyset$ ) fazer
     $v \leftarrow$  um qualquer elemento de  $S$ ;  $S \leftarrow S \setminus \{v\}$ ;
    Se  $Max < ES[v] + D[v]$  então  $Max \leftarrow ES[v] + D[v]$ ;  $v_f \leftarrow v$ ;
    Para todo  $w \in G.Adjs[v]$  fazer
        Se  $ES[w] < ES[v] + D[v]$  então
             $ES[w] \leftarrow ES[v] + D[v]$ ;  $Pai[w] \leftarrow v$ ;
             $GrauE[w] \leftarrow GrauE[v] - 1$ ;
        Se  $GrauE[w] = 0$  então  $S \leftarrow S \cup \{w\}$ ;
    ESCREVECAMINHO( $v_f, Pai$ ); escrever( $Max$ );
  
```

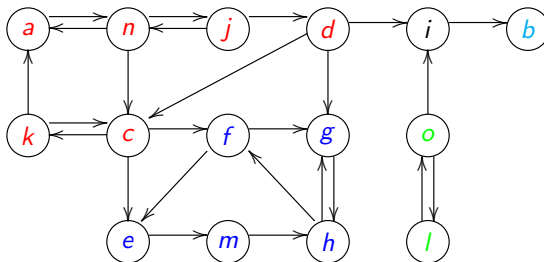
Componentes fortemente conexos

Seja $G = (V, E)$ um grafo **dirigido**. Uma **componente fortemente conexa** G é um subgrafo $\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ tal que $V_{\mathcal{C}}$ é um conjunto *máximo* de nós *acessíveis uns dos outros*. Por definição, u é **acessível de** v se $u = v$ ou existe um **percurso** de v para u .



Componentes fortemente conexos

Seja $G = (V, E)$ um grafo **dirigido**. Uma **componente fortemente conexa** G é um subgrafo $\mathcal{C} = (V_{\mathcal{C}}, E_{\mathcal{C}})$ tal que $V_{\mathcal{C}}$ é um conjunto *máximo* de nós *acessíveis uns dos outros*. Por definição, u é **acessível de** v se $u = v$ ou existe um **percurso** de v para u .



Componentes fortemente conexas

Algoritmo de Kosaraju-Sharir

```

Usar  $DFS(G)$  para obter pilha  $S$  com os nós por ordem decrescente de  $t\_final[\cdot]$ 
Para  $v \in G.V$  fazer  $cor[v] \leftarrow \text{branco}$ ;
Enquanto  $(S \neq \{\})$  fazer
     $v \leftarrow POP(S)$ ;
    Se  $cor[v] = \text{branco}$  então  $DFS\_VISIT(v, G^T)$  e indica os nós visitados;
  
```

$G^T = (V, A^T)$ denota o **grafo transposto** de $G = (V, A)$, obtém-se de G se se trocar o sentido dos arcos, sendo, $A^T = \{(y, x) \mid (x, y) \in A\}$.

Complexidade temporal do algoritmo de Kosaraju-Sharir

O algoritmo de Kosaraju-Sharir tem complexidade $\Theta(|V| + |A|)$, (ou seja, linear na estrutura do grafo), se o grafo for representado por listas de adjacências.

Correção do Algoritmo Kosaraju-Sharir

- O *grafo das componentes fortemente conexas* G_{scc} de um grafo dirigido é um grafo dirigido acíclico (DAG).

G_{scc} : os nós correspondem às componentes fortemente conexas de G e os ramos são os pares (C, C') tais que $C \neq C'$ e existe algum ramo de algum nó de C para algum nó de C' em G .

Se G_{scc} não fosse acíclico, quaisquer dois nós x e y que estivessem em componentes C_x e C_y (distintas) envolvidas num ciclo seriam acessíveis um do outro em G . Isso é absurdo, pois contradiz a noção de componente fortemente conexa.

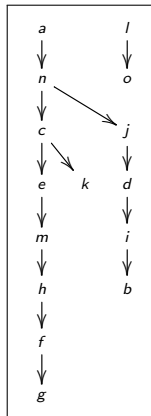
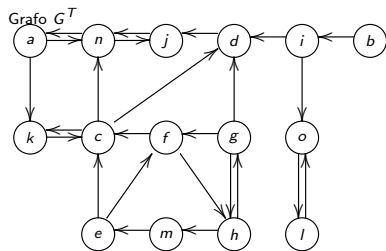
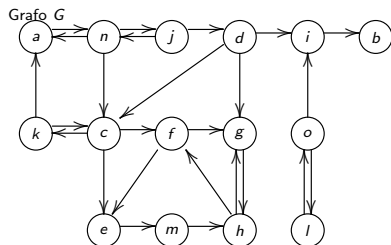
- As componentes fortemente conexas de G e G^T têm os mesmos nós.

Qualquer percurso de x para y em G é um percurso de y para x em G^T (e vice-versa).

- Uma ordenação topológica das componentes de G corresponde a uma ordenação topológica por ordem inversa (da cronológica) para as componentes de G^T . O DAG de componentes de G^T é o transposto do DAG de componentes de G .

- A ordem dada por S para visita de G^T faz com que as componentes acessíveis de uma dada componente já estejam visitadas quando entra nessa componente.

Componentes fortemente conexos (exemplo)



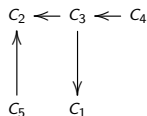
Pilha

l
o
a
n
j
d
i
b
c
k
e
m
h
f
g

Componentes
fortemente conexos

$$\begin{aligned} C_1 &= \{l, o\} \\ C_2 &= \{a, n, j, k, c, d\} \\ C_3 &= \{i\} \\ C_4 &= \{b\} \\ C_5 &= \{e, f, h, g, m\} \end{aligned}$$

DAG componentes em G^T



Árvores geradoras com peso mínimo/máximo

Exemplo

Uma companhia de distribuição de gás natural pretende construir uma rede que assegure a distribuição a um certo número de locais a partir de um dado local. Dados os custos da ligação entre cada par de locais, há que determinar as ligações a efetuar de modo a reduzir os custos globais.

Resolução

- As árvores são os grafos não dirigidos conexos com menos ramos.
- Determinar uma **árvore geradora de peso mínimo** (*minimum spanning tree*) num grafo $G = (V, E, d)$ não dirigido, finito e **conexo** com valores associados aos ramos, em que $d : E \rightarrow \mathbb{R}_0^+$ indica o valor associado a cada ramo.
- Designação alternativa: árvore de suporte de peso mínimo/máximo.
- **Algoritmos de Prim** (1957) e de **Kruskal** (1956).

Baseiam-se em **estratégias “greedy”** (gulosas, ávidas, gananciosas).

Em cada iteração, a seleção localmente ótima. Não haverá retrocesso para analisar outras possibilidades.

Algoritmo de Kruskal [1956] - minimum spanning tree

- São escolhidos sucessivamente os $|V| - 1$ ramos da árvore de suporte;
- os ramos são analisados por ordem crescente de valores de peso, e
- o ramo corrente só não fará parte da árvore de suporte se o grafo resultante da sua junção à floresta construída até esse passo ficar com um ciclo.

Algoritmo de Kruskal - pseudocódigo

Dados: Um grafo $G = (V, E, d)$ não dirigido, conexo, com valores nos ramos.

Resultado: O conjunto de ramos T na árvore mínima de suporte de G .

Ordenar E por ordem crescente de valores nos ramos.

$T \leftarrow \emptyset$; $\mathcal{C} \leftarrow \{\{v\} \mid v \in V\}$;

Enquanto $|T| \neq |V| - 1$ fazer

Seja $\langle u, v \rangle \in E$ o primeiro ramo não escolhido (na ordem considerada).

Sejam C_u e C_v os elementos de \mathcal{C} tais que $u \in C_u$ e $v \in C_v$.

Se $C_u \neq C_v$ então $T \leftarrow T \cup \{\langle u, v \rangle\}$; $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{C_u, C_v\}) \cup \{C_u \cup C_v\}$;

Algoritmo de Kruskal - pseudocódigo

[CLRS 23.2]

MST_KRUSKAL(G, w)

$T \leftarrow \emptyset$

Para cada vértice $v \in G.V$

MAKE_SET(v)

Ordenar as arestas de $G.E$ por ordem não decrescente de peso w

Para cada aresta $(u, v) \in G.E$ (por ordem não decrescente de peso) fazer

 Se **FIND_SET**(u) \neq **FIND_SET**(v)

$T \leftarrow T \cup \{(u, v)\}$

UNION(u, v)

Algoritmo de Kruskal - pseudocódigo

Árvore geradora com peso mínimo:

ALGORITMOKRUSKAL(G)

1. $Q \leftarrow$ Fila que representa E por ordem crescente de valores nos ramos;
2. $T \leftarrow \emptyset$;
3. $\mathcal{C} \leftarrow \text{INIT_SINGLETONS}(V)$;
4. Enquanto $(|T| \neq |V| - 1 \wedge \text{QUEUEISEMPTY}(Q) = \text{false})$ fazer
5. $\langle u, v \rangle \leftarrow \text{DEQUEUE}(Q)$;
6. Se $\text{FINDSET}(u, \mathcal{C}) \neq \text{FINDSET}(v, \mathcal{C})$ então
7. $T \leftarrow T \cup \{\langle u, v \rangle\}$;
8. $\text{UNION}(u, v, \mathcal{C})$;

A condição $\text{QUEUEISEMPTY}(Q) = \text{false}$ é redundante se o grafo for conexo. Contudo, permite que o algoritmo possa ser aplicado a um grafo não conexo para obter a **floresta de árvores geradoras mínimas das suas componentes conexas**.

Árvore geradora com peso máximo: obtém-se por aplicação do algoritmo se se ordenar os ramos por ordem decrescente de peso inicialmente.

Algoritmo de Prim [1957] - Minimum spanning tree

- São escolhidos sucessivamente os $|V|$ vértices da árvore;
- em cada passo, é ligado, à sub-árvore já construída, o vértice que está **mais próximo** dos já nela incluídos.
- O primeiro vértice pode ser qualquer um dos vértices do grafo.

Árvore geradora com peso máximo: obtém-se por aplicação do algoritmo se em cada iteração se ligar o nó **mais afastado** dos que já estão na árvore (na implementação, usa *“heap de máximo”*).

Algoritmo de Prim – pseudocódigo

ALGORITMO PRIM(G, s) // [CLRS 23.2]	
Para cada $v \in V$ fazer { $pai[v] \leftarrow \text{NULL}$; $dist[v] \leftarrow \infty$; ok [v] \leftarrow false ; }	$\Theta(V)$
$dist[s] \leftarrow 0$;	$O(1)$
$Q \leftarrow \text{Mk_PQ_HEAPMIN}(dist, V)$;	$\Theta(V)$
$T \leftarrow \{\}$;	$O(1)$
Enquanto (PQ_NOT_EMPTY (Q)) fazer	
$v \leftarrow \text{EXTRACTMIN}(Q)$;	$O(\log_2 V)$
$T \leftarrow T \cup \{(pai[v], v)\}$; ok [v] \leftarrow true ; /* ok [v] indica se v já está em T */	$O(1)$
Para cada $w \in Adjs[v]$ fazer	
Se ok [w] = false e $d(v, w) < dist[w]$ então	$O(1)$
$dist[w] \leftarrow d(v, w)$;	$O(1)$
$pai[w] \leftarrow v$;	$O(1)$
DECREASEKEY ($Q, w, dist[w]$);	$O(\log_2 V)$

NB: Apenas as distâncias dos nós que estão na fila podem ser alteradas

Complexidade Temporal $O(|E| \log |V|)$, se for suportado por uma **heap de mínimo**.
Como G é conexo, $|E| \geq |V| - 1$. O ciclo “Enquanto” domina a complexidade, sendo:

$$O\left(\sum_{v \in V} (1 + \log_2 |V| + |Adjs[v]| \log_2 |V|)\right) = O(|V| \log_2 |V| + |E| \log_2 |V|) = O(|E| \log_2 |V|)$$

Algoritmo de Prim – pseudocódigo

ALGORITMO PRIM(G, s) // [CLRS 23.2]	
Para cada $v \in V$ fazer { $pai[v] \leftarrow \text{NULL}$; $dist[v] \leftarrow \infty$; ok [v] \leftarrow false ; }	$\Theta(V)$
$dist[s] \leftarrow 0$;	$O(1)$
$Q \leftarrow \text{Mk_PQ_HEAPMIN}(dist, V)$;	$\Theta(V)$
$T \leftarrow \{\}$;	$O(1)$
Enquanto (PQ_NOT_EMPTY (Q)) fazer	
$v \leftarrow \text{EXTRACTMIN}(Q)$;	$O(\log_2 V)$
$T \leftarrow T \cup \{(pai[v], v)\}$; ok [v] \leftarrow true ; /* ok [v] indica se v já está em T */	$O(1)$
Para cada $w \in \text{Adjs}[v]$ fazer	
Se ok [w] = false e $d(v, w) < dist[w]$ então	$O(1)$
$dist[w] \leftarrow d(v, w)$;	$O(1)$
$pai[w] \leftarrow v$;	$O(1)$
DECREASEKEY ($Q, w, dist[w]$);	$O(\log_2 V)$

NB: Apenas as distâncias dos nós que estão na fila podem ser alteradas

Complexidade Temporal $O(|E| \log |V|)$, se for suportado por uma **heap de mínimo**. Como G é conexo, $|E| \geq |V| - 1$. O ciclo “Enquanto” domina a complexidade, sendo:

$$O\left(\sum_{v \in V} (1 + \log_2 |V| + |\text{Adjs}[v]| \log_2 |V|)\right) = O(|V| \log_2 |V| + |E| \log_2 |V|) = O(|E| \log_2 |V|)$$

Algoritmo de Prim – pseudocódigo

ALGORITMO PRIM(G, s) // [CLRS 23.2]	
Para cada $v \in V$ fazer { $pai[v] \leftarrow \text{NULL}$; $dist[v] \leftarrow \infty$; ok [v] \leftarrow false ; }	$\Theta(V)$
$dist[s] \leftarrow 0$;	$O(1)$
$Q \leftarrow \text{Mk_PQ_HEAPMIN}(dist, V)$;	$\Theta(V)$
$T \leftarrow \{\}$;	$O(1)$
Enquanto (PQ_NOT_EMPTY (Q)) fazer	
$v \leftarrow \text{EXTRACTMIN}(Q)$;	$O(\log_2 V)$
$T \leftarrow T \cup \{(pai[v], v)\}$; ok [v] \leftarrow true ; /* ok [v] indica se v já está em T */	$O(1)$
Para cada $w \in \text{Adjs}[v]$ fazer	
Se ok [w] = false e $d(v, w) < dist[w]$ então	$O(1)$
$dist[w] \leftarrow d(v, w)$;	$O(1)$
$pai[w] \leftarrow v$;	$O(1)$
DECREASEKEY ($Q, w, dist[w]$);	$O(\log_2 V)$

NB: Apenas as distâncias dos nós que estão na fila podem ser alteradas

Complexidade Temporal $O(|E| \log |V|)$, se for suportado por uma **heap de mínimo**. Como G é conexo, $|E| \geq |V| - 1$. O ciclo “Enquanto” domina a complexidade, sendo:

$$O\left(\sum_{v \in V} (1 + \log_2 |V| + |\text{Adjs}[v]| \log_2 |V|)\right) = O(|V| \log_2 |V| + |E| \log_2 |V|) = O(|E| \log_2 |V|)$$

Correção dos algoritmos de Prim e Kruskal

A correção dos algoritmos descritos resulta da propriedade seguinte

Propriedade das árvores geradoras de peso mínimo

Seja T uma árvore geradora mínima de um grafo $G = (V, E, d)$ não dirigido e conexo. Para toda a partição $\{V_1, V_2\}$ do conjunto de vértices V , a árvore T tem algum ramo $\langle v_1, v_2 \rangle$ tal que $v_1 \in V_1$, $v_2 \in V_2$, e $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$.

Prova: (por redução ao absurdo) Seja T uma árvore geradora mínima de G e suponhamos que $\{V_1, V_2\}$ é uma partição de V tal que T não contém nenhum ramo $\langle v_1, v_2 \rangle$ com $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$, $v_1 \in V_1$ e $v_2 \in V_2$. Seja $\langle v_1, v_2 \rangle$ um tal ramo. Como T é uma árvore geradora de G , existe um e um só caminho entre v_1 e v_2 em T . Esse caminho tem que ter algum ramo $\langle x, y \rangle$ com $x \in V_1$ e $y \in V_2$, pois, caso contrário, os nós em V_1 (respectivamente, em V_2) só estariam ligados em T a nós em V_1 (respectivamente, em V_2), e a árvore T não seria conexa (o que é absurdo). Note-se que é possível que ou $x = v_1$ ou $y = v_2$. Pela hipótese inicial, $d(x, y) > d(v_1, v_2)$. Por outro lado, se substituirmos $\langle x, y \rangle$ em T por $\langle v_1, v_2 \rangle$, o grafo resultante ainda é uma árvore geradora de G e tem "peso" menor do que a árvore T , o que contradiz o facto de T ser mínima. Portanto, a árvore T tem de ter algum dos ramos de menor peso nesse corte (por definição, o corte determinado pela partição $\{V_1, V_2\}$ de V é o conjunto de ramos que ligam vértices de V_1 a vértices de V_2).

Correção dos algoritmos de Prim e Kruskal

A correção dos algoritmos descritos resulta da propriedade seguinte

Propriedade das árvores geradoras de peso mínimo

Seja T uma árvore geradora mínima de um grafo $G = (V, E, d)$ não dirigido e conexo. Para toda a partição $\{V_1, V_2\}$ do conjunto de vértices V , a árvore T tem algum ramo $\langle v_1, v_2 \rangle$ tal que $v_1 \in V_1$, $v_2 \in V_2$, e $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$.

Prova: (por redução ao absurdo) Seja T uma árvore geradora mínima de G e suponhamos que $\{V_1, V_2\}$ é uma partição de V tal que T não contém nenhum ramo $\langle v_1, v_2 \rangle$ com $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$, $v_1 \in V_1$ e $v_2 \in V_2$. Seja $\langle v_1, v_2 \rangle$ um tal ramo. Como T é uma árvore geradora de G , existe um e um só caminho entre v_1 e v_2 em T . Esse caminho tem que ter algum ramo $\langle x, y \rangle$ com $x \in V_1$ e $y \in V_2$, pois, caso contrário, os nós em V_1 (respectivamente, em V_2) só estariam ligados em T a nós em V_1 (respectivamente, em V_2), e a árvore T não seria conexa (o que é absurdo). Note-se que é possível que ou $x = v_1$ ou $y = v_2$. Pela hipótese inicial, $d(x, y) > d(v_1, v_2)$. Por outro lado, se substituirmos $\langle x, y \rangle$ em T por $\langle v_1, v_2 \rangle$, o grafo resultante ainda é uma árvore geradora de G e tem “peso” menor do que a árvore T , o que contradiz o facto de T ser mínima. Portanto, a árvore T tem de ter algum dos ramos de menor peso nesse corte (por definição, o corte determinado pela partição $\{V_1, V_2\}$ de V é o conjunto de ramos que ligam vértices de V_1 a vértices de V_2).

Correção dos algoritmos de Kruskal e de Prim

Da propriedade anterior conclui-se que:

- no algoritmo de Prim, é *seguro* ligar o vértice v à sub-árvore já construída. Em cada iteração do ciclo “Enquanto”, V_1 seria o conjunto dos vértices que já estão na sub-árvore e V_2 seria o conjunto dos restantes.

Para cada $v \in V_2$, o valor de $dist[v]$ é o custo dos ramos mais leves com extremidade em v e que estão no **corte definido por** $\{V_1, V_2\}$. Este invariante é preservado pelo ciclo.

- no algoritmo de Kruskal, quando $\langle u, v \rangle$ é escolhido para ligar duas componentes, se tomarmos V_1 como os nós da componente que contém u e V_2 como os restantes nós, podemos concluir que $\langle u, v \rangle$ é *seguro*.

Alguma árvore geradora mínima contém $\langle u, v \rangle$, pois este ramo tem peso mínimo no **corte definido por** $\{V_1, V_2\}$.

Filas de prioridade: **heapMin.h** e **heapMax.h**

```
typedef struct qnode {
    int vert, vertkey;
} QNODE;

typedef struct heapMin {
    int sizeMax, size;
    QNODE *a;      // fila -- heap de minimo -- array de pares (nó do grafo e chave)
    int *pos_a;    // array que indica a posição de cada nó do grafo na fila a[]
} HEAPMIN;

HEAPMIN *build_heap_min(int v[], int n);
int extractMin(HEAPMIN *q); // retorna v
void decreaseKey(int v, int newkey, HEAPMIN *q);
int heap_isEmpty(HEAPMIN *q); // retorna 1 ou 0

void insert(int v, int key, HEAPMIN *q);
void write_heap(HEAPMIN *q);
void destroy_heap(HEAPMIN *q);

#define POSINVALIDA 0
#define LEFT(i) (2*(i))      // indice do filho esquerdo de a[i] em a
#define RIGHT(i) (2*(i)+1)   // indice do filho direito de a[i] em a
#define PARENT(i) ((i)/2)    // indice do pai de a[i] em a
```

heapMin.h: função build_heap_min

```

HEAPMIN *build_heap_min(int vec[], int n){
    // supor que vetor vec[] guarda elementos nas posições 1 a n
    // cria heapMin correspondente em tempo  $O(n)$ 
    HEAPMIN *q = (HEAPMIN *)malloc(sizeof(HEAPMIN));
    int i;
    q -> a = (QNODE *) malloc(sizeof(QNODE)*(n+1));
    q -> pos_a = (int *) malloc(sizeof(int)*(n+1));
    q -> sizeMax = n; // posicao 0 nao vai ser ocupada
    q -> size = n;
    for (i=1; i<= n; i++) {
        q -> a[i].vert = i;
        q -> a[i].vertkey = vec[i];
        q -> pos_a[i] = i; // posicao inicial do elemento i na heap
    }

    for (i=n/2; i>=1; i--)
        heapify(i,q);
    return q;
}

```


heapMin.h – Função heapify

```
static void heapify(int i,HEAPMIN *q) {  
    // para heap de minimo  
    int l, r, smallest;  
    l = LEFT(i);  
    if (l > q -> size) l = i;  
    r = RIGHT(i);  
    if (r > q -> size) r = i;  
  
    smallest = i;  
    if (compare(l,smallest,q) < 0)  
        smallest = l;  
    if (compare(r,smallest,q) < 0)  
        smallest = r;  
  
    if (i != smallest) {  
        swap(i,smallest,q);  
        heapify(smallest,q);  
    }  
}
```

heapMin.h: funções swap e decreaseKey

```
static void swap(int i,int j,HEAPMIN *q){
    QNODE aux;
    q -> pos_a[q -> a[i].vert] = j;
    q -> pos_a[q -> a[j].vert] = i;
    aux = q -> a[i];
    q -> a[i] = q -> a[j];
    q -> a[j] = aux;
}

void decreaseKey(int vertv, int newkey, HEAPMIN *q){
    int i = q -> pos_a[vertv];
    q -> a[i].vertkey = newkey;

    while(i > 1 && compare(i,PARENT(i),q) < 0){
        swap(i,PARENT(i),q);
        i = PARENT(i);
    }
}
```

heapMin.h – Função extractMin

```
int extractMin(HEAPMIN *q) {  
    int vertv = q -> a[1].vert;  
    swap(1,q->size,q);  
    q -> pos_a[vertv] = POSINVALIDA; // assinala vertv como removido  
    q -> size--;  
    heapify(1,q);  
    return vertv;  
}
```

Caminhos mínimos em grafos com pesos positivos

Seja $G = (V, E, d)$ um grafo dirigido, finito e com pesos (*distâncias* ou *valores*) positivos associados aos ramos, $d(u, v) > 0$, para todo $(u, v) \in E$.

- A **distância associada a um percurso de u para v** é a soma das distâncias associadas aos ramos que constituem o percurso.
- Assumimos que a distância mínima de um nó do grafo a si mesmo é zero.

Dependendo da aplicação, podemos querer encontrar um:

- caminho mínimo de s para t , para **um par** $(s, t) \in V \times V$, com $s \neq t$;
- caminho mínimo **de s para cada um** dos outros nós, para $s \in V$ **fixo**;
- caminho mínimo de s para t , para **todos os pares** $(s, t) \in V \times V$, $s \neq t$.

Algoritmo de Dijkstra (1959)

Restrição de aplicabilidade: Os valores nos ramos têm de ser positivos.

Caminhos mínimos a partir de um nó origem s :

ALGORITMODIJKSTRA(G, s)

```

1. Para cada  $v \in G.V$  fazer {  $pai[v] \leftarrow \text{NULL}$ ;  $dist[v] \leftarrow \infty$ ; }
2.  $dist[s] \leftarrow 0$ ;
3.  $Q \leftarrow \text{MK\_PQ\_HEAPMIN}(dist, G.V)$ ;
4. Enquanto ( $\text{PQ\_NOT\_EMPTY}(Q)$ ) fazer
5.      $v \leftarrow \text{EXTRACTMIN}(Q)$ ;
6.     Para cada  $w \in G.Adjs[v]$  fazer
7.         Se  $dist[v] + G.d(v, w) < dist[w]$  então
8.              $dist[w] \leftarrow dist[v] + G.d(v, w)$ ;
9.              $pai[w] \leftarrow v$ ;
10.    DECREASEKEY( $Q, w, dist[w]$ );
```

Melhoramento: Sair se $dist[v] = \infty$ na linha 5. (pois não há caminho de s os nós em $Q \cup \{v\}$)

Caminho mínimo de s para t , com s e t fixos: sair quando t é extraído de Q , colocando Se ($v = t$) então retornar; entre as linhas 5 e 6.

Algoritmo de Dijkstra (1959)

Restrição de aplicabilidade: Os valores nos ramos têm de ser positivos.

Caminhos mínimos a partir de um nó origem s :

ALGORITMODIJKSTRA(G, s)

```

1. Para cada  $v \in G.V$  fazer {  $\text{pai}[v] \leftarrow \text{NULL}$ ;  $\text{dist}[v] \leftarrow \infty$ ; }
2.  $\text{dist}[s] \leftarrow 0$ ;
3.  $Q \leftarrow \text{MK\_PQ\_HEAPMIN}(\text{dist}, G.V)$ ;
4. Enquanto ( $\text{PQ\_NOT\_EMPTY}(Q)$ ) fazer
5.      $v \leftarrow \text{EXTRACTMIN}(Q)$ ;
6.     Para cada  $w \in G.Adjs[v]$  fazer
7.         Se  $\text{dist}[v] + G.d(v, w) < \text{dist}[w]$  então
8.              $\text{dist}[w] \leftarrow \text{dist}[v] + G.d(v, w)$ ;
9.              $\text{pai}[w] \leftarrow v$ ;
10.    DECREASEKEY( $Q, w, \text{dist}[w]$ );
```

Melhoramento: Sair se $\text{dist}[v] = \infty$ na linha 5. (pois não há caminho de s os nós em $Q \cup \{v\}$)

Caminho mínimo de s para t , com s e t fixos: sair quando t é extraído de Q , colocando Se ($v = t$) então retornar; entre as linhas 5 e 6.

Complexidade temporal do algoritmo de Dijkstra

ALGORITMO DIJKSTRA(G, s)

```

1. Para cada  $v \in G.V$  fazer {  $pai[v] \leftarrow \text{NULL}$ ;  $dist[v] \leftarrow \infty$ ; }
2.  $dist[s] \leftarrow 0$ ;
3.  $Q \leftarrow \text{MK\_PQ\_HEAPMIN}(dist, G.V)$ ;
4. Enquanto ( $\text{PQ\_NOT\_EMPTY}(Q)$ ) fazer
5.    $v \leftarrow \text{EXTRACTMIN}(Q)$ ;
6.   Para cada  $w \in G.Adjs[v]$  fazer
7.     Se  $dist[v] + G.d(v, w) < dist[w]$  então
8.        $dist[w] \leftarrow dist[v] + G.d(v, w)$ ;
9.        $pai[w] \leftarrow v$ ;
10.     $\text{DECREASEKEY}(Q, w, dist[w])$ ;

```

Se G for dado por **listas de adjacências** e a fila de prioridade Q for suportada por uma **heap de mínimo**, tem complexidade temporal $O((|V| + |E|) \log_2 |V|)$, pois é dominada pelo ciclo “Enquanto”:

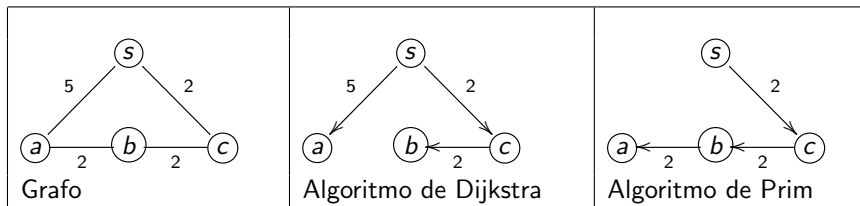
$$O\left(\sum_{v \in V} (1 + \log_2 |V| + |Adj[s][v]| \log_2 |V|)\right) = O((|V| + |E|) \log_2 |V|).$$

Mantemos a expressão assim pois não sabemos qual é a ordem de grandeza de $|E|$ relativamente a $|V|$.

Árvores de peso mínimo / Árvores de caminhos mínimos

O algoritmo de Dijkstra pode ser aplicado a grafos $G = (V, E, d)$ **não dirigidos** (que podem ser vistos como grafos dirigidos simétricos). Quando G é conexo, **a árvore dos caminhos mínimos com origem em s contém todos os nós mas nem sempre é uma árvore geradora mínima de G** . Consequentemente,

o algoritmo de Prim não pode ser usado para determinar os caminhos mínimos com origem em s .



Prova de correção do algoritmo de Dijkstra

Usamos $\delta(s, v)$ para denotar a **distância mínima** de s a v em G , para $v \in V$.

O algoritmo de Dijkstra mantém o invariante seguinte, para $k \geq 1$.

No **final da iteração k do ciclo “Enquanto”**, seja Q_k o conjunto de nós que estão na fila Q e $M_k = V \setminus Q_k$ o conjunto de nós que já saíram de Q . Tem-se:

- ① $dist[r] = \delta(s, r)$, para todo $r \in M_k$;
 - ② $dist[r]$ é a distância mínima de s a r em G se os percursos só puderem passar por vértices de $M_k \cup \{r\}$, para todo $r \in Q_k$.
-

Prova (por indução sobre $k \geq 1$):

(*Caso de base*) Para $k = 1$, o nó que **sai de Q** (linha 5) é s e, no bloco 6-10, $dist$ é atualizado, ficando $dist[w] = d(s, w)$, para todo $w \in Adj[s]$ (e mantém $dist[r] = \infty$ para os restantes $r \neq s$). Logo, as condições 1. e 2. verificam-se, já que, $M_1 = \{s\}$, $dist[s] = 0 = \delta(s, s)$, por definição, e para $r \in Q_1 = V \setminus \{s\}$, o caminho mínimo de s para r que só pode passar por $M_1 \cup \{r\} = \{s, r\}$ é dado por (s, r) ou não existe, para $r \in V \setminus \{s\}$.

Prova de correção do algoritmo de Dijkstra

Usamos $\delta(s, v)$ para denotar a **distância mínima** de s a v em G , para $v \in V$.

O algoritmo de Dijkstra mantém o invariante seguinte, para $k \geq 1$.

No **final da iteração k do ciclo “Enquanto”**, seja Q_k o conjunto de nós que estão na fila Q e $M_k = V \setminus Q_k$ o conjunto de nós que já saíram de Q . Tem-se:

- ① $dist[r] = \delta(s, r)$, para todo $r \in M_k$;
 - ② $dist[r]$ é a distância mínima de s a r em G se os percursos só puderem passar por vértices de $M_k \cup \{r\}$, para todo $r \in Q_k$.
-

Prova (por indução sobre $k \geq 1$):

(*Caso de base*) Para $k = 1$, o nó que **sai de Q** (linha 5) é s e, no bloco 6-10, $dist$ é atualizado, ficando $dist[w] = d(s, w)$, para todo $w \in Adj[s]$ (e mantém $dist[r] = \infty$ para os restantes $r \neq s$). Logo, as condições 1. e 2. verificam-se, já que, $M_1 = \{s\}$, $dist[s] = 0 = \delta(s, s)$, por definição, e para $r \in Q_1 = V \setminus \{s\}$, o caminho mínimo de s para r que só pode passar por $M_1 \cup \{r\} = \{s, r\}$ é dado por (s, r) ou não existe, para $r \in V \setminus \{s\}$.

Prova de correção do algoritmo de Dijkstra (cont.)

(cont.) Prova (por indução sobre $k \geq 1$):

(Hereditariedade) Suponhamos, como hipótese de indução, que o invariante se verifica no final da iteração k , para $k \geq 1$ fixo, e que $\mathcal{M}_k \neq V$, ou seja, $Q \neq \{\}$. Vamos mostrar que então o invariante se verifica no final da iteração $k + 1$.

Iremos analisar os dois casos seguintes:

- (Caso A) não existem vértices em Q_k acessíveis de s
- (Caso B) existem vértices em Q_k acessíveis de s

Caso A

Todo $r \in Q_k$ está, por convenção, a distância mínima ∞ de s e, de acordo com o invariante, no final da iteração k , tem-se $dist[r] = \infty$ para todo $r \in Q_k$ (como se definiu inicialmente). O vértice v que sai da fila na iteração $k + 1$ tem $dist[v] = \infty$ e, assim, como $dist[v] + d(v, w) = \infty + d(v, w) = \infty$, não altera o valor de $dist[w]$, para nenhum $w \in Adjs[v]$. Logo, todos os vértices em $r \in Q_{k+1}$ se manterão com $dist[r] = \infty$, e a condição 2. do invariante mantém-se.

Prova de correção do algoritmo de Dijkstra (cont.)

(cont.) Prova (por indução sobre $k \geq 1$):

Seja $\hat{d}(\gamma)$ o comprimento de um percurso γ , ou seja, $\hat{d}(\gamma) = \sum_{(x,y) \in \gamma} d(x,y)$.

Caso B

- Seja $w \in Q_k$ um vértice que se encontra a distância mínima de s , ou seja tal que $\delta(s, w) = \min\{\delta(s, r) \mid r \in Q_k\}$. Seja $\gamma_{s,w}$ um caminho mínimo de s para w em G . Logo, tem-se $\hat{d}(\gamma_{s,w}) = \delta(s, w)$.
- Se $\gamma_{s,w}$ só tem um ramo, então $w \in Adjs[s]$, e do (caso de base, $k = 1$) segue $dist[w] = \delta(s, w) = \hat{d}(\gamma_{s,w})$.
- Se $\gamma_{s,w}$ tem mais do que um ramo, seja u o vértice que precede w no caminho $\gamma_{s,w}$, e $\gamma_{s,u}$ o sub-caminho até u .
 - $\delta(s, u) = \hat{d}(\gamma_{s,u})$
 - $u \notin Q_k$ pois $\delta(s, w) = \delta(s, u) + d(u, w)$ implica que $\delta(s, u) < \delta(s, w)$. Se u estivesse em Q_k seria escolhido em vez de w . Então $u \in M_k$, e, pela hipótese de indução, $dist[u] = \delta(s, u)$ e $dist[w] \leq \hat{d}(\gamma_{s,w})$. Logo, $dist[w] = \hat{d}(\gamma_{s,w}) = \delta(s, w)$.

Prova de correção do algoritmo de Dijkstra (cont.)

Caso B (cont.)

- Portanto, no algoritmo de Dijkstra, o **vértice v que se retira de Q na iteração $k + 1$** satisfaz $dist[v] = \delta(s, v)$ (é um vértice de Q_k que está a distância mínima de s , dado que $dist[v]$ não seria alterado em nenhuma das iterações seguintes). Como $\mathcal{M}_{k+1} = \mathcal{M}_k \cup \{v\}$, a condição 1. do invariante verifica-se no final da iteração $k + 1$.
 - Importa observar que se $r \in Adj[s] \cap \mathcal{M}_k$, o valor de $dist[r]$ não pode ser reduzido na iteração $k + 1$ pois, pela hipótese de indução, $dist[r] = \delta(s, r)$ e, portanto, $dist[v] + d(v, r) \geq dist[r]$, por definição de caminho mínimo.
- Resta ver que, para todo $r \in \mathcal{Q}_{k+1}$, no final da iteração $k + 1$, o valor de $dist[r]$ é a distância mínima de s a r se os percursos só puderem passar por vértices de $\mathcal{M}_{k+1} \cup \{r\}$. Tais percursos são caminhos sendo:
 - ou *caminhos mínimos de s para r que não passam por $Q_k \setminus \{r\}$*
Nesse caso, pela hipótese de indução, $dist[r]$ é já a distância mínima de s a r com essa restrição.
 - ou *caminhos mínimos que passam em v mas não em $Q_k \setminus \{r, v\}$*
Notar que $Q_k \setminus \{r, v\} = Q_{k+1} \setminus \{r\}$. Se se verificar o segundo caso (mas não o primeiro), tal caminho mínimo passa por v e, por ser mínimo terá de ser da forma $\Gamma_{s,v}, (v, r)$, para $\Gamma_{s,v}$ mínimo. Mas, r é adjacente a v e $dist[v] + d(v, r) = \delta(s, v) + d(v, r) = \hat{d}(\Gamma_{s,v}, (v, r)) < dist[r]$. Portanto, $dist[r]$ é atualizado no bloco 6-10 na iteração $k + 1$ ficando com $\hat{d}(\Gamma_{s,v}, (v, r))$.

Caminhos de capacidade máxima (adaptação do Algoritmo de Dijkstra)

Seja $G = (V, E, c)$ um grafo dirigido, finito, $c(u, v) \geq 0$ indica a **capacidade** do ramo (u, v) . A **capacidade de um percurso** é o **mínimo** das capacidades dos ramos que constituem o percurso.

Problema:

Para um nó origem s , determinar um *percurso com capacidade máxima* de s para v , para cada $v \neq s$.

CAMINHOSCAPACIDADEMAXIMA(G, s)

1. Para cada $v \in V$ fazer { $pai[v] \leftarrow \text{NULL}$; $cap[v] \leftarrow 0$;}
2. $cap[s] \leftarrow \infty$;
3. $Q \leftarrow \text{MK_PQ_HEAPMAX}(cap, V)$;
4. Enquanto ($\text{PQ_NOT_EMPTY}(Q)$) fazer
5. $v \leftarrow \text{EXTRACTMAX}(Q)$;
6. Para cada $w \in \text{Adjs}[v]$ fazer
7. Se $\min(cap[v], c(v, w)) > cap[w]$ então
8. $cap[w] \leftarrow \min(cap[v], c(v, w))$;
9. $pai[w] \leftarrow v$;
10. $\text{INCREASEKEY}(Q, w, cap[w])$;

Caminhos de capacidade máxima (adaptação do Algoritmo de Dijkstra)

• Complexidade temporal

- Se G for dado por **listas de adjacências** e a fila de prioridade Q for suportada por uma **heap de máximo**, tem complexidade temporal $O((|V| + |E|) \log_2 |V|)$, como o algoritmo de Dijkstra.

- **Correção:** o ciclo “Enquanto” preserva o invariante seguinte, para todo $k \geq 1$: sendo Q_k o conjunto de nós que estão na fila Q e $M_k = V \setminus Q_k$ o conjunto de nós que já saíram de Q , no final da iteração k , tem-se
 - 1 para $r \in M_k$, o valor $cap[r]$ é a capacidade máxima dos percursos de s para r em G , para todo $r \in M_k$;
 - 2 para $r \in Q_k$, o valor $cap[r]$ é a capacidade máxima dos percursos de s para r em G se os percursos só puderem passar por nós de $M_k \cup \{r\}$.
- **Propriedade que explora:** um percurso γ_{st} de capacidade máxima **não tem de ter subestrutura ótima**. Mas, é verdade que se $\gamma_{st} = \gamma_{sv} \gamma_{vt}$, para algum v , **podemos substituir** cada um dos percursos γ_{sv} e γ_{vt} por caminhos γ_{sv}^* e γ_{vt}^* de capacidade máxima.

Caminhos de capacidade máxima em grafos não dirigidos

Propriedade

Se $G = (V, E, c)$ for um grafo não dirigido e conexo, a árvore geradora de **peso máximo criada a partir da raiz s** por adaptação do algoritmo de Prim contém um caminho de capacidade máxima de s para v , para cada $v \in V \setminus \{s\}$.

- Por isso, em instâncias deste tipo, o algoritmo de Prim (adaptado para obter árvores de peso máximo) seria uma alternativa ao que apresentámos acima.
- Esta propriedade resulta da definição de caminho de capacidade máxima e da seguinte propriedade estrutural das árvores de suporte de peso máximo:

Seja T uma árvore geradora de peso máximo de um grafo $G = (V, E, d)$ não dirigido e conexo. Qualquer que seja a partição $\{V_1, V_2\}$ do conjunto de vértices V , a árvore T tem algum ramo $\langle v_1, v_2 \rangle$ com $v_1 \in V_1$ e $v_2 \in V_2$ e tal que $d(v_1, v_2) = \max\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$.

Algoritmo de Floyd-Warshall

Problema:

Determinar o comprimento do caminho mínimo de s para t , para **todos os pares** $(s, t) \in V \times V$, $s \neq t$.

- Pode ser resolvido usando o algoritmo de Dijkstra
 - Para cada nó v_i (origem), aplicar o algoritmo de Dijkstra para determinar D_{ij}^* , para todo j . Complexidade: $O(|V|(|E| + |V|) \log_2 |V|)$.
 - Para grafos densos, com $|E| \in \Theta(|V|^2)$, seria $O(n^3 \log_2 n)$.
- Mas, o **algoritmo de Floyd-Warshall** (1962), tem complexidade $\Theta(n^3)$.

Supõe-se que inicialmente $D_{ij} = 0$, $D_{ij} = d(i, j)$, se $(i, j) \in E$; e, caso contrário, $D_{ij} = \infty$ se $i \neq j$

ALGORITMOFLOYD-WARSHALL(D, n)

Para $k \leftarrow 1$ até n fazer

Para $i \leftarrow 1$ até n fazer

Para $j \leftarrow 1$ até n fazer

Se $D[i, j] > D[i, k] + D[k, j]$ então $D[i, j] \leftarrow D[i, k] + D[k, j]$;

Algoritmo de Floyd-Warshall

Problema:

Determinar o comprimento do caminho mínimo de s para t , para **todos os pares** $(s, t) \in V \times V$, $s \neq t$.

- Pode ser resolvido usando o algoritmo de Dijkstra
 - Para cada nó v_i (origem), aplicar o algoritmo de Dijkstra para determinar D_{ij}^* , para todo j . Complexidade: $O(|V|(|E| + |V|) \log_2 |V|)$.
 - Para grafos densos, com $|E| \in \Theta(|V|^2)$, seria $O(n^3 \log_2 n)$.
- Mas, o **algoritmo de Floyd-Warshall** (1962), tem complexidade $\Theta(n^3)$.

Supõe-se que inicialmente $D_{ii} = 0$, $D_{ij} = d(i, j)$, se $(i, j) \in E$; e, caso contrário, $D_{ij} = \infty$ se $i \neq j$

ALGORITMOFLOYD-WARSHALL(D, n)

Para $k \leftarrow 1$ até n fazer

Para $i \leftarrow 1$ até n fazer

Para $j \leftarrow 1$ até n fazer

Se $D[i, j] > D[i, k] + D[k, j]$ então $D[i, j] \leftarrow D[i, k] + D[k, j]$;