Algoritmos em Grafos

Ana Paula Tomás

Desenho e Análise de Algoritmos
Universidade do Porto

Abril 2024

Árvores geradoras com peso mínimo/máximo

Problema

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 $d : E \to \mathbb{R}_0^+$ 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

Ideia do Algoritmo de Kruskal

- 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 $T \leftarrow T \cup \{<u, v>\}$; $\mathcal{C} \leftarrow (\mathcal{C} \setminus \{C_u, C_v\}) \cup \{C_u \cup C_v\}$;

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. $\mathcal{C} \leftarrow$ Init_Singletons(V);
4. Enquanto ($|T| \neq |V| - 1 \wedge$ QueueIsEmpty(Q) = `false`) fazer
5. $\quad \langle u, v \rangle \leftarrow$ Dequeue(Q);
6. $\quad$ Se FindSet($u, \mathcal{C}$) $\neq$ FindSet($v, \mathcal{C}$) então
7. $\quad\quad T \leftarrow T \cup \{\langle u, v \rangle\}$;
8. $\quad\quad$ Union($u, v, \mathcal{C}$);

A condição QueueIsEmpty(Q) = `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

Ideia do algoritmo de Prim

- 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

AlgoritmoPrim(G, s) // [CLRS 23.2]	
Para cada $v \in V$ fazer { $pai[v] \leftarrow$ NULL; $dist[v] \leftarrow \infty$; **ok**[**v**] $\leftarrow$ `false`; }	$\Theta(\|V\|)$
$dist[s] \leftarrow 0$;	$O(1)$
$Q \leftarrow$ Mk_PQ_Heapmin($dist, \|V\|$);	$\Theta(\|V\|)$
Enquanto (PQ_Not_Empty(Q)) fazer	
$v \leftarrow$ ExtractMin(Q);	$O(\log_2 \|V\|)$
ok[**v**] $\leftarrow$ `true`; /* $ok[v]$ indica se v já está na árvore */	$O(1)$
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($Q, w, dist[w]$);	$O(\log_2 \|V\|)$

Algoritmo de Prim – pseudocódigo

AlgoritmoPrim(G, s) // [CLRS 23.2]	
Para cada $v \in V$ fazer { $pai[v] \leftarrow$ NULL; $dist[v] \leftarrow \infty$; **ok**[**v**] $\leftarrow$ false; }	$\Theta(\|V\|)$
$dist[s] \leftarrow 0$;	$O(1)$
$Q \leftarrow$ Mk_PQ_Heapmin($dist, \|V\|$);	$\Theta(\|V\|)$
Enquanto (PQ_Not_Empty(Q)) fazer	
$v \leftarrow$ ExtractMin(Q);	$O(\log_2 \|V\|)$
ok[**v**] $\leftarrow$ true; /* $ok[v]$ indica se v já está na árvore */	$O(1)$
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

A árvore tem raíz s e ramos $\langle pai[v], v \rangle$ para os restantes nós.

Algoritmo de Prim – pseudocódigo

AlgoritmoPrim(G, s) // [CLRS 23.2]	
Para cada $v \in V$ fazer { $pai[v] \leftarrow$ NULL; $dist[v] \leftarrow \infty$; **ok[v]** $\leftarrow$ false; }	$\Theta(\|V\|)$
$dist[s] \leftarrow 0$;	$O(1)$
$Q \leftarrow$ Mk_PQ_Heapmin($dist, \|V\|$);	$\Theta(\|V\|)$
Enquanto (PQ_Not_Empty(Q)) fazer	
$v \leftarrow$ ExtractMin(Q);	$O(\log_2 \|V\|)$
ok[v] $\leftarrow$ true; /* $ok[v]$ indica se v já está na árvore */	$O(1)$
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

A árvore tem raíz s e ramos $\langle pai[v], v \rangle$ para os restantes nós.

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:

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

Algoritmo de Prim para árvore geradora de peso **máximo**

AlgoritmoPrim(G, s) // para peso máximo	
Para cada $v \in V$ fazer { $pai[v] \leftarrow$ NULL; $dist[v] \leftarrow 0$; **ok**[**v**] $\leftarrow$ false; }	$\Theta(\|V\|)$
$dist[s] \leftarrow \infty$;	$O(1)$
$Q \leftarrow$ Mk_PQ_Heapmax($dist, \|V\|$);	$\Theta(\|V\|)$
Enquanto (PQ_Not_Empty(Q)) fazer	
$v \leftarrow$ ExtractMax(Q);	$O(\log_2 \|V\|)$
ok[**v**] $\leftarrow$ true;	$O(1)$
Para cada $w \in Adjs[v]$ fazer	
Se **ok**[**w**] = false e $d(v, w) > dist[w]$ então	$O(1)$
$dist[w] \leftarrow d(v, w)$;	$O(1)$
$pai[w] \leftarrow v$;	$O(1)$
IncreaseKey($Q, w, dist[w]$);	$O(\log_2 \|V\|)$

Algoritmo de Prim para árvore geradora de peso **máximo**

AlgoritmoPrim(G, s) // para peso máximo	
Para cada $v \in V$ fazer { $pai[v] \leftarrow$ NULL; $dist[v] \leftarrow 0$; **ok**[**v**] $\leftarrow$ false; }	$\Theta(\|V\|)$
$dist[s] \leftarrow \infty$;	$O(1)$
$Q \leftarrow$ Mk_PQ_Heapmax($dist, \|V\|$);	$\Theta(\|V\|)$
Enquanto (PQ_Not_Empty(Q)) fazer	
$v \leftarrow$ ExtractMax(Q);	$O(\log_2 \|V\|)$
ok[**v**] $\leftarrow$ true;	$O(1)$
Para cada $w \in Adjs[v]$ fazer	
Se **ok**[**w**] = false e $d(v,w) > dist[w]$ então	$O(1)$
$dist[w] \leftarrow d(v,w)$;	$O(1)$
$pai[w] \leftarrow v$;	$O(1)$
IncreaseKey($Q, w, dist[w]$);	$O(\log_2 \|V\|)$

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

Correção dos algoritmos de Prim e Kruskal

A correção dos algoritmos descritos resulta das propriedades seguintes

Propriedade I 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 $d(v_1, v_2) = \min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$.

Correção dos algoritmos de Prim e Kruskal

A correção dos algoritmos descritos resulta das propriedades seguintes

Propriedade I 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

Propriedade II 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, se existirem dois ou mais ramos no corte $\{V_1, V_2\}$ com peso $\min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y \rangle \in E\}$, então ou T contém todos esses ramos ou qualquer um deles pode ser substituir um outro ramo ramo nesse corte com peso igual.

Correção dos algoritmos de Prim e Kruskal

Propriedade II 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, se existirem dois ou mais ramos no corte $\{V_1, V_2\}$ com peso $\min\{d(x, y) \mid x \in V_1, y \in V_2, \langle x, y\rangle \in E\}$, então ou T contém todos esses ramos ou qualquer um deles pode ser substituir um outro ramo ramo nesse corte com peso igual.

Prova:

Seja $\langle v_1, v_2\rangle$ um tal ramo não incluído na árvore T. Numa árvore, qualquer par de nós está ligado por um caminho único. Assim, existia um caminho γ em T de v_1 para v_2. Ao acrescentar $\langle v_1, v_2\rangle$ forma-se um ciclo. Se substituirmos um ramo $\langle x, y\rangle$ de γ com $x \in V_1$ e $y \in V_2$ por $\langle v_1, v_2\rangle$ obtemos uma árvore de suporte. Se todos os $\langle x, y\rangle$ nessas condições tiverem peso superior $d(v_1, v_2)$ então a árvore T não seria ótima. Portanto, existe algum $\langle x, y\rangle$ com peso igual a $d(v_1, v_2)$. A árvore T' resultante teria peso igual ao de T.

Correção dos algoritmos de Kruskal e de Prim

Das propriedades anteriores 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**

- Uma *heap* é suportada por um vetor e pode ser vista como uma árvore binária **completa** (apenas o último nível pode não estar completo). Numa *heap* de mínimo, a **chave** de qualquer nó é **menor ou igual** que a chave dos seus filhos. Numa *heap* de máximo a chave de qualquer nó é **maior ou igual** que a chave dos seus filhos.

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

- Uma *heap* é suportada por um vetor e pode ser vista como uma árvore binária **completa** (apenas o último nível pode não estar completo). Numa *heap* de mínimo, a **chave** de qualquer nó é **menor ou igual** que a chave dos seus filhos. Numa *heap* de máximo a chave de qualquer nó é **maior ou igual** que a chave dos seus filhos.
- Numa *heap* de mínimo (máximo), o elemento que tem chave mínima (máxima) está na **posição de índice 1** do vetor (i.e., na raíz da árvore).

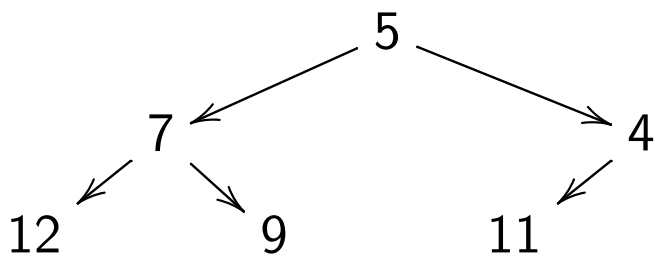
Filas de prioridade: **heapMin.h** e **heapMax.h**

- Uma *heap* é suportada por um vetor e pode ser vista como uma árvore binária **completa** (apenas o último nível pode não estar completo). Numa *heap* de mínimo, a **chave** de qualquer nó é **menor ou igual** que a chave dos seus filhos. Numa *heap* de máximo a chave de qualquer nó é **maior ou igual** que a chave dos seus filhos.
- Numa *heap* de mínimo (máximo), o elemento que tem chave mínima (máxima) está na **posição de índice 1** do vetor (i.e., na raíz da árvore).O **pai** do nó i está na posição $i/2$. O **filho esquerdo** do nó i está na posição $2i$. O **filho direito** do nó i está na posição $2i + 1$.

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

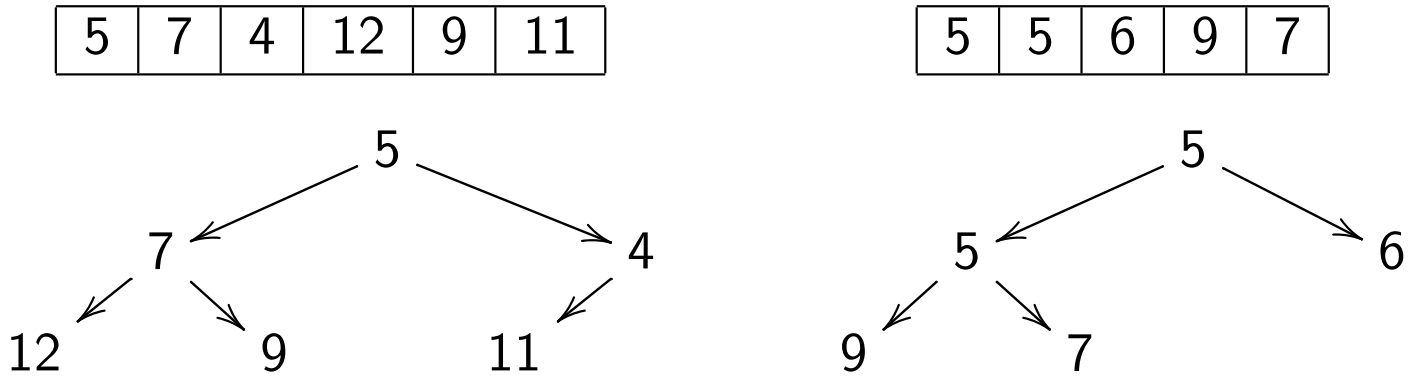
- Uma *heap* é suportada por um vetor e pode ser vista como uma árvore binária **completa** (apenas o último nível pode não estar completo). Numa *heap* de mínimo, a **chave** de qualquer nó é **menor ou igual** que a chave dos seus filhos. Numa *heap* de máximo a chave de qualquer nó é **maior ou igual** que a chave dos seus filhos.
- Numa *heap* de mínimo (máximo), o elemento que tem chave mínima (máxima) está na **posição de índice 1** do vetor (i.e., na raíz da árvore).O **pai** do nó i está na posição $i/2$. O **filho esquerdo** do nó i está na posição $2i$. O **filho direito** do nó i está na posição $2i + 1$.
- Exemplos de *heaps* de mínimo:

5	7	4	12	9	11



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

- Uma *heap* é suportada por um vetor e pode ser vista como uma árvore binária **completa** (apenas o último nível pode não estar completo). Numa *heap* de mínimo, a **chave** de qualquer nó é **menor ou igual** que a chave dos seus filhos. Numa *heap* de máximo a chave de qualquer nó é **maior ou igual** que a chave dos seus filhos.
- Numa *heap* de mínimo (máximo), o elemento que tem chave mínima (máxima) está na **posição de índice 1** do vetor (i.e., na raíz da árvore).O **pai** do nó i está na posição $i/2$. O **filho esquerdo** do nó i está na posição $2i$. O **filho direito** do nó i está na posição $2i + 1$.
- Exemplos de *heaps* de mínimo:



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

- Utilizamos *heaps* de mínimo (ou de máximo), por exemplo, nos algoritmos de Prim e de Dijkstra, para representar **filas de prioridade**.

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

- Utilizamos *heaps* de mínimo (ou de máximo), por exemplo, nos algoritmos de Prim e de Dijkstra, para representar **filas de prioridade**. Nestes algoritmos, a chave de v é $dist[v]$. Cada nó da *heap* guarda $dist[v]$ e também v (sendo v o identificador de um nó do grafo). Na implementação disponibilizada, cada nó é do tipo `QNODE` e a fila de prioridade é do tipo `HEAPMIN`:

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

- Utilizamos *heaps* de mínimo (ou de máximo), por exemplo, nos algoritmos de Prim e de Dijkstra, para representar **filas de prioridade**. Nestes algoritmos, a chave de v é $dist[v]$. Cada nó da *heap* guarda $dist[v]$ e também v (sendo v o identificador de um nó do grafo). Na implementação disponibilizada, cada nó é do tipo `QNODE` e a fila de prioridade é do tipo `HEAPMIN`:

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

```
typedef struct heapMin {
  int sizeMax, size;
  QNODE *a;
  int *pos_a;
} HEAPMIN;
```

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

- Utilizamos *heaps* de mínimo (ou de máximo), por exemplo, nos algoritmos de Prim e de Dijkstra, para representar **filas de prioridade**. Nestes algoritmos, a chave de v é $dist[v]$. Cada nó da *heap* guarda $dist[v]$ e também v (sendo v o identificador de um nó do grafo). Na implementação disponibilizada, cada nó é do tipo `QNODE` e a fila de prioridade é do tipo `HEAPMIN`:

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

```
typedef struct heapMin {
  int sizeMax, size;
  QNODE *a;
  int *pos_a;
} HEAPMIN;
```

- Nos algoritmos de Prim e de Dijkstra, a chave de v pode variar (ser reduzida ou aumentada, conforme a aplicação). Para rapidamente aceder ao nó que tem v na heap, a estrutura `HEAPMIN` mantém um array `pos_a[]` que indica a posição de cada **nó do grafo** no vetor `a[]` (que representa a heap).

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

- Utilizamos *heaps* de mínimo (ou de máximo), por exemplo, nos algoritmos de Prim e de Dijkstra, para representar **filas de prioridade**. Nestes algoritmos, a chave de v é $dist[v]$. Cada nó da *heap* guarda $dist[v]$ e também v (sendo v o identificador de um nó do grafo). Na implementação disponibilizada, cada nó é do tipo `QNODE` e a fila de prioridade é do tipo `HEAPMIN`:

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

```
typedef struct heapMin {
  int sizeMax, size;
  QNODE *a;
  int *pos_a;
} HEAPMIN;
```

- Nos algoritmos de Prim e de Dijkstra, a chave de v pode variar (ser reduzida ou aumentada, conforme a aplicação). Para rapidamente aceder ao nó que tem v na heap, a estrutura `HEAPMIN` mantém um array `pos_a[]` que indica a posição de cada **nó do grafo** no vetor `a[]` (que representa a heap).
- `sizeMax` é o número máximo de elementos que a heap pode conter; `size` é o número de elementos que contém num dado momento.

Exemplo com HEAPMAX

Suponha que para uma **heap de máximo** do tipo `HEAPMAX`, apontada por q, o contéudo de `size` é 10, de `sizemax` é 12, e o conteúdo de a[1], a[2], . . . a[10], a[11], a[12] é:

7	9	5	1	12	4	8	10	6	2	11	3
15	9	13	7	9	2	7	3	3	4	14	20

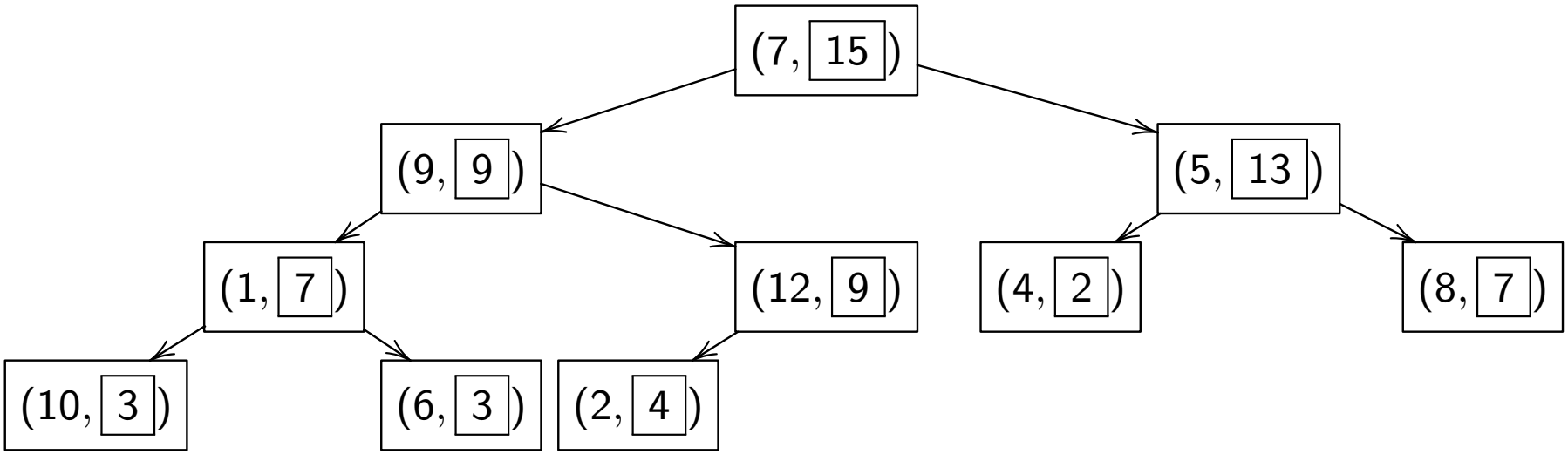
A representação da fila de prioridade (`size=10`) por uma árvore binária seria:

Exemplo com HEAPMAX

Suponha que para uma **heap de máximo** do tipo `HEAPMAX`, apontada por q, o contéudo de `size` é 10, de `sizemax` é 12, e o conteúdo de a[1], a[2], . . . a[10], a[11], a[12] é:

7	9	5	1	12	4	8	10	6	2	11	3
15	9	13	7	9	2	7	3	3	4	14	20

A representação da fila de prioridade (`size=10`) por uma árvore binária seria:



Em cada nó, a chave `vertkey` está assinalada num quadrado. O outro valor do par é `vert`.
NB: A chave tinha de ser o valor na segunda linha pois `Q.a[1]` tem a chave máxima.

`Q.pos_a[12]=5. PARENT(2)=1. RIGHT(7)=15. LEFT(3)=6. Q.a[LEFT(3)]=(4,2). Q.pos_a[11]=0=Q.pos_a[12].`

Exemplo de extração do máximo (em HEAPMAX)

Se executar `ExtractMax` a seguir, retornará 7, que é o nó do grafo em `a[1]`, e:

- troca `a[1]` com `a[10]` e reflete a extração e troca em `pos_a[.]`, ficando `pos_a[7]=0` e `pos_a[2]=1`. O conteúdo de `a[.]` passa a ser

2	9	5	1	12	4	8	10	6	7	11	3
4	9	13	7	9	2	7	3	3	15	14	20

- reduz `size` para 9 e aplica `heapify` **a partir do nó 1** para restabelecer a **"propriedade de heap"**, se necessário (como é neste caso).

Exemplo de extração do máximo (em HEAPMAX)

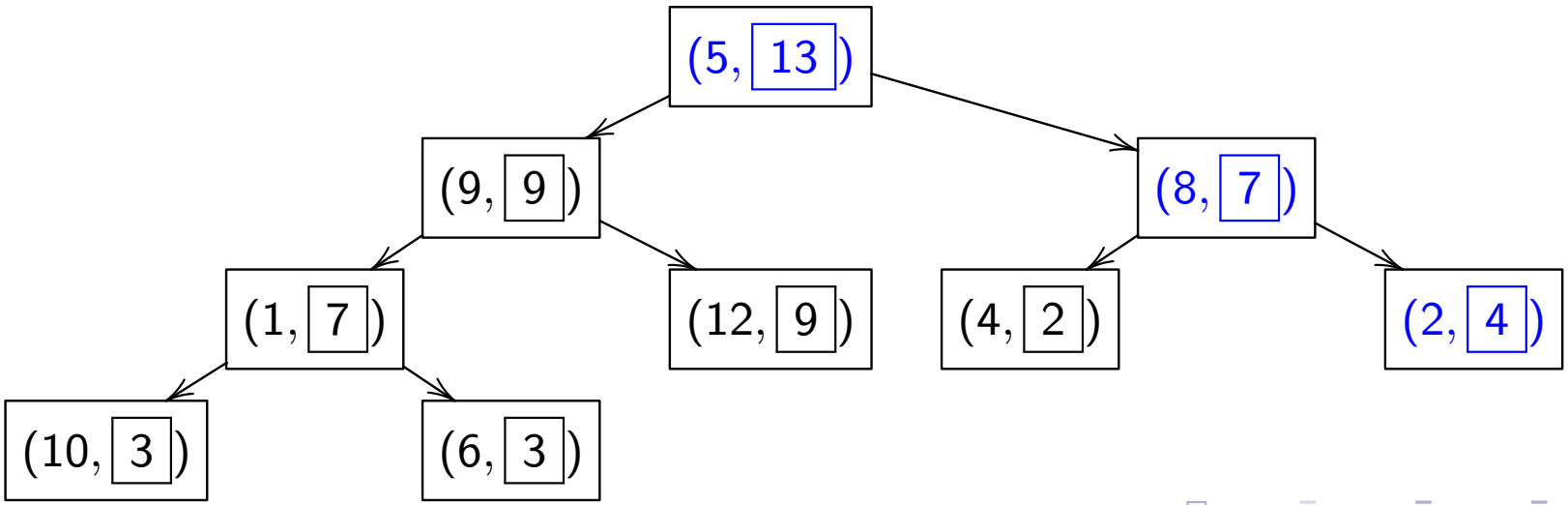
Se executar `ExtractMax` a seguir, retornará 7, que é o nó do grafo em `a[1]`, e:

- troca `a[1]` com `a[10]` e reflete a extração e troca em `pos_a[.]`, ficando `pos_a[7]=0` e `pos_a[2]=1`. O conteúdo de `a[.]` passa a ser

2	9	5	1	12	4	8	10	6	7	11	3
4	9	13	7	9	2	7	3	3	15	14	20

- reduz `size` para 9 e aplica `heapify` **a partir do nó 1** para restabelecer a **"propriedade de heap"**, se necessário (como é neste caso).

`heapify(1) -> swap(1,3) -> heapify(3) -> swap(3,7) -> heapify(7)`



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

Observar a complexidade das operações

```
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);        //  COMPLEXIDADE: O(n)
int extractMin(HEAPMIN *q);   // retorna v          COMPLEXIDADE: O(log_2 size)
void decreaseKey(int v, int newkey, HEAPMIN *q);    //  COMPLEXIDADE: O(log_2 size)
int heap_isEmpty(HEAPMIN *q);   // retorna 1 ou 0   // COMPLEXIDADE: O(1)

void insert(int v, int key, HEAPMIN *q);   // COMPLEXIDADE: O(log_2 size)
void write_heap(HEAPMIN *q);            // COMPLEXIDADE: O(size)
void destroy_heap(HEAPMIN *q);          // COMPLEXIDADE: O(1)

#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--)   // n/2 é o primeiro pai (desde o fim)
    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;
}
```

Complexidade da operação **heapify** (em HEAPMIN)

- No pior caso, o tempo de execução de `heapify(i,q)` resulta do tempo gasto para obter `smallest`, efetuar `swap(i,smallest,q)` e executar a recursão `heapify(smallest,q)`.

Complexidade da operação **heapify** (em HEAPMIN)

- No pior caso, o tempo de execução de `heapify(i,q)` resulta do tempo gasto para obter `smallest`, efetuar `swap(i,smallest,q)` e executar a recursão `heapify(smallest,q)`.
- Na recursão, analisa uma sub-árvore filha do nó i. Se a árvore que tem raíz i tiver n_i nós, **a maior das sub-árvores filhas não tem mais do que** $2n_i/3$ nós, por se tratar de heaps binárias.

Complexidade da operação **heapify** (em HEAPMIN)

- No pior caso, o tempo de execução de `heapify(i,q)` resulta do tempo gasto para obter `smallest`, efetuar `swap(i,smallest,q)` e executar a recursão `heapify(smallest,q)`.
- Na recursão, analisa uma sub-árvore filha do nó i. Se a árvore que tem raíz i tiver n_i nós, **a maior das sub-árvores filhas não tem mais do que** $2n_i/3$ nós, por se tratar de heaps binárias.
- Assim, o tempo de execução de `heapify(i,q)` pode ser descrito por $T(n_i) \leq T(2n_i/3) + c$, para alguma constante $c > 0$.

Complexidade da operação **heapify** (em HEAPMIN)

- No pior caso, o tempo de execução de `heapify(i,q)` resulta do tempo gasto para obter `smallest`, efetuar `swap(i,smallest,q)` e executar a recursão `heapify(smallest,q)`.
- Na recursão, analisa uma sub-árvore filha do nó i. Se a árvore que tem raíz i tiver n_i nós, **a maior das sub-árvores filhas não tem mais do que** $2n_i/3$ nós, por se tratar de heaps binárias.
- Assim, o tempo de execução de `heapify(i,q)` pode ser descrito por $T(n_i) \leq T(2n_i/3) + c$, para alguma constante $c > 0$.
- Daqui resulta que $T(n_i) \in O(\log n_i)$, pois a solução da recorrência $T(n) = T(2n/3) + c$ satisfaz ; $T(n) \in \Theta(\log_2 n)$.

Complexidade da operação **heapify** (em HEAPMIN)

- No pior caso, o tempo de execução de `heapify(i,q)` resulta do tempo gasto para obter `smallest`, efetuar `swap(i,smallest,q)` e executar a recursão `heapify(smallest,q)`.
- Na recursão, analisa uma sub-árvore filha do nó i. Se a árvore que tem raíz i tiver n_i nós, **a maior das sub-árvores filhas não tem mais do que** $2n_i/3$ nós, por se tratar de heaps binárias.
- Assim, o tempo de execução de `heapify(i,q)` pode ser descrito por $T(n_i) \leq T(2n_i/3) + c$, para alguma constante $c > 0$.
- Daqui resulta que $T(n_i) \in O(\log n_i)$, pois a solução da recorrência $T(n) = T(2n/3) + c$ satisfaz ; $T(n) \in \Theta(\log_2 n)$.
- $T(n_i) \in O(\log_2 n_i)$ significa que $T(n_i) \in O(h)$, sendo h a altura da árvore com raíz no nó i.

Complexidade da operação **build_heap_min**

- Qualquer chamada de `heapify` custa $O(\log n)$, se se tiver n nós na heap. Assim, é fácil concluir que `build_heap_min` é $O(n \log n)$. Mas, é possível mostrar que é $\Theta(n)$ analisando a complexidade com mais rigor.
- Uma heap com n elementos tem altura $\lfloor \log_2(n) \rfloor$ e tem no máximo $\lceil \frac{n}{2^{h+1}} \rceil$ nós com altura h. Assim, o tempo da execução de

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

pode ser caraterizado como $O\left(\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil h\right) = O(n)$.

Tal resulta de

$$O\left(\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil h\right) = O\left(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h}\right) = O(2n) = O(n)$$

pois, como $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$, se $|x| < 1$, concluimos que

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$$

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 { $pai[v] \leftarrow$ NULL; $dist[v] \leftarrow \infty$;}
2. $dist[s] \leftarrow 0$;
3. $Q \leftarrow$ Mk_PQ_Heapmin$(dist, G.V)$;
4. Enquanto (PQ_Not_Empty(Q)) fazer
5. $\quad v \leftarrow$ ExtractMin(Q);
6. $\quad$ Para cada $w \in G.Adjs[v]$ fazer
7. $\quad\quad$ Se $dist[v] + G.d(v, w) < dist[w]$ então
8. $\quad\quad\quad dist[w] \leftarrow dist[v] + G.d(v, w)$;
9. $\quad\quad\quad pai[w] \leftarrow v$;
10. $\quad\quad\quad$ DecreaseKey$(Q, w, dist[w])$;

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

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 ∈ G.V fazer { pai[v] ← NULL; dist[v] ← ∞;}
2.  dist[s] ← 0;
3.  Q ← Mk_PQ_Heapmin(dist, G.V);
4.  Enquanto (PQ_Not_Empty(Q)) fazer
5.      v ← ExtractMin(Q);
6.      Para cada w ∈ G.Adjs[v] fazer
7.          Se dist[v] + G.d(v, w) < dist[w] então
8.              dist[w] ← dist[v] + G.d(v, w);
9.              pai[w] ← v;
10.             DecreaseKey(Q, w, dist[w]);
```

Melhoramento: Sair se $dist[v] = \infty$ na linha 5. (não há caminho de s para 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

```
AlgoritmoDijkstra(G, s)
1.  | Para cada v ∈ G.V fazer { pai[v] ← NULL; dist[v] ← ∞;}
2.  | dist[s] ← 0;
3.  | Q ← Mk_PQ_Heapmin(dist, G.V);
4.  | Enquanto (PQ_Not_Empty(Q)) fazer
5.  |       v ← ExtractMin(Q);
6.  |       Para cada w ∈ G.Adjs[v] fazer
7.  |           Se dist[v] + G.d(v, w) < dist[w] então
8.  |               dist[w] ← dist[v] + G.d(v, w);
9.  |               pai[w] ← v;
10. |               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":

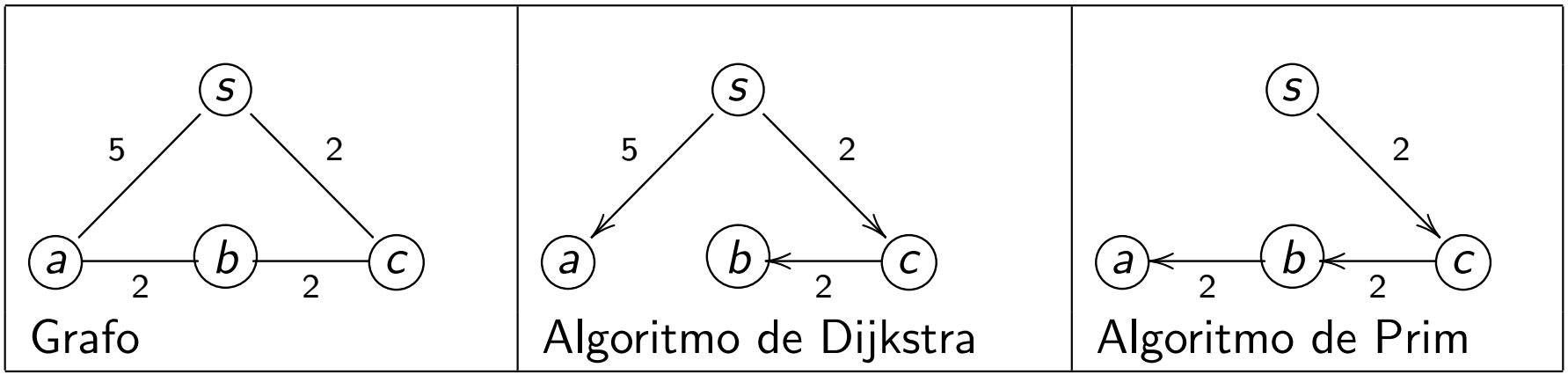
$$O(\sum_{v \in V} (1 + \log_2 |V| + |Adjs[v]| \log_2 |V|)) = O((|V| + |E|) \log_2 |V|).$$

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 $\mathcal{Q}_k$ o conjunto de nós que estão na fila Q e $\mathcal{M}_k = V \setminus \mathcal{Q}_k$ o conjunto de nós que já sairam de Q. Tem-se:

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

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 $\mathcal{Q}_k$ o conjunto de nós que estão na fila Q e $\mathcal{M}_k = V \setminus \mathcal{Q}_k$ o conjunto de nós que já sairam de Q. Tem-se:

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

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 $w \in Adjs[s]$ (e mantém $dist[r] = \infty$ para os restantes $r \neq s$). Logo, as condições 1. e 2. verificam-se, já que, $\mathcal{M}_1 = \{s\}$, $dist[s] = 0 = \delta(s, s)$, por definição, e para $r \in \mathcal{Q}_1 = V \setminus \{s\}$, 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 $k+1$.

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 $\mathcal{Q}_k$ acessíveis de s
- (Caso B) existem vértices em $\mathcal{Q}_k$ acessíveis de 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 $\mathcal{Q}_k$ acessíveis de s
- (Caso B) existem vértices em $\mathcal{Q}_k$ acessíveis de s

Caso A

Todo $r \in \mathcal{Q}_k$ 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 $r \in \mathcal{Q}_k$ (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 $r \in \mathcal{Q}_{k+1}$ 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 $w \in \mathcal{Q}_k$ um vértice que se encontra a distância mínima de s, ou seja tal que $\delta(s,w) = \min\{\delta(s,r) \mid r \in \mathcal{Q}_k\}$. Seja $\gamma_{s,w}$ um **caminho mínimo** de s para w em G. Logo, tem-se $\hat{d}(\gamma_{s,w}) = \delta(s,w)$.

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 $w \in \mathcal{Q}_k$ um vértice que se encontra a distância mínima de s, ou seja tal que $\delta(s,w) = \min\{\delta(s,r) \mid r \in \mathcal{Q}_k\}$. 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 $dist[w] = \delta(s,w) = \hat{d}(\gamma_{s,w})$.

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 $w \in \mathcal{Q}_k$ um vértice que se encontra a distância mínima de s, ou seja tal que $\delta(s,w) = \min\{\delta(s,r) \mid r \in \mathcal{Q}_k\}$. 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.

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 $w \in \mathcal{Q}_k$ um vértice que se encontra a distância mínima de s, ou seja tal que $\delta(s,w) = \min\{\delta(s,r) \mid r \in \mathcal{Q}_k\}$. 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 $k+1$.

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 $r \in Adjs[v] \cap \mathcal{M}_k$, 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.

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 $r \in Adjs[v] \cap \mathcal{M}_k$, 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.

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 $r \in Adjs[v] \cap \mathcal{M}_k$, 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:
Determinar um *percurso com capacidade máxima* de s para v, para cada $v \neq s$, para um nó origem s dado.

CaminhosCapacidadeMaxima(G, s)

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

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 $\mathcal{Q}_k$ o conjunto de nós que estão na fila Q e $\mathcal{M}_k = V \setminus \mathcal{Q}_k$ o conjunto de nós que já sairam de Q, no final da iteração k, tem-se

1. para $r \in \mathcal{M}_k$, o valor $cap[r]$ é a capacidade máxima dos percursos de s para r em G, para todo $r \in \mathcal{M}_k$;
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 $\mathcal{M}_k \cup \{r\}$.

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 $\mathcal{Q}_k$ o conjunto de nós que estão na fila Q e $\mathcal{M}_k = V \setminus \mathcal{Q}_k$ o conjunto de nós que já sairam de Q, no final da iteração k, tem-se

1. para $r \in \mathcal{M}_k$, o valor $cap[r]$ é a capacidade máxima dos percursos de s para r em G, para todo $r \in \mathcal{M}_k$;
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 $\mathcal{M}_k \cup \{r\}$.

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 $\gamma^{\star}_{sv}$ e $\gamma^{\star}_{vt}$ 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 raíz** 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** $(s, t) \in V \times V$, $s \neq t$.

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 em $O(n^3 \log_2 n)$.**

  Para cada nó v_i (origem), aplicar o algoritmo de Dijkstra para calcular $D^{\star}_{ij}$, para todo j. Complexidade: $O(|V|(|E| + |V|) \log_2 |V|)$. Para grafos densos, i.e., com $|E| \in \Theta(|V|^2)$, seria $O(n^3 \log_2 n)$.

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 em $O(n^3 \log_2 n)$.**

  Para cada nó v_i (origem), aplicar o algoritmo de Dijkstra para calcular $D^{\star}_{ij}$, para todo j. Complexidade: $O(|V|(|E|+|V|)\log_2 |V|)$. Para grafos densos, i.e., com $|E| \in \Theta(|V|^2)$, seria $O(n^3 \log_2 n)$.

- Mas, o **algoritmo de Floyd-Warshall** (1962) resolve em $\Theta(n^3)$.

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

  Na chamada: $D_{ii} = 0$, $D_{ij} = d(i,j)$, se $(i,j) \in E$; e, caso contrário $D_{ij} = \infty$.